CLASSIFICATION AND DETECTION OF COMPUTER INTRUSIONS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Sandeep Kumar

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 1995

This thesis is dedicated to my parents, and to Bharati.

ACKNOWLEDGMENTS

I would like to sincerely thank two people who have been instrumental in my obtaining a Ph.D. My wife Bharati, who got me started when the inertia and apprehension of a long-term commitment seemed insurmountable, and my advisor Dr. Eugene Spafford, for his encouragement and sagesse on several occasions when the temptation to throw it all away seemed too strong.

The COAST laboratory in which I worked provided a stimulating, benign and encouraging environment for the discussion of semi-baked ideas. There I firmly understood that mutual cooperation can serve as a catalyst to high-quality work and that the time demands thus placed on individuals are well worth it. Thanks to all its members, former and current: Taimur Aslam, Mark Crosbie, Bryn Dole, Ivan Krsul, Steve Lodin, Christoph Schuba, and Frank Wang. Special thanks to Christoph for proofreading most of my writing and giving very valuable technical and stylistic consistency feedback.

Discussions and guidance from my committee members Dr. John Korb, Dr. Samuel Wagstaff and Dr. Michal Young were very valuable. They continually steered me to the cause of "science" when I seemed hopelessly lost in the "engineering" aspect of things. Dr. Mikhail Atallah was extremely helpful in proof-reading one of my earlier reports and for pointing out that one of my theoretical observations had already been published elsewhere.

The Computer Science department of the University of California, Davis was very helpful in providing me with a facility to generate stable audit trails, without which it would have been difficult to conduct my experiments. Many thanks go to Dr. Matthew Bishop and to Christopher Wee, who championed my cause with their administrative staff.

The facilities staff at the Computer Sciences department at Purdue deserve special mention. In addition to the fundamental sustenance they provide to the department by keeping the hardware and software "well tuned," "well oiled," and up-to-date, they have been extremely prompt and cheerful while suffering through the innumerable queries that I have put to them in the course of my half decade of stay in West Lafayette.

This work was supported, in part, by: Department of Defense contract MDA 904-93-C-4081; by gifts from Sun Microsystems, Bell Northern Research, and Hughes Research Laboratories; equipment loaned to the COAST group by the U.S. Air Force; and a contract with Trident Data Systems. This support is gratefully acknowledged.

DISCARD THIS PAGE

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Figure Page

LIST OF TABLES

ABSTRACT

Kumar, Sandeep. Ph.D., Purdue University, August 1995. Classification and Detection of Computer Intrusions. Major Professor: Eugene H. Spafford.

Some computer security breaches cannot be prevented using access and information flow control techniques. These breaches may be a consequence of system software bugs, hardware or software failures, incorrect system administration procedures, or failure of the system authentication module. Intrusion detection techniques can have a significant role in the detection of computer abuse in such cases.

This dissertation describes a pattern matching approach to representing and detecting intrusions, a hitherto untried approach in this field. We have classified intrusions on the basis of structural interrelationships among observable system events. The classification formalizes detection of specific exploitations by examining their manifestations in the system event trace. Thus, we can talk about intrusion signatures belonging to particular categories in the classification, instead of vulnerabilities that result in intrusions.

The classification developed in this dissertation can also be used for developing computational models to detect intrusions in each category by exploiting the common structural interrelationships of events comprising the signatures in that category. We can then look at signatures of interest that can be matched efficiently, instead of attempting to devise a comprehensive set of techniques to detect any violation of the security policy. We define and justify a computational model in which intrusions from our classification can be represented and matched. We also present experimental results based on an implementation of the model tested against real-world intrusions.

1. INTRODUCTION

In this chapter we motivate the need for securing computer systems and discuss the role of intrusion detection in their security. We give a broad overview of the field of intrusion detection as it is presented in the literature. In the next chapter we survey approaches that have been taken in other systems for detecting intrusions.

1.1 Computer Security and its Role

One broad definition of a secure computer system is given by Garfinkel and Spafford [GS91] as one that *can be depended upon to behave as it is expected to.* The dependence on the expected behavior being the same as exhibited behavior is referred to as *trust* in the security of the computer system. The level of trust indicates the confidence in the expected behavior of the computer system. The expected behavior is formalized into the *security policy* of the computer system and governs the goals that the system must meet. This policy may include functionality requirements if they are necessary for the effective functioning of the computer system.

A narrower definition of computer security is based on the realization of *confidentiality*, *integrity*, and *availability* in a computer system [RS91]. Confidentiality requires that information be accessible only to those authorized for it, integrity requires that information remain unaltered by accidents or malicious attempts, and availability means that the computer system remains working without degradation of access and provides resources to authorized users when they need it. By this definition, an unreliable computer system is unsecure if availability is part of its security requirements.

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

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

Threats to Security

As a society we are becoming increasingly dependent on the rapid access and processing of information. As this demand has increased, more information is being stored on computers. The increased use of computers has made rapid tabulation of data from different sources possible. Correlation of information from different sources has allowed additional information to be inferred that may be difficult to obtain directly. The proliferation of inexpensive computers and of computer networks has exacerbated the problem of unauthorized access and tampering with data. Increased connectivity not only provides access to larger and varied resources of data more quickly than ever before, it also provides an access path to the data from virtually anywhere on the network [Pow95]. In many cases, such as the Internet worm attack of 1988 [Spa89], network intruders have easily overcome the password authentication mechanisms designed to protect systems.

With an increased understanding of how systems work, intruders have become skilled at determining weaknesses in systems and exploiting them to obtain such increased privileges that they can do anything on the system. Intruders also use patterns of intrusion that are difficult to trace and identify. They frequently use several levels of indirection before breaking into target systems and rarely indulge in sudden bursts of suspicious or anomalous activity. They also cover their tracks so that their activity on the penetrated system is not easily discovered[1].

Threats such as viruses [Coh87] and worms [SH82] do not need human supervision and are capable of replicating and traveling to connected computer systems. Unleashed at one computer, by the time they are discovered it may be impossible to trace their origin or the extent of infection. Then there may be threats from trojan horses which do not replicate but are programmed to unleash destructive activity on a precondition compiled into the program [Tho87].

Detecting these Threats

Most computer systems provide an access control mechanism as their first line of defense [Lam69, Lam71]. However, this only limits whether access to an object in the system is permitted but does not model or restrict what a subject may do with the object itself if it has the access to manipulate it [Den82]. Access control therefore does not model and cannot prevent unauthorized information flow through the system because such flow can take place with authorized accesses to the objects. Moreover, in systems where access controls are discretionary, the responsibility of protecting data rests on the end user. This often requires that users understand the protection mechanisms offered by the systems and how to achieve the desired security using these mechanisms.

Information flow can be controlled to enhance security by applying models such as the Bell and LaPadula model [BL73] to provide secrecy, or the Biba model [Bib77]

[1] For an account of a real intrusion that originated in Europe and targeted several military computers in the U.S. see the book by Cliff Stoll [Sto88].

to provide integrity. However, security comes at the expense of convenience. Both models are conservative and restrict read and write operations to ensure that confidentiality and integrity of data in the system cannot be compromised. If both models are jointly used, the resulting model only permits accesses to objects at the same security classification level as the subject. Thus, a completely secure system may not be very useful.

Access controls and protection models are not helpful against insider threats or compromise of the authentication module. If a password is weak and is compromised, access control measures cannot prevent the loss or corruption of information that the compromised user was authorized to access. In general, static methods of assuring security properties in a system may simply be insufficient, or make the system overly restrictive to its users. For example, static techniques may not be able to prevent violation of security policy that results from browsing of data files; and mandatory access controls [oDS85] that only permit users access to data for which they have appropriate clearance make the system cumbersome to use. A dynamic method, such as behavior tracking, is therefore needed to detect and perhaps prevent breaches in security.

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

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

1.2 What is Intrusion Detection?

An intrusion is defined by Heady et al. [HLMS90] as

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

An earlier study done by Anderson [And80] uses the term "threat" in this same sense and defines it to be

> the potential possibility of a deliberate unauthorized attempt to
> - access information,
> - manipulate information, or
> - render a system unreliable or unusable.

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

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

One of the earliest reports that outlines how intrusions may be detected by identifying "abnormal" behavior is the work by Anderson [And80]. In his influential report, Anderson presents a threat model that classifies threats as *external penetrations*, *internal penetrations*, and *misfeasance* and uses this classification to develop a security monitoring surveillance system based on detecting anomalies in user behavior. External penetrations are defined as intrusions that are carried out by unauthorized computer system users; internal penetrations are those that are carried out by authorized users of computer systems who are not authorized for the data that is compromised; and misfeasance is defined as misuse of authorized data and other resources by otherwise authorized users.

In contrast, misuse intrusion detection refers to intrusions that follow well-defined patterns of attack that exploit weaknesses in system and application software. Such patterns can be precisely written in advance. For example, exploitation of the `fingerd` and `sendmail` bugs used in the Internet Worm attack [Spa89] would come under this category. This technique represents knowledge about bad or unacceptable behavior [Sma92] and seeks to detect it directly, as opposed to anomaly intrusion detection, which seeks to detect the complement of normal behavior.

The above mentioned schemes of classifying intrusions are based on their method of detection. Another classification scheme, based on intrusion types, presented by Smaha [Sma88] classifies intrusions into the following six types:

Attempted break-in: often detected by atypical behavior profiles or violations of security constraints.

Masquerade attack: often detected by atypical behavior profiles or violations of security constraints.

Penetration of the security control system: usually detected by monitoring for specific patterns of activity.

Leakage: often detected by atypical usage of I/O resources.

Denial of Service: often detected by atypical usage of system resources.

Malicious use: often detected by atypical behavior profiles, violations of security constraints, or use of special privileges.

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

1.2.1 Premise and Limitations of Intrusion Detection

Anomaly Detection

The central premise of anomaly intrusion detection is that intrusive activity is a subset of anomalous activity. This might seem reasonable, considering that if an outsider breaks into a computer account with no notion of the compromised user's pattern of resource usage, there is a good chance that his behavior will be anomalous.

Often, however, intrusive activity can be carried out as a sum of individual activities, none of which is independently anomalous. Ideally, the set of anomalous activities is the same as the set of intrusive activities. Then, flagging all anomalous activities exactly flags all intrusive activities, resulting in no false positives or false negatives. However, intrusive activity does not always coincide with anomalous activity. There are four possibilities, each with a non-zero probability:

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

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

3. *Not intrusive* and *not anomalous*. These are true negatives: the activity is not intrusive and is not reported as intrusive.

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

When false negatives are not desirable, thresholds that define an anomaly are set low. This results in many false positives and detracts from the efficacy of automated mechanisms for intrusion detection. It creates additional burdens for the security officer as well, who must investigate each incident and discard many.

Anomaly detectors also tend to be computationally expensive because several metrics are often maintained that need to be updated against every system activity.

Misuse Detection

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

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

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

This approach also assumes the integrity of the event data. Thus, attacks that involve spoofing, which produce the same events but from a different source, cannot be reliably detected.

1.3 Terminology

This section explains several terms used throughout the dissertation. Some of the terms have well-accepted definitions among security professionals, while others have been used in a specific way in this dissertation. For consistency, all cited definitions have been taken from the Dictionary of Data and Computer Security [LS87]. When explaining a term, references to other terms that are defined in this section have been italicized.

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

Audit trail/Event stream. An audit trail is defined in [LS87] as a chronological record of system activities that is sufficient to enable the reconstruction, review and examination of the sequence of environments and activities surrounding or leading to each event in the path of a transaction from its inception to output of final results.

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

C2 security rating of computer systems. A Department of Defense security evaluation criteria class requiring auditing and protection of encrypted passwords, among others, as described in the Orange Book [oDS85]. The primary motivation behind the Orange Book was the need to quantify security and trust, because different organizations and different types of information require different types of security [RS91]. Briefly, the Orange Book defines four categories of security protection: `D` – minimal security, `C` – discretionary protection, `B` – mandatory protection, and `A` – verified protection. Each class requires a specific set of criteria to be met by computer systems in that category.

Exploitation. An exploitation is a set of actions that result in a violation of the security policy of a computer system. Intruders exploit system *vulnerabilities* or *flaws* to gain unauthorized access to the system. These exploitations can often be encoded as *signatures* that can be matched against the *audit trail* to detect them.

Flaw. A flaw is defined in [LS87] as an error of commission, omission or oversight in a system that allows protection mechanisms to be bypassed. We use vulnerabilities and flaws synonymously.

Matching model. This refers to the computational framework in which *signatures* are encoded and matched against the *audit trail.* In this dissertation, when we refer to "our matching model," we are referring to the computational framework presented in Chapter 4.

Security policy. A security policy is defined in [LS87] as the set of laws, rules, and practices that regulate how an organization manages, protects and distributes sensitive information .

Signature. In the context of misuse intrusion detection, a signature is the specification of features, conditions, arrangements and interrelationships among *events* that signify a break-in or other misuse, or their attempt. "Patterns" and "intrusion patterns" are used in the same sense as a signature.

Vulnerability. A vulnerability is defined in [LS87] as a weakness in automated system security procedures, administrative controls, internal controls etc. that could be exploited by a threat to gain unauthorized access to information or to disrupt critical processing. Anderson [And80] defines a vulnerability in a less abstract way as a known or suspected flaw in the hardware or software design or operation of a system that exposes it to penetration of its information.

1.4 A Note on the Use of Examples

The basic goal of all operating systems is to provide a *convenient* and *efficient* interface to computer system resources [SG94]. In so doing, they partition the set of services exported to the user in similar ways, even though the details may differ. Generic studies of operating system flaws, such as those done by Linde [Lin75] and Landwehr et al. [LBMC93] have shown striking similarities among operating system vulnerabilities. In each categorization of their study, examples have been drawn from several operating systems. If operating systems have similar vulnerabilities, and offer similar user visible resource abstractions, then the methods of exploiting these vulnerabilities are also likely to be similar.

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

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

1.5 Thesis Statement and Outline

This dissertation provides an answer to the question: *Can we usefully and effectively detect computer intrusions by applying pattern matching techniques?* A more complete statement of the thesis is:

> *It is possible to classify a large subset of currently known computer security exploitations in a simple classification scheme based on the time complexity of detecting the vulnerabilities. A single computational model can be used to represent and monitor exploitations in all the categories using pattern matching techniques.*

This dissertation applies pattern matching to intrusion detection. It offers a view of computer security breaches not from their origin or intended effect, but from their manifestation in the system activity trace. Previous researchers[2] have looked at computer vulnerabilities from the viewpoint of cataloging and classifying them so that the classification can provide a useful feedback to software engineers. By being aware of the nature and statistics of flaws at different stages of the software life cycle, engineers can take efforts to minimize their occurrence. Work has also been done to use pattern directed approaches to detect vulnerabilities in source code, for example, in the RISOS project [A+76].

Whereas published literature contains analyses of vulnerabilities in terms of their origin and their possible prevention, we have focused on the *runtime exploitation* of vulnerabilities. Thus, we use the term *signature*, or *intrusion* instead of vulnerabilities to denote entities populating our classification scheme. Using this scheme, new intrusions can be understood and characterized in terms of the structure of events needed to detect them. This classification scheme is presented in Section 3.1.

[2] A good description of their work can be found in the study by Aslam [Asl95].

To represent and detect computer intrusions efficiently we have devised a model. This model uses the classification scheme of Section 3.1 to group intrusions and instantiates the generic requirements that we propose and defend in this thesis. These generic requirements must be addressed by all computer intrusion detectors that use a pattern matching approach. Our model, presented in Chapter 4, answers two key questions of the intrusion detection problem: (1) *How do we effectively represent computer intrusions in a generic fashion?* and (2) *How do we monitor for their occurrence?* The model is based on Colored Petri Nets and uses a modified net to represent intrusion scenarios. Detection of intrusions is posed as an acceptance problem in the model.

We show the effectiveness of the pattern matching approach by building a prototype of our model in C++ that detects intrusions on a system where the security audit trail is generated. The prototype is structured as a library that can be embedded in application programs. We have designed a simple syntax to represent intrusions and a compiler that translates these descriptions into C++ code that realizes the matching behavior of the patterns. The software architecture of the prototype and simulation results are presented in Chapter 6.

1.6 Summary

Intrusion detection is an important component of the security controls and mechanisms provided in a system. It usually forms the last line of defense against security threats. These mechanisms are intended to detect breaches of policy that cannot be easily detected using other methods. Intrusion detection is usually based on one of two models: the *anomaly* and the *misuse* model. Both models make assumptions about the nature of intrusive activity that can be detected. This dissertation proposes a classification scheme of intrusions based on their manifestation in system events and applies pattern matching techniques to represent and detect them.

2. RELATED WORK IN INTRUSION DETECTION

This chapter describes the architecture of several prior intrusion detection systems. None of them uses pattern matching directly to represent and detect intrusions. We also describe the generic model of intrusion detection proposed by Dorothy Denning [Den87], which is still accurate as an abstract model of most intrusion detection systems.

In Section 3.2 we present the requirements of a pattern matching solution to any intrusion detector that uses pattern matching to detect intrusions.

2.1 Introduction

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

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

2.2 Anomaly Intrusion Detection

In this section we discuss systems and techniques that base their decision on the variance of predicted or expected behavior from observed behavior. These techniques do not base their decision on the occurrence of specific fixed activities.

2.2.1 Statistical Approaches

The following, based on NIDES [LTG+92], serves to illustrate the generic process of anomaly detection, which is primarily statistical in nature. The anomaly detector observes the activity of subjects and generates profiles for them that represent their behavior. These profiles are designed to use little memory to store their internal state, and to be efficient to update because every profile may potentially be updated for every audit record.

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

$$a_1 S_1^2 + a_2 S_2^2 + \cdots + a_n S_n^2, \quad a_i > 0$$

where a_i reflects the relative weight of the metric M_i. In general, the measures M_1, $M_2, \ldots, M_n$ may not be mutually independent, and may require a more complex function for combining them. Anomaly measures are just numbers without a well-defined theoretical basis for combining them. For example, using multiplication of independent anomaly measures as a basis of combination is theoretically sound in *likelihood* computations, but the relationship between anomaly measures and Bayesian likelihood numbers is not clear.

There are several types of measures comprising a profile, which include:

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

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

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

4. Ordinal Measures — These measure activity whose outcome is a numeric value, such as the amount of CPU and I/O used by a particular user. While categorical measures count the 'number' of times an activity occurred, ordinal measures compute statistics on the numerical value of the activity outcome.

The current behavior of each user is maintained in a profile. At regular intervals the current profile is merged with the stored profile[1]. Anomalous behavior is determined by comparing the current profile with the stored profile.

2.2.1.1 Pros and Cons of Statistical Intrusion Detection

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

Statistical intrusion detection systems also have several disadvantages:

[1]This is true of NIDES [LTG+92], but in some systems the profiles do not change once determined.

- Statistical measures are insensitive to the order of occurrence of events. That is, a purely statistical intrusion detection system may miss intrusions that are indicated by sequential interrelationships among events.
- Purely statistical intrusion detection systems can be trained gradually to a point where behavior, once regarded abnormal, is considered normal. Intruders who know that they are being monitored by anomaly detectors can train such systems. Thus, most existing intrusion detection schemes combine both a statistical part to measure aberration of behavior, and a misuse part that monitors the occurrence of specific patterns of events.
- It is difficult to determine thresholds above which an anomaly should be considered intrusive. Setting a threshold too low results in false positives and setting it too high results in false negatives.
- There is a limit to the types of behaviors that can be modeled using purely statistical methods. Application of statistical techniques to the formulation of anomalies requires the assumption that the underlying data comes from a quasi-stationary process, an assumption that may not always hold. More accurate models such as generalized Markov chains are more complex and time-consuming to build.

2.2.2 Feature Selection

A difficult problem in anomaly intrusion detection is determining the correspondence between anomalous activity and intrusive activity. Given a set of heuristically chosen measures that can have a bearing on detecting intrusions, the subset that accurately predicts or classifies intrusions has to be determined. This is called feature selection. Determining the right measures is complicated because the appropriate subset of measures depends on the types of intrusions being detected. One set of measures will not likely be adequate for all types of intrusions. Predefined notions of the relevance of particular measures to detect intrusions might miss intrusions unique to a particular environment. The set of optimal measures for detecting intrusions must be determined dynamically for best results.

Consider an initial list of n measures as potentially relevant to predicting intrusions. The number of possible subsets of these n measures, which is the power set of these measures, is 2^n. Because the search space is exponentially related to the number of measures, an exhaustive search for the optimal subset of measures is not efficient. Maccabe et al. present a genetic approach to searching through this space for the right subset of metrics [HLMS90]. Using a learning classifier scheme they generate an initial set of measures which is refined in the *rule evaluation* mode using genetic operators of crossover and mutation. Subsets of the measures under consideration having low predictability of intrusions are weeded out and replaced by applying genetic operators to yield stronger measure subsets. The method assumes that combining higher predictability measure subsets allows searching the space of metrics more efficiently than other heuristic techniques.

For a survey of other feature selection techniques see Doak [Doa92].

2.2.3 Combining Individual Anomaly Measures to Get a Single Measure

If we assume that the right set of anomaly metrics can somehow be determined, how do we then combine the anomaly values of all the metrics to get a single number? One method is to use Bayesian statistics, applied either from first principles or through belief networks. Another approach, used in NIDES [LTG+92], is to combine them using covariant matrices.

2.2.3.1 Bayesian Statistics

Let $A_1, A_2, \ldots, A_n$ be n measures used to determine if an intrusion is occurring on a system at any given moment. Each A_i measures a different aspect of the system, such as the amount of disk I/O activity, or the number of page faults in the system. Let each measure A_i have two values, 1 implying that the measure is anomalous, and 0 otherwise. Let I be the hypothesis that the system is currently undergoing an intrusive attack. The reliability and sensitivity of each anomaly measure A_i is determined by the numbers $P(A_i = 1|I)$ and $P(A_i = 1|\neg I)$. The combined belief in I given the values of each A_i, is given by Bayes' theorem as:

$$P(I|A_1, A_2, \ldots, A_n) = P(A_1, A_2, \ldots, A_n|I)\frac{P(I)}{P(A_1, A_2, \ldots, A_n)}$$

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

$$P(A_1, A_2, \ldots, A_n|I) = \Pi_{i=1}^{n} P(A_i|I)$$

and

$$P(A_1, A_2, \ldots, A_n|\neg I) = \Pi_{i=1}^{n} P(A_i|\neg I)$$

which leads to

$$\frac{P(I|A_1, A_2, \ldots, A_n)}{P(\neg I|A_1, A_2, \ldots, A_n)} = \frac{P(I)}{P(\neg I)}\frac{\Pi_{i=1}^{n} P(A_i|I)}{\Pi_{i=1}^{n} P(A_i|\neg I)}$$

That is, we can determine the odds[2] of an intrusion given the values of various anomaly measures, from the prior odds of the intrusion and the likelihood of each measure being anomalous when an intrusion is occurring, i.e., the terms $\frac{P(A_i|I)}{P(A_i|\neg I)}$.

To derive a more realistic estimate of $P(I|A_1, A_2, \ldots, A_n)$, however, we must take the interdependence of the various measures A_i into account.

2.2.3.2 Covariance Matrices

NIDES [LTG+92] uses covariance matrices to account for the interrelationships among measures. If the measures $A_1, \ldots, A_n$ are represented by the vector A, then the compound anomaly measure is determined by

$$A^T C^{-1} A$$

[2] $\text{odds}(A) = \frac{P(A)}{P(\neg A)}$.

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

2.2.3.3 Belief Networks

Future systems might use Bayesian or other belief networks to combine anomaly measures. Bayesian networks [Pea88] allow the representation of causal dependencies between random variables in graphical form and permit the calculation of the joint probability distribution of the random variables by specifying only a small set of probabilities that relate only to neighboring nodes. This set consists of the prior probabilities of all the root nodes (nodes without parents) and the conditional probabilities of all the non-root nodes given all possible combinations of their direct predecessors. Bayesian networks, which are DAGs with arcs representing causal dependence between the parent and the child, permit absorption of evidence when the values of some random variables become known, and provide a computational framework for determining the conditional values of the remaining random variables, given this evidence.

As an example, consider the trivial Bayesian network model of an intrusion shown in Figure 2.1.

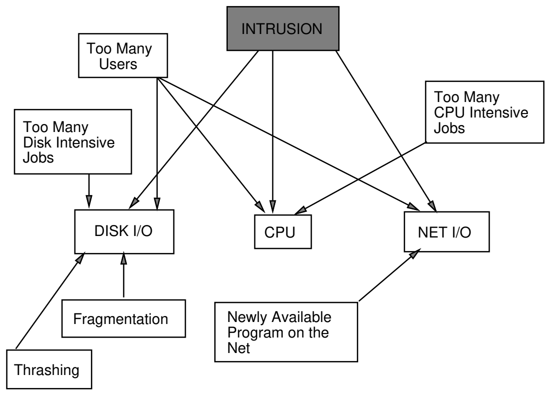

Figure 2.1 A Trivial Bayesian Network Modeling Intrusive Activity

Each box represents a binary random variable with values representing either its normal or abnormal condition. If we can observe the values of some of these variables, we can use Bayesian network calculus to determine $P(\text{Intrusion}|\text{Evidence})$. However, in general it is not trivial to determine the *a priori* probability values of the root nodes and the link matrices for each directed arc. For a good introduction to Bayesian Networks see the article by Charniak [Cha91].

2.2.4 Predictive Pattern Generation

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

The approach of time-based inductive generalization described by Teng and Chen [Che88, TCL90] uses time-based rules that characterize the normal behavior patterns of users. The rules, generated inductively, are modified dynamically during the learning phase and only "good" rules, i.e., rules with a high accuracy of prediction and a high level of confidence remain in the system. A rule has high accuracy of prediction if it is correct most of the time, and it has a high level of confidence if it can be successfully applied many times in observed data. An example of a rule generated by TIM [TCL90] may be

$$\text{E1} \rightarrow \text{E2} \rightarrow \text{E3} \Rightarrow (\text{E4} = 95\%,\ \text{E5} = 5\%)$$

where E1—E5 are security events. This rule, which is based on previously observed data, says that for the pattern of observed events E1 followed by E2 followed by E3, the probability of seeing E4 is 95% and that of E5 is 5%. TIM can generate more general rules that incorporate temporal relationships among events.

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

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

The strengths claimed for this approach are:

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

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

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

2.2.5 Neural Networks

The basic approach here is to train the neural net on a sequence of information units [FHRS90] (from here on referred to as *commands*), each of which may be at a more abstract level than an audit record. The input to the net consists of the current command and the past w commands; where w is the size of the window of past commands that the neural net takes into account in predicting the next command. Once the neural net is trained on a set of representative command sequences of a user, the net constitutes the profile of the user, and the fraction of incorrectly predicted next events then measures, in some sense, the variance of the user behavior from his profile. A conceptual diagram depicting the use of neural nets is shown in Figure 2.2. The arrows directed at the input layer form the sequence of the last w commands issued by the user. Every input in this idealized representation encodes several values or levels, each of which uniquely identifies a command. Thus the values of the inputs at the input layer correspond exactly to the sequence of the last w commands. The output layer conceptually consists of a single multi-level output that predicts the next command to be issued by the user.

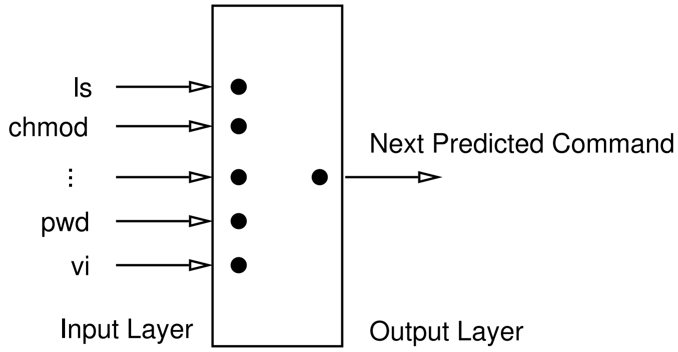

Figure 2.2 A Conceptual Use of Neural Nets in Intrusion Detection

For a good introduction to neural networks and learning in neural nets by back propagation see the book by Winston [Win92].

Some of the drawbacks of this approach are:

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

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

Some advantages of this approach are:

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

2. Neural nets cope well with noisy data.

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

2.2.6 Bayesian Classification

Bayesian classification, described by Cheeseman [CHS91], is a technique of unsupervised classification of data. Its implementation, Autoclass [CKS+88], searches for classes in the given data using Bayesian statistical techniques. This technique attempts to determine the most likely processes that generate the data. It does not partition the given data into classes but defines a probabilistic membership function of each datum in the most likely determined classes. Some advantages of this approach are:

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

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

3. Continuous and discrete attributes may be freely mixed.

In statistical intrusion detection we are concerned with a classification of observed behavior. Techniques used till now have concentrated on supervised classification in which user profiles are created based on each user's observed behavior. The Bayesian classification method would permit the determination of the optimal number of classes (probablistically computed), grouping users with similar profiles, and thus yielding a natural classification of a set of users.

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

2.3 Misuse Intrusion Detection

Misuse intrusion detection refers to the detection of intrusions by precisely defining them ahead of time and watching for their occurrence. There is a misuse component in most intrusion detection systems because statistical techniques alone are not adequate to detect all types of intrusions. The limitations of statistical anomaly detectors are outlined in Section 2.2.1.1.

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

A misuse intrusion detector that simply flags intrusions based on the pattern of input events assumes that the state transition of the system (computer) leads to a compromised state when exercised with the intrusion pattern, regardless of the initial state of the system. Therefore, simply specifying an intrusion signature without the beginning state specification is sometimes insufficient to capture an intrusion scenario fully. For a security model definition of an intrusion and a pattern oriented approach to its detection, see also Gligor and Shieh [SG91].

In the following sections we describe the various approaches to misuse detection.

2.3.1 Using Conditional Probability to Predict Misuse Intrusions

This method of predicting intrusions is similar to the one outlined in Section 2.2.3.1 except that the "evidence" is now a sequence of external events rather than values of anomaly measures. For misuse intrusion detection we are interested in determining the conditional probability

$$P(\text{Intrusion}|\text{Event Pattern})$$

Applying Bayes law, as before, to the above equation, we get

$$P(\text{Intrusion}|\text{Event Pattern}) = P(\text{Event Pattern}|\text{Intrusion})\frac{P(\text{Intrusion})}{P(\text{Event Pattern})} \quad (2.1)$$

Consider the campus network of an university as the domain within which the conditional probability of intrusion is to be predicted. A security expert associated with the campus wide network might be able to quantify the prior probability of occurrence of an intrusion on the campus system, or $P(\text{Intrusion})$, based on his experience. Further, if the intrusion reports from all of the campus systems are tabulated, one can determine, for each type of event sequence comprising an intrusion, its $P(\text{Event Sequence}|\text{Intrusion})$. The relative frequency of occurrence of the event sequence in the entire intrusion set gives this probability. Similarly, given a set of intrusion-free audit trails, one can determine, by inspection and tabulation, the probability $P(\text{Event Sequence}|\neg\text{Intrusion})$. Given the two conditional probabilities, one can easily determine the left hand side of Equation 2.1 above from simple Bayesian arithmetic because the prior probability of an event sequence is

$$P(\text{Event Sequence}) = (P(ES|I) - P(ES|\neg I)) \cdot P(I) + P(ES|\neg I)$$

where ES stands for event sequence and I stands for intrusion.

2.3.2 Production/Expert Systems in Intrusion Detection

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

An example of the use of such systems in intrusion detection is described by Snapp and Smaha [SS92]. This system encodes knowledge about attacks as `if-then` implication rules in CLIPS [Gia92] and asserts facts corresponding to audit trail events. Rules are encoded to specify the conditions requisite for an attack in their `if` part. When all the conditions on the left side of a rule are satisfied, the actions on the right side are performed.

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

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

   (a) No inbuilt or natural handling of sequential order of data. That is, the working memory elements (fact base) that match the left sides of productions to determine eligible rules for firing are not recognized by the system to be sequential. Furthermore, the left side of a production rule specifies that its elements are connected with the AND relation. To match a natural ordering of facts within this framework, the Rete match procedure [For82] tests the ordering constraints for every eligible pair after the sets of working elements conforming to the left side of the production have been generated.

   (b) The expertise incorporated in the production/expert system is only as good as that of the security officer whose skills are modeled, which may not be comprehensive [Lun93]. This is a practical consideration, and is probably a concern at the lack of a concerted effort on the part of security experts to attempt to distill their knowledge into a comprehensive security rule set. However, if rule sets need to be tailored and optimized for individual environments, then it might not be possible to circumvent this limitation.

   (c) This technique can only detect known vulnerabilities.

   (d) There are software engineering concerns in the maintenance of the knowledge base [Lun93]. That is, additions and deletions of rules in the rule set must take the interactions of the changes with the rest of the rule set into consideration.

2. to combine various intrusion measures and construct a cohesive picture of intrusions – do uncertainty reasoning. The limitations of production systems that use uncertainty reasoning are well-known. See the book by Judea Pearl [Pea88] for a good description. Also see Section 2.3.5 for a list of these limitations.

2.3.3 State Transition Analysis

In this approach, taken in STAT [PK92] and implemented for UNIX in USTAT [Ilg92], attacks are represented as a sequence of state transitions of the monitored system. States in the attack pattern correspond to system states and have boolean assertions associated with them that must be satisfied to transit to that state. Successive states are connected by arcs that represent the events required for changing state. The types of allowable events are built into the model and need not correspond one-to-one with audit records. Attack patterns can only specify a sequence of events so more complex ways of specifying events are not permitted. Furthermore, there is no general purpose mechanism to prune partial matches of attacks other than through assertion primitives built into the model.

2.3.4 Keystroke Monitoring

This technique utilizes user keystrokes to determine the occurrence of an attack. The primary means is to pattern match for specific keystroke sequences that indicate an attack. The disadvantages of this approach are the lack of reliable mechanisms for user keystroke capture without operating system support, and the myriad ways of expressing the same attack at the keystroke level. Furthermore, without a semantic analysis of the keystrokes, aliases provided in user shells such as the Korn shell [BK89] can easily defeat this technique. User login shells often provide the facility of associating parameterized shorthand names for command sequences. These are called aliases and are similar to macro definitions. Because this technique only analyzes keystrokes, automated attacks that are a result of malicious program executions cannot be detected.

2.3.5 Model-Based Intrusion Detection

This approach was proposed by Garvey and Lunt [GL91] and is a variation on misuse intrusion detection that combines models of misuse with evidential reasoning to support conclusions about the occurrence of a misuse. There is a database of attack scenarios, each of which comprises a sequence of behaviors making up the attack. At any given moment, a subset of these attack scenarios are considered as the likely ones by which the system might currently be under attack. An attempt is made to verify these scenarios by seeking information in the audit trail to substantiate or refute them (this process is termed in [GL91] as the *anticipator*.) The anticipator generates the next set of behaviors to be verified in the audit trail, based on the current active models, and passes these sets to the *planner*. The planner determines how the hypothesized behavior is reflected in the audit data and translates it into a system dependent audit trail match. This mapping from behavior to activity must be such as to be easily recognized in the audit trail, and must have a high likelihood of appearing in the behavior. That is to say

$$\frac{P(\text{Activity}|\text{Behavior})}{P(\text{Activity}|\neg\ \text{Behavior})}$$

must be large.

As evidence for some scenarios accumulates, while for others the evidence drops, the list of active models is updated. The evidential reasoning calculus built into the system permits one to update the likelihood of occurrence of the attack scenarios in the active models list.

The advantages of model-based intrusion detection are:

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

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

3. The *planner* provides independence of representation from the underlying audit trail representation.

The disadvantages of model-based intrusion detection are:

1. This approach places additional burden on the person creating the intrusion detection model to assign meaningful and accurate evidence numbers to various parts of the graph representing the model.

2. The runtime efficiency of this approach has not been demonstrated by building a prototype. It is not clear from the model description how behaviors can be compiled efficiently in the planner and the effect this will have on the runtime behavior of the detector.

Model-based intrusion detection does not replace the statistical anomaly portion of intrusion detection systems, but complements it. For a thorough treatment of reasoning in the presence of uncertainty see the book by Judea Pearl [Pea88].

2.4 A Generic Intrusion Detection Model

Dorothy Denning, in 1987, established a model of intrusion detection independent of the system, type of input, and the specific intrusions to be monitored [Den87]. A brief description of the generic model is helpful in relating specific examples of intrusion detection systems presented in earlier sections to the model and viewing how these systems fit into or enhance it. The model is still accurate in describing the architecture of many current systems.

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

The Activity Profile can also generate new profiles dynamically for newly created subjects and objects based on pattern templates. If new users are added to the system, or new files created, these templates instantiate new profiles for them. The Activity Profile can also generate anomaly records when some statistical variable takes on an anomalous value, for example when `NumErrs` takes on an inordinately high value. The Rule Set represents a generic inferencing mechanism and uses event records, anomaly records, and time expirations, among others, to control the activity of the other components and to update their state. Denning [Den87], however, uses rule-based systems to explain the inferencing mechanism and the nature of interaction with the other components.

Comparison with Other Systems

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

- How the rules comprising the Rule Set are determined.
- Whether the rule set is coded *a priori* or if it can adapt and modify itself depending on the type of intrusions.
- The nature of interaction between the Rule Set and the Activity Profile.

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

However, some newer techniques of anomaly detection do not map well into the internal details of the Activity Profile. For example, the neural net approach of anomaly detection does not easily fit the framework of smart variables and the calculation of a number for an anomaly value. Learning and adaptation of rule sets and profiles is not modeled well. It is also not clear in which module TIM [TCL90] would be placed. TIM detects behavioral anomalies and therefore might be a candidate for being placed in the Activity Profile, but it does so by generating rules and firing them when conditions in the `if` part of the rules is satisfied, which also makes it a candidate for being part of the Rule Set. Very recent approaches like model-based approaches are too different to fit this framework directly.

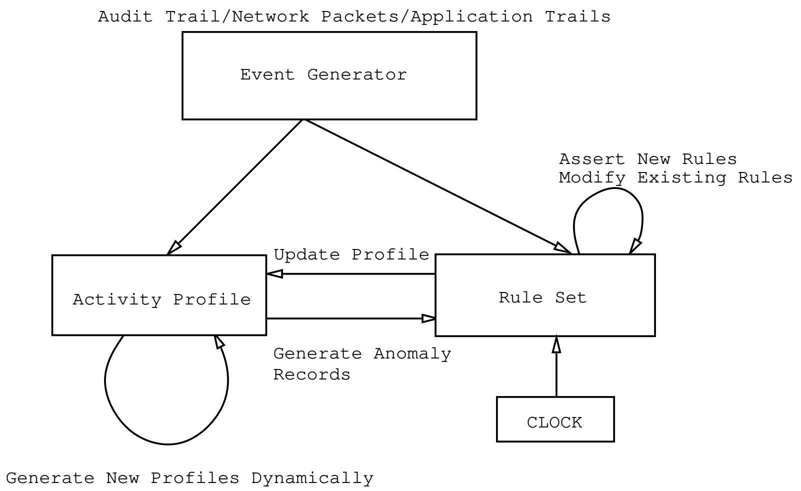

Figure 2.3 A Generic Intrusion Detection Model

2.5 Shortcomings of Current Intrusion Detection Systems

The following is a commentary on the weaknesses of intrusion detection systems taken as a whole. Different implementations rate differently along these axes of comparison. Our approach offers a new approach to building misuse detectors that is efficient and easy to maintain. Performance results for our prototype implementation are described in Chapter 6 and coverage results are presented in Chapter 7.

No Generic Building Methodology. In general, the cost of building an intrusion detection system from scratch is substantial. This is because of the lack of a structured methodolgy for building these systems. No such structuring insights have emerged from the field itself. This may partly be a result of a lack of common agreement on the techniques for detecting intrusions and partly because intrusion detection is a young field of study, initiated by Anderson in 1980 [And80].

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

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

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

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

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

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

2.6 Summary of Intrusion Detection Techniques

Several intrusion detection systems have been proposed and implemented. Most of them derive from the statistical intrusion detection model of Dorothy Denning [Den87]. Some of them, for example NIDX [BK88], Haystack [Sma88], IDES [LJL+89], MIDAS [SSHW88], Wisdom and Sense [LV89] and CMDS [Pro94] use the audit trail generated by a C2 or higher rated computer, for input. Others, for example NICE [MMA, HLMS90] and NSM [HLM91] try to analyze intrusions by analyzing network connections and the flow of information in a network. Others still, such as DIDS [SBD+91] have expanded the scope of detection by distributing anomaly detection across a heterogeneous network and centrally analyzing partial results of these distributed sources to detect potential intrusions that may be missed by the individual analysis of each source.

Among non-statistical approaches to intrusion detection is the work by Teng [TCL90] that analyzes individual user audit trails and attempts to infer the sequential relationships between events; and the neural net modeling of behavior by Simonian et al. [FHRS90].

Approaches to misuse intrusion detection include language-based approaches to represent and detect intrusions such as ASAX [HCMM92], developing an Application Programming Interface, i.e., a set of library function calls employed for representing and detecting intrusions, such as in STALKER [Sma95], expert systems such as MIDAS [SSHW88] and NIDX [BK88], and high level state machines to encode and match signatures such as STAT [PK92] and USTAT [Ilg92].

A promising approach for future intrusion detection systems might involve Bayesia-n classification, currently implemented in Autoclass [CKS+88, CHS91]. Audit trail reduction and browsing is described by Wetmore [Wet93] and Moitra [Moi], while a non-parametric pattern recognition technique is discussed by Lankewicz [Lan92]. Audit trail reduction techniques permit the compression of audit data into coarser, more abstract events, that may be queried later by the security officer to retrieve information rapidly and efficiently. Non-parametric techniques for anomaly detection have the advantage that they make no assumptions about the statistical distribution of the underlying data, and are useful when such assumptions do not hold.

3. A SCHEME FOR CLASSIFYING INTRUSION SIGNATURES

The goal of intrusion detection by examining the audit trail is to determine when a computer system has entered, or is likely to enter, a faulty or intruded state. This chapter focuses on "examination" of the audit trail in the context of misuse intrusion detection by showing how common features of the examination process can be used to categorize intrusion signatures.

In the first part of this chapter we introduce an abstract hierarchy for classifying intrusion signatures based on the structural interrelationships among the events that compose the signature. These structural interrelationships are defined over high-level events or activities, which are themselves defined in terms of low level audit trail events. The abstract hierarchy can be instantiated into a concrete hierarchy by precisely defining a high-level event in terms of low level audit trail events. An instantiated hierarchy permits the determination of precise theoretical complexity bounds of matching signatures in each level of the hierarchy.

The abstract hierarchy presented here does not classify security vulnerabilities. Instead, it classifies signatures that are used to detect the exploitation of vulnerabilities. In the latter part of the chapter we describe how pattern matching can be used to "examine" the audit trail for the occurrence of these signatures. We show how traditional pattern matching is inadequate to represent and match intrusion signatures by presenting the requirements that a framework of pattern matching solution must provide to represent intrusion signatures.

We instantiate the abstract hierarchy by defining a "high-level event" as a DAG and combining it with the pattern matching requirements, into a model of matching. This model, presented in Chapter 4, can represent signatures in each level of the hierarchy. A prototype implementation of the model is presented in Chapter 6.

3.1 A Hierarchy of Intrusion Signatures

We define the hierarchy of intrusion signatures, which is partitioned on the structural interrelationships among the elements of the signatures, in terms of high-level events. The benefits of this hierarchy are:

- It permits a classification of signatures based on common characteristics that are abstracted from any particular way of specifying and matching them. Such a classification provides conceptual benefits of understanding intrusion signatures. By specifying that a signature belongs to a particular class, it conveys the structural interrelationships of high-level events that comprise the signature. Thus, instead of referring to an exploitation as a race condition attack involving the `exec` system call, one might refer to it as a *sequence* pattern. This abstracts details of the exploitation by removing mention of `exec`, the temporal nature of the attack, or other system dependent properties.

  As a consequence of partitioning based on the structural interrelationship among high-level events, seemingly intuitive partitions based on temporal characteristics of attacks are subsumed under categories in this hierarchy. Our partition does not treat any attribute of events specially and temporal characteristics are treated as properties of a particular "time" attribute.

- It offers another way to partition intrusions based on what can be precisely monitored as opposed to how it is monitored. The traditional way intrusions have been partitioned has been to group them based on a technique for detection. This has resulted in the generic approaches of "anomaly" and "misuse" detection. By treating intrusion detection as signature matching, intrusions can be classified based on the manifestations of their exploitation in the audit trail. Many intrusions that are detected as anomalies, such as an unusual number of failed logins, can be represented and matched as signatures.

- If the patterns that we are interested in modeling for intrusion detection possess common characteristics, exploiting these characteristics might result in more efficient matching solutions.

3.1.1 Classify Vulnerabilities or Signatures?

As mentioned earlier, our hierarchy classifies signatures rather than vulnerabilities. In this section we justify why this is more meaningful from the viewpoint of detection.

Cataloging of software bugs has been of keen interest to software engineers. The rationale for this interest is summarized by Bezier [Bez83] as:

> It is important to establish categories for bugs if you take the goal of bug prevention seriously. If a particular kind of bug recurs or seems to dominate the kinds of bugs you have, then it is possible through education, training, new controls, revised controls, documentation, inspection, and a variety of other methods to reduce the incidence of that kind of bug. If you have no statistics on the frequency of bugs, you cannot have a rational perspective on where and how to allocate your limited bug prevention resources.

The predominant view taken here is that of prevention. By studying the point of origin and the nature of significant flaws in software systems, one can devise techniques to reduce them. Studies that have focused on security vulnerabilities in operating systems, such as the one done by Landwehr et al. [LBMC93], have also taken the view of prevention. The intent is to learn from mistakes so that future systems might avoid repeating mistakes.

No study to date has been reported that classifies flaws based on the difficulty of the *runtime* detection of flaw exploitations. A classification scheme intended to provide feedback to build secure software may be quite different from a classification scheme based on the technique for detecting exploitations of flaws in system software. For example, function parameter validation as a category in the former classification scheme is useful because it corresponds directly to preventive steps that can be taken to avoid flaws resulting from such problems. However, when viewed from the perspective of runtime detection, we are primarily concerned with the manifestation of the exploitation of the flaw in the running system. Two faults that are a result of improper or incomplete parameter checking may result in very different manifestations and thus should be classified in different categories.

Studies that have focused on penetration analysis, for example the RISOS project study [A+76] and the study by Landwehr et al. [LBMC93] have attempted to develop a syntax-directed approach to the detection of security flaws. The aim of these studies is to develop patterns indicative of flaws that can be matched in system source code. We are proposing a dynamic characterization of flaws based on their manifestation in a running system because that is what intrusion detection attempts to do.

3.1.2 Our Classification

Our abstract classification hierarchy has four categories in which a category at a higher level subsumes the category below it in terms of the signatures that can be represented in the category. Precise bounds on matching in each category can be made by instantiating this abstract category. Instantiation requires a precise definition of the structure of a high-level event in terms of low-level audit trail events. For example, a simple instantiation of the hierarchy can be made by defining a high-level event to be the same as an audit trail event. This results in a particular distribution of intrusion signatures in the various categories of the hierarchy. By defining a high-level event structure in more complex ways, this distribution can be shifted to move signatures from higher levels to lower levels. The "best" choice of such a definition depends in part on the nature of the audit trail. A high-level event serves to encapsulate differences in audit trails so that intrusion signatures remain relatively unchanged when written using high-level events. As a useful example, we have instantiated this abstract hierarchy for the Sun BSM [Sun93b] audit trail by defining a high-level event as a DAG of audit trail events. This choice is based on the Sun implementation of recording *read* and *write* system calls in the audit trail as separate events for each flag specified as an argument to these calls. This instantiation is presented in Chapter 4.

In the discussion below, we use the term "thing" for a "high-level event." Our classification scheme, in increasing order of representability of signatures, is:

1. *Existence.* The fact that something existed is sufficient to detect the intrusion attempt. Existence patterns can be thought of as system state predicates that can be evaluated by inspecting the state of the system at a fixed time, rather than predicates on events. Examples include searching for specific permissions on special files, looking for the presence of certain files, or ensuring that file contents follow a specific format, both syntactic and semantic. Existence patterns look for evidence that may have been left behind by an intruder. Existence patterns are needed, for example, when all security-relevant activity is not reflected in the audit trail, or when the integrity of the audit trail is questionable. This can happen when file systems are remotely mounted, or when the audit trail is destroyed by the intruder. Existence patterns may then be devised to scan the file system for the presence of unauthorized setid[1] files, unsafe permissions on devices, etc.

   Although the focus of our classification is to categorize signatures, not vulnerabilities, an extension of existence patterns can also be used to detect vulnerabilities in systems. A study conducted by Bishop on UNIX vulnerabilities [Bis95] revealed that as many as 95% of all vulnerabilities in his study originated from configuration problems. These can be modeled and detected using existence patterns. Checks performed by static analysis tools such as COPS [FS90] and TIGER [SSH93] can also be modeled by existence patterns. Example vulnerabilities:

   - Cert Advisory 93:03 [CER]. The default permissions on a number of files and directories in SunOS 4.1 were being set incorrectly. Because UNIX

[1]Both setuid and setgid.

models devices as files, incorrect permissions may permit kernel memory to be read for passwords or devices read from or written to.

- Cert Advisory 93:15 [CER]. The device `/dev/audio` was world readable so any user with an account on the system could listen to any conversation that was within audible range of the built-in microphone.

An existence signature to detect these vulnerabilities checks to see if the permissions on the relevant files are set incorrectly.

The time required to match patterns of this type is constant per event and is independent of the history of events leading up to the current event.

2. *Sequence.* The fact that several "things" happened in strict sequence is sufficient to specify the intrusion. The time to process an event for sequence patterns depends on the events in the event stream that occurred before the event. If patterns can specify constraints that hold on the data fields associated with events, then matching sequences is computationally at least as difficult as solving NP Complete problems, i.e., it is NP Hard. Simple constraints involving only equality tests between "things" comprising a sequence pattern can represent NP Complete problems. For example, to determine the Hamiltonian path of an arbitrary graph with n vertices, a sequence pattern can be devised that selects n distinct vertices (specified using equality constraints) such that consecutive vertices in the selection are valid edges in the graph.

   Two special cases of this category relevant to intrusion detection are:

   1. *Interval.* "Things" happened an interval x apart within a specified accuracy Δ. This is specified by the condition that an event occur no earlier than $x - \Delta$ and no later than $x + \Delta$ units of time after another event.

2. *Duration.* This requires that "things" existed or happened for not more than nor less than a certain interval of time. Duration of complex events can often be specified as interval constraints between simpler ones.

Both types of requirements can be handled within the framework of sequence patterns by using appropriate context expressions, or constraints. As an example of a sequence pattern, consider the representation of a race condition attack that involves switching a link to a setuid shell script file. This scenario exploits the `#!` mechanism of determining the executable file to run in the `exec()` system call. In some older UNIX kernels `exec` reads the first two bytes of the program file it is trying to execute to determine if it is a shell file. If the first two bytes are `#!`, it reads the next several bytes to determine the name of the interpreter to run and gives the name of the current file (the one containing the `#!`) as an argument to it. The attack works by making a link to a setuid shell file and invoking the program through the link. The link is then quickly pointed to a malicious script. If the race condition succeeds, the malicious script is executed because it is passed as the argument to the interpreter. Furthermore, the malicious script is executed with the same user id as the owner of the original file (the file with the `#!`) because exec uses the permissions on that file to determine the user id of the interpreter that is invoked when the file is setuid. The timing numbers below are purely illustrative. `T.b` is the time when command `b` is done and so on.

```
a. ln setuid_shell_script FOO
b. FOO &
c. rm FOO (200ms<=T.c-T.b<=1s)      #unlink FOO between 200ms
                                    #and 1s of invoking FOO
d. ln any_shell_script FOO (T.d-T.b<=1s) #relink it within 1s
```

A pictorial representation of this pattern might look as shown in Figure 3.1, which is a sequence of three actions: creation of an alias, execution, change of the alias.

CREATE AN ALIAS TO A FILE	EXECUTE A SETID SHELL SCRIPT BY INVOKING IT THROUGH AN ALIAS	CHANGE THE ALIAS TO POINT TO ANOTHER SCRIPT

Context: The aliases in box_1, box_2, and box_3 are the same and the time requirements are satisfied.

Figure 3.1 A Race Condition Attack Represented as a Sequence Pattern

Other, similar examples of sequence patterns include:

- Race condition attacks in which a process running with elevated privileges accesses an object. The process first checks (usually with the `access` system call in UNIX) whether it is permitted access to the object without elevated privileges and, if so, accesses the object. Because these two operations are not atomic when taken together, there is a window of opportunity in which the process's notion of the object can be made to change. This can result in unauthorized accesses to arbitrary objects in the system. An example of this attack is the `lpr` attack[2] discussed on the bugtraq [Bug] mailing list by Jeremy Epstein on 10/21/94:

  For example, if lpr checks whether you have access to a file being queued (using the access() system call), but lpd fails to verify that the file is still what it was before (i.e., if its a symbolic link it hasn't been

[2]A window of opportunity still remains between the time `lpd` makes its check, and accesses the file, but that is a replay of the same race condition at a lower level as the race condition between the access check by `lpr` and printing by `lpd`.

changed, if it was a file when spooled it hasn't become a symbolic link), then you could get printouts of files you have no rights to...

- Cert Advisory 93:18 [CER] which address a vulnerability in `/usr/etc/modload` and `$OPENWINHOME/bin/loadmodule` in some Sun Microsystems, Inc. architectures. In these architectures `loadmodule` runs as setuid root without resetting `IFS`[3]. It calls a program with the absolute name starting with `/bin` and does it using a call to `system()`. Because `system()` uses the shell to parse its arguments, a program called `bin` can be invoked by setting IFS and PATH[4] appropriately. If a program called `bin` is in the caller's path, it is executed as user root.

  A simple signature for this vulnerability is to test every process when it begins execution to ensure that the program the process corresponds to resides in a trusted area if the process is executing with elevated privileges.

3. *RE Patterns.* These are extended regular expressions involving events and permit the direct specification of AND as a primitive to construct patterns. Synchronization between subpatterns can be represented through the AND primitive. Representability of regular expressions also provides the use of non-determinism, repetition, and the use of alternation in pattern specifications.

   Examples of these patterns include intrusion signatures that often specify several activities to be done jointly, but in any order. Such signatures can be written using the AND primitive. For example, attack scenario number four described by Bishop [Bis83] provides a root shell by exploiting `/bin/mail`. `/bin/mail` is the local mail delivery program that worked in earlier versions of UNIX by changing the user id of the mail file to that of the recipient's uid,

[3] IFS, or the internal field separator, is a user assignable variable in some user shells, such as the Korn shell [BK89], that determines how an input line is separated into command words.

[4] PATH is a user assignable variable provided in most user shells that defines the search path to look for commands to be executed.

but failed to clear the setuid bit on the file. One attack script for exploiting this scenario is the following, described by Koral Ilgun [Ilg92]:

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

AND pattern features are required to represent this signature when translated literally. `touch` is not related to `cp` and `chmod`, but must precede `mail`. The pattern might be represented as

```
(touch AND (cp; chmod)); mail
```

where ; indicates sequence. This example is explained further in Section 4.1.

This category is a superset of sequence patterns.

4. *Other Patterns.* This category contains all other intrusion signatures that cannot be represented directly in one of the earlier categories. Examples of these patterns include:

- *Patterns that require embedded negation.* Matching negation patterns involves searching through the entire search space for *absence of match.* For example, it is not possible to directly specify in our hierarchy that a pattern successfully match when a `read` followed by a `write` is not followed by a `close` within five seconds.

  Our interpretation of negation is "not followed by," instead of the greedy match "anything but." Thus, in our interpretation, the regular expression pattern $a\overline{bcd}.*$ specifies a not followed by bcd, followed by anything. This pattern fails to match the input $abcde$ in our interpretation. With a greedy interpretation of matching, the pattern matches the input because in that interpretation, the pattern specifies a followed by anything except bcd, followed by anything. This is easily realized in the input as $a|b|cde$, where b matches $\overline{bcd}$ and cde matches $.*$.

Matching negation patterns requires an exhaustive search because NP-Complete problems and their negation counterparts can be represented as signatures. For example, the problem: does an arbitrary graph *not* have a Hamiltonian cycle? If such signatures required less than an exhaustive search for matching, we could derive clues to the problem NP = co-NP. That, in turn, would provide clues to P = NP. Both are unsolved open problems.

- *Patterns that involve generalized selection.* For example, to specify a successful match if any $x-3$ out of x conditions are satisfied, all possible ways of selecting $x-3$ conditions out of x have to be represented in the pattern.

The relationship between the categories is shown graphically in Figure 3.2. The categories from top to the bottom represent increasing representibility of intrusion signatures. That is

$$\text{Existence} \subseteq \text{Sequence} \subseteq \text{RE patterns} \subseteq \text{Other patterns}$$

Interval and Duration are subsets of the category Sequence.

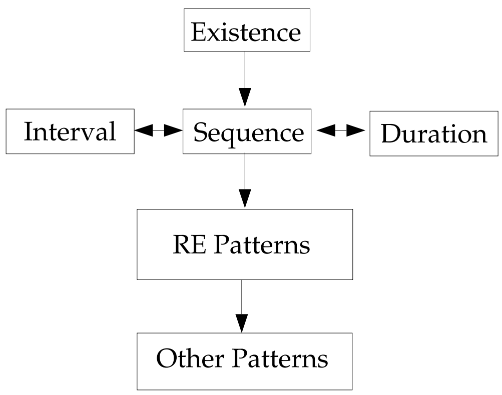

Figure 3.2 The Abstract Signature Classification Hierarchy

3.1.3 Relevance of this Classification

This classification yields a categorization of intrusion signatures that is independent of any underlying computational framework of matching. The classification also serves as the basis for instantiating any such computational framework. We have populated this hierarchy with intrusion signatures [KS] and the majority of the intrusions we studied were contained in the first three categories.

Using this classification as the basis for a computational framework can be approached in two ways. Each category in the classification can be treated independently and a computational procedure devised that matches signatures in that category. This yields disparate solutions to the matching problem in each category. Alternatively, a unified procedure can be devised in which all categories can be represented and matched in one model. The approach taken in this dissertation is that of a unified model of matching, which is presented in Chapter 4.

While our focus in deriving this classification has been to study exploitations of vulnerabilities in the UNIX operating system, we contend that the hierarchy is also valid for other operating systems. This point was presented in Section 1.4. If operating systems have similar methods of exploitation, then the manifestations of these exploitations in the audit trail are also similar. For example, race condition attacks, which are present in many operating systems, are often of the same generic type and may be modeled for detection as *sequence* patterns.

3.2 Intrusion Detection as Pattern Matching

In this section we show how pattern matching can be used to examine or monitor for signatures in the audit trail. Our approach encodes signatures as a formal, structured representation of low-level system events that constitute the exploitation of the attack. We show how this approach can be used with any underlying abstracted event stream. We discuss the benefits of using pattern matching for detecting intrusions. We also discuss generic requirements that any intrusion detector using a pattern matching approach must satisfy when run in the current paradigm of audit trail generation.

3.2.1 Intrusion Signatures as Patterns to be Matched

To show the likeness of intrusion signatures and patterns in the sense of classical pattern matching, consider the monitoring of Clarke-Wilson [CW89] integrity triples in a computer system using the system generated audit trail. Clarke-Wilson triples are devised to ensure the integrity of important data and specify that only authorized programs running as specific user ids are permitted to write to files whose integrity must be preserved. This is similar to the maintenance of the integrity of the password file on UNIX systems by allowing only some programs, like `chfn`[5] to alter it. One pattern that might be used for this purpose associates and matches a sub-signature for creating a process with another that writes to files. By appropriately specifying that the created process is the same as the one that writes, and retrieving the user id, the program name and the file name from the context of the match, one can monitor Clarke-Wilson integrity triples by pattern matching. See Figure 3.3 for a pictorial representation of the signature.

The approach of viewing intrusion signatures as patterns to be detected by matching them against the audit trail has the following benefits:

Event Layout Independence. The pattern specifications do not include a description of the layout of events. Instead, they import an event interface. A pattern only needs to use what data an event can provide, regardless of how the event provides it. This insulates the pattern specification from layout variations in the event stream. Because pattern specifications are declarative, a standardized representation of patterns enables them to be exchanged between users running variants of the same operating system, with varying audit trail formats. For each such system the translation mechanism of converting the pattern to its underlying matching automaton, and the encapsulation of the audit data within

[5] `chfn` is used to change information about users which is stored in a well-known file, `/etc/passwd`.

the event interface is ported. Once this is done, signatures can be reused among systems.

Declarative Specification. Patterns representing intrusion signatures can be specified by defining what needs to be matched, not how it is matched. That is, the pattern is not encoded by the signature writer as code that explicity performs the matching. This permits sequencing and partial order constraints on events to be represented in a direct declarative manner. The benefit is the clean separation of the matching from the specification of what needs to be matched.

Dynamic Pattern Creation. Patterns representing attacks can be dynamically created at the time of need. This facilitates complex matching requirements to be specified as a hierarchy of pattern matches. A pattern matched at a lower level in this conceptual hierarchy can create and trigger the matching of a pattern at a higher level thus permitting a layered structure of representing complex matches. Furthermore, patterns tailored on conditions only known at runtime can be created.

Event Source Independence. As the pattern matching process only makes use of the event interface visible to it, events that correspond directly to underlying system events or those that are artificially generated can be used the same way. Synthetic events can be generated and used by augmenting the event interface with a description of these events. The interpretation of these synthetic events can be made completely application dependent, being done by the particular patterns that make use of them.

Multiple Event Streams. Multiple event streams can be used together to match against patterns for each stream without the need for combining the two into one stream. For example, IP datagrams and C2 audit events can be handled together to corroborate evidence of intrusion. As no assumptions are made about the nature of these event streams, this mechanism can be naturally used to process multiple sources of the same event type, for example in distributed intrusion detection.

Portability. Intrusion signatures can be moved across sites without rewriting them to accommodate fine differences in each vendor's implementation of the audit trail. Because pattern specifications are declarative, a standardized representation of patterns enables them to be exchanged between users running variants of the same flavor of operating system, with varying audit trail formats.

3.2.2 The Nature of Intrusion Signatures

In this section we outline the general, abstract requirements that pattern specifications must incorporate to represent the full range and generality of intrusion scenarios. These requirements were derived from a study of computer security vulnerabilities described in Bishop [Bis83], CERT advisories [CER], and the COPS security tool [FS90]. The examples are illustrated using UNIX vulnerabilities and a C2 audit trail. This is only because of our familiarity with them and should not be construed as a limitation of this approach.

Context Representation. The patterns must be able to represent the context essential to accurately specify an intrusion. The more accurately one can specify an intrusion, the more one can limit false positives and unwanted matches. The context includes the pre-condition(s) that may need to be satisfied before matching the event group specified by the pattern. The pre-condition verifies that the system is in a state from which the set of actions carried out as specified in the pattern result in an intrusion. Some signatures may not require a pre-condition.

The other type of context involves expressions on the values of event fields. These values may be taken from more than one event. For example, when encoding Clarke-Wilson triples as patterns, one needs to remember the user id and the program name associated with every process spawned to match it against every write (or open for write) to ensure that only those writes to files that are permitted by certain programs executing on behalf of certain users occur. Figure 3.3 shows this pictorially.

A PROGRAM STARTS UP	A PROCESS WRITES TO A FILE
PR = this program's name PID = this process's pid	F = this file's name PID' = this process's pid UID = this process's uid

Context: $PID = PID' \wedge$ Clarke-Wilson access triples do not permit PR running as user id UID to write to file F.

Figure 3.3 Monitoring Clarke-Wilson Triples as a Pattern Match

As another example of the use of signature contexts, consider the representation of the signature: *Raise the audit level of any user for whom there are three or more failed login attempts within two minutes.* The block diagram of the signature might look as shown in Figure 3.4. The sequence of failed logins as depicted implies that $t_3 \geq t_2 \geq t_1$.

FAILED LOGIN	FAILED LOGIN	FAILED LOGIN
user = U time = t_1	user = U' time = t_2	user = U'' time = t_3

Context: if$(U == U' == U'' \wedge t_3 - t_1 < 2m)$raise_audit_level$(U)$;

Figure 3.4 Three Failed Login Attempts as a Signature

Matching in the presence of context is more difficult than matching where only the order of occurrence of events is specified, as in regular expression matching [AHU74]. If the evaluation of the context is linear in its size (of representation) then matching is NP-Complete [KS94]. This means that in general there are no known deterministic algorithms that perform significantly better than trying all possible ways of matching the pattern.

Follows Semantics. The patterns must intrinsically specify the following special case of discrete approximate pattern matching: if the event sequence e_1,e_2,$\ldots$, e_n matches the pattern, then so does e_1,$[x_{11}, x_{12}, \ldots, x_{1l}]$,$e_2$,$[x_{21}, x_{22}, \ldots, x_{2m}]$,$\ldots$,$e_n$, where x_{ij} are arbitrary events. That is, the insertion of an arbitrary number and type of events between any successive events of a matching event sequence continues to render the pattern matched. We refer to this specialization as matching with the *follows* semantics. If this problem is framed in terms of the *edit distance* of converting the input to the pattern, with deletion costs $= 0$, insertion costs $=$ mismatch costs $= 1$, it is to determine if the minimum cost of converting the input to the pattern is 0. This requirement is justified when one considers how event streams (e.g., audit trails) are generated in modern computer systems. Multiple sources of events, for example from several processes, overlap in the final event trail. Because event trail managers (a process) usually collect and write events in the order in which they are received, a single logical thread of events, for example one associated with a process, is interspersed with events belonging to other active entities in the system.

Discrete approximate matching has been extensively studied by Wagner and Fischer [WF74], Myers and Miller [MM89], Yates and Gonnet [BYG89], Manber and Wu [WM91] and Knight [Kni93]. Matching with the *follows* semantics without context representation has the same complexity as matching regular expressions. This has a linear time solution in the deterministic case (ignoring preprocessing) and polynomial time solution when simulating the non-deterministic pattern. These results can be found in the book by Aho et al. [ASU86].

Specification of Actions. The patterns should be able to specify the execution of arbitrary code fragments, both within the pattern's context and when the pattern is matched. For example, it might be desirable to increase the amount of data audited for a user when a suspicious pattern is matched. Some generic mechanism for specifying such actions must be provided without needing to enumerate a priori all the special functions that might be needed for this purpose. Fixing the set of functions that can be used within a pattern when the model of matching is designed is difficult. It is also too restrictive to the pattern writer to work with a fixed set of functions in writing patterns. A general mechanism, for example one based on a virtual machine model that allows complex, user specifiable functions to be constructed from a small, simpler set of instructions, is more desirable. Such a mechanism might also be used to query the system for state information, changing the event trail manager, or to effect state changes in the system itself.

Consider the Clarke-Wilson example of Figure 3.3. The mechanism of storing the CW-triples and checking them against the pattern context can be specified using a function that takes the program name, the user id, and the file name as arguments and determines if it is an allowed triple. The function declaration for such a function might look like:

```
//return 1 if triple not permitted, 0 otherwise
int disallowed(String prog, int uid, String file)
```

Representation of Invariants. The following special types of patterns must be easy to represent: $p_1 \wedge \overline{p_2} \wedge \overline{p_3} \wedge \ldots$ In other words, the pattern is considered matched if a sequence of events $e_1, \ldots, e_n$ satisfies p_1 but does *not* satisfy $p_2, p_3, \ldots$ This often allows operational details of matching (like garbage collection of partially matched signatures that will never completely match) to be specified by the pattern writer without needing to build such mechanisms into the matching solution. This provides flexibility and control to the pattern writer in cases when built-in behavior is inefficient or simply does not provide the mechanism to express the special cases for a particular pattern in which partial matches can never be fully matched.

For example, in the example of Clarke-Wilson triples, one would like to specify that if a process fails to write to a file (or open a file), then the partial match that matches the spawned process but awaits the write should be destroyed because the match will never complete once the process has exited. Rather than bind such detailed behavior into the matching solution we find it conceptually simpler for the pattern writer to encode it as part of the pattern itself. The pattern specification would then look like: *match the spawning of a process and its subsequent writing to a file so long as the process has not exited.*

We are not particularly concerned here with the theoretical completeness of this approach of specifying all conceivable situations in which partial matches can be fruitfully deleted. Based on an empirical study of intrusion signatures we view this mechanism to be sufficient. Our inclination between efficiency and generality is towards efficiency. We have attempted to devise constraints for a model that can represent and efficiently match a large proportion of the common cases (which is inextricably tied to empiricism), and not to devise a general-purpose solution in which every possible condition can be represented and matched.

3.2.3 System and Other Considerations

In addition to the model requirements presented in the previous section, we have found it useful to place additional constraints on a particular instantiation of the model [KS95]. These can be viewed as system constraints on the final packaged misuse detector. These constraints attempt to answer the question: if we could devise a model of matching that met the requirements of Section 3.2.2, how would we structure a system around it? What would be desirable characteristics of the model and the system? We believe the following to be some of those desirable characteristics.

Dynamic addition and removal of patterns. This is the ability to add and remove patterns to be matched as the matching proceeds. This ability serves several useful purposes. For example, it enables the short-lived instantiation of specially tailored patterns to confirm or deny evidence in model-based intrusion detection [Section 2.3.5]. Or, it might allow coarse patterns to generate successively more refined patterns to confirm or deny intrusive activity once they are matched.

This ability also provides more control to a security officer in charge of securing the system. He can weed out unnecessary or unuseful patterns and add new ones to the system without bringing down the system and re-starting it. There are added benefits of easier testability and the significant capability of embedding mechanisms for the automated generation of newer, better signatures by one of several techniques including genetic algorithms [Koz92].

Incremental Matching. By this we mean that the events in the event stream are made available one at a time and the matcher must indicate all successful matches after each given event. In other words, all the events are not available *a priori* for preprocessing. This requirement often makes the matching solution computationally difficult. In addition to incremental matching we have found it useful for the matching to be online, i.e. matching is done concurrently with the generation of events. This is because signature context often requires the availability of system state information which is usually meaningless in an off-line solution.

Prioritization of Patterns. In the case of several patterns it must be possible to give matching preference to some patterns over others. This requirement is one of proper distribution of limited computing resources. If the matching of events against patterns proceeds at a rate faster than the event generation rate, prioritization may not be necessary. But, in a setting in which the monitored machine and the event processing machine are the same, it might be desirable to temporarily disable some patterns so that matching proceeds more rapidly for the remaining patterns.

All Matches. The matching solution must provide for all matches of all patterns in the system. From a security perspective, it is often more desirable to know all the specific violations rather than knowing that a violation has occurred, or knowing the first violation as soon as it occurs.

From the enumeration above we have ignored performance requirements such as efficiency or real-time behavior, low resource overhead, and scalability of the solution with respect to the number of patterns to be matched simultaneously. These are important requirements but good values of these measures cannot be quantified independent of the specifics of the environment in which the detector will run.

3.2.4 Further Advantages of a Pattern Matching Approach

There are several added benefits of viewing misuse detection as a pattern matching solution. By considering intrusion signatures as patterns, the audit trail as an abstracted event stream, and the detector as a pattern matcher, we can cleanly separate the major components of a generic misuse detector. This enables different solutions to be substituted for each component without changing the overall engineering structure of the system considerably. The event stream encapsulates the syntactic and data representation differences present in various audit trails. Semantic differences may be more difficult to subsume without changing the signature. Because matching is done on the information contained in the events, of which the matcher has no information, any abstracted event stream will do, for example network packets. This makes the system more portable. Furthermore, if the pattern representation is standardized, patterns can be distributed to other sites which may run a different version of a similar operating system and a different version of the audit trail. Each site need only write a structural description of the audit trail once for all its patterns.

A simplified misuse detector can then be an application program that uses a mechanism to dispatch incoming events to patterns and uses calls to a pattern matching library to do the matching of those patterns. This means that building misuse detectors no longer requires learning specialized tools, techniques, and theories before using them as building blocks for a misuse detector. It can be as simple as understanding and using a matching library.

Pattern matching has been extensively studied as a discipline. It is amenable to several optimizations that can make a system built around it practical and efficient. For example, the evaluation of context is amenable to compiler optimization techniques. It might also be possible to combine several patterns together into a joint pattern with better matching characteristics. The use of pattern invariants allows the pattern writer to encode patterns that do not need to rely on primitives built into the matching procedure to manage the matching. One example is the ability to clean up partial matches once it is determined that they will never match. This frees the matching subsystem from having to provide a complete set of such primitives and, in the process, coupling the semantics of pattern matching with the semantics of the primitives.

Conceptually, patterns representing vulnerabilities in our model subsume static methods of intrusion detection such as those incorporated in tools such as COPS [FS90] and TIGER [SSH93]. By specifying that a pattern does not match events but instead satisfies a context when created, all the checks that these tools make can be verified. Thus, we can encode tests that not only verify that the system is initially clean but also continues to remain so as the system continues to function.

This approach is, then, limited only by the expressive power of the patterns and the computational intractability of matching imposed by their generality. Within the framework of the outlined model, patterns can be designed to perform tasks beyond the traditionally defined domain of misuse detection. We believe that with a well designed model for representing patterns, simple anomalies can also be represented and detected in this framework.

3.2.5 Disadvantages of a Pattern Matching Approach

Given a well constructed pattern that represents an intrusion scenario it might not be too difficult to match it against the event stream. A difficult problem, however, is the identification and extraction of the core crucial elements from exploitation descriptions, such as those described in the bugtraq [Bug] and 8lgm [8lg] mailing lists, and turning them into general descriptions for detecting variations and permutations of the vulnerabilities. Currently it requires human expertise to do the translation and there is no easy way to automate the process. Abstracting high quality patterns from attack scenarios is much like extracting virus signatures from infected files. The patterns should not conflict with each other, be general enough to capture variations of the same basic attack yet accurately represent the intrusion to reduce false positives and false negatives, and be simple enough to keep the matching computationally tractable.

While this technique only works for vulnerabilities that are known and for which patterns have been devised, it is the case that newer vulnerabilities are often different ways of exploiting well-known problems in system software. This approach examines the trace of a running system for 'behaviors' in an attempt to monitor suspicious behavior. A well written signature can reduce the effects of aliasing so that it is possible to represent the crux of an intrusion that is unchanged by minor rearrangements of the exploitation scenario and is insensitive to the path taken to effect the intrusion.

Signature analysis assumes the integrity of event data. Thus, attacks that involve spoofing, which produce the same events (but from an untrusted source) cannot be reliably detected. Furthermore, passive methods of security breaches such as wiretapping cannot be detected at the time of the breach because they do not produce a detectable signature.

3.3 Summary

In this chapter we presented a scheme to represent intrusion patterns based on the complexity of matching. Because representation of context is fundamental to the representation of intrusion signatures, our classification assumes it at each level of categorization. Most of the intrusions we studied can be represented in the first three categories of our classification. These categories serve to group signatures, not vulnerabilities. Different encodings of the same security vulnerability can be made based on the desired accuracy of detection, resulting in a corresponding tradeoff in the complexity of detection. We believe this categorization is also applicable to other operating systems.

We also outlined requirements that must be satisfied by a pattern matching solution if the monitoring of intrusion signatures is to be done using pattern matching. Different intrusion detection systems may make different tradeoffs among these requirements but all systems will have to address all the requirements to some degree. These requirements were empirically derived from a study of commonly occurring intrusions described by Bishop [Bis83], made public via advisories such as those put out by CERT [CER], and embedded in tools like COPS [FS90]. We also outlined some system considerations that might be useful when implementing these requirements in a practical system.

The pattern matching approach should be viewed as a technique specifically tailored for intrusion detection. Thus, the pattern requirements are not intended to provide a general-purpose audit trail analysis because we are not primarily concerned with the specification and matching of every conceivable interrelationship among events. Instead, we want to provide a mechanism that is simple and efficient, and permits the specification of a large percentage of intrusions.

For effective misuse detection, a pattern matching approach sometimes requires the use of facilities that are not currently provided by protection mechanisms and audit trails available on computer systems. Our technique assumes the availability of these facilities. For example, we implicitly assume that for proper detection of intrusions, complex programs with a history of bugs generate a high-level audit trail that can be used for this purpose.

4. A MODEL INSTANTIATION

In Section 3.1 we presented a classification hierarchy to categorize intrusion signatures based on the structural interrelationship among events used to represent the signatures. In Section 3.2 we presented the requirements that patterns in all categories of the classification must meet to represent the full range of commonly occurring intrusions. These requirements included the specification of context, actions, and invariants in intrusion patterns. In this chapter we present a model of matching that we have devised for misuse intrusion detection that is a synthesis of the classification hierarchy and the generic pattern requirements. Signatures devised using this model can use context, actions, and invariants and span all classes of the hierarchy.

The model is based on Colored Petri Nets, described by Jensen [Jen92]. Each intrusion signature is represented as a Colored Petri net. The notion of one or more start states and exactly one final state in the net are used to define matching in the model. Context is saved as the colors of a token. Matching by definition specifies the "follows" semantics. Conditions are specified using guard expressions and actions are represented using state actions. The theoretical properties of this model are studied in Chapter 5. We have built a prototype of the model in C++ and tested it with intrusion signatures derived from real vulnerability data. The structure of the prototype and the simulation results are presented in Chapter 6.

4.1 The Model

In this section we introduce the model informally with an example. In Section 4.3 we define the model more rigorously. We have translated the example exploitation used in this section into a pattern literally, but that is only to highlight the various features of the model using the fewest examples. Examples of some intrusion signatures that we used to test the prototype are given in the appendix. Consider the representation of the following attack scenario that was briefly explained in Section 3.1.2.

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

This attack exploits the vulnerability in early versions of `/bin/mail` and the weakness in the structure of the mail delivery subsystem. `/bin/mail` is a local mail delivery program that delivers mail to local mailboxes. It worked in early versions of UNIX by changing the user id (owner) of the recipient's mailbox file to the recipient's user id, but failed to clear any other permission bits on the file. An attacker could exploit this behavior by waiting for user root's mailbox to be empty and then copying `/bin/sh` to root's mailbox file. This was possible because the system wide mail directory `/usr/spool/mail` was writable by everyone. Because the newly created mailbox file was owned by the attacker, he could set its setuid bit. In the final step of the exploitation, he would simply mail an empty message to root that `/bin/mail` would append to the mailbox file and change the file ownership to root. Because `/bin/mail` does not check or reset any other bits on the mailbox file the attacker then had a setuid root shell.

A literal representation of this attack is represented graphically in Figure 4.1. The horizontal chain of circles (states) and vertical bars (transitions) encode the activity

```
cp /bin/sh /usr/spool/mail/root
chmod 4755 /usr/spool/mail/root
```

while the diagonal chain encodes the activity

```
touch x
```

The transition labeled `t7` represents a synchronization point at which both chains must have matched for the pattern to be matched further. In describing the pattern graph we refer to it as a Colored Petri Automaton, or CPA. When referring to circles in a CPA, we use the notation of "state" over "place" because that is closer to the more familiar finite state automata terminology.

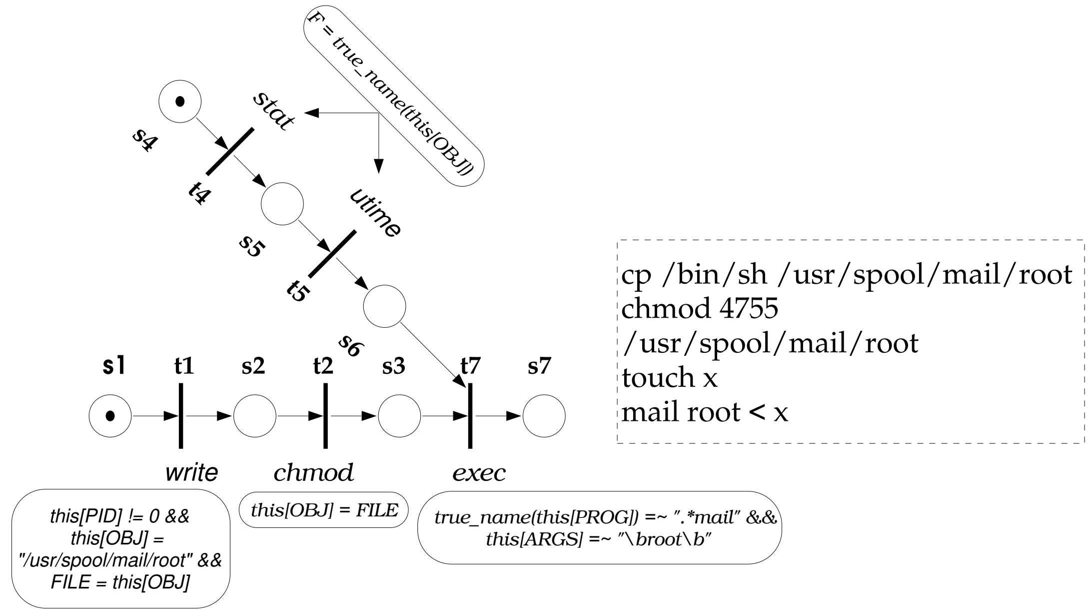

Figure 4.1 Representing Synchronization of Events

The circles in Figure 4.1 represent system states and the thick bars the transitions. `s1` and `s4` are the initial states of the CPA, and `s7` is its final state. A CPA requires the specification of ≥ 1 initial states and exactly one final state. A start state must have no arcs incident on it, and a final state must have no arcs emanating from it. At the start of a match, a *token* is placed in each initial state.

A CPA may have a set of variables associated with it. Assignment to these variables is equivalent to their unification. This means that variables can be assigned a *unique* value only once. Attempting to assign different values to a variable causes the assignment to fail. It also means that variable assignment need not only be specified as `var = val`, but may also be specified as `val = var`. Assignment and testing for equality between variables or between variables and values is the same operation. This particular semantics of CPA variables is useful in optimizing the evaluation of *guard* expressions [Section 5.3]. It also permits precomputing the values of variables at certain nodes of the CPA because of their assignment at earlier occurring nodes of the CPA.

Each token maintains its own local copy of these variables because each token can make its own variable "bindings" as it flows from a start state to the final state. In CP-Net terminology, each token is colored, and its color can be thought of as an n-tuple of strings, where the pattern has n variables. In Figure 4.1, variables FILE and F are CPA variables. We use CPA variables synonymously with token local variables because the model uses token colors to represent CPA variables.

A CPA also contains a set of directed arcs that connect states to transitions and vice-versa. Each transition is associated with an event type, called its label, which must occur in the input event stream before the transition will *fire*. In Figure 4.1, transition `t1` is labeled with the event *write*, `t4` is labeled with the event *stat* and so on. The labels correspond directly to the event types against which the CPA is matched. The model provides for a special label `CLK` that corresponds to timer events. Timer events are useful in specifying time bounds for matching or for specifying periodically occurring activity. Transitions may also be labeled with ϵ to indicate that tokens may flow across the transition without being triggered by an event. An ϵ transition cannot change the variable binding of tokens that cross it. Nondeterminism can be specified by labeling more than one outgoing transition of a state with the same label, or with ϵ events. An event can fire multiple transitions labeled with that event. This permits matching patterns with the `AND` semantics. Precluding this concurrent behavior would specify pattern matching with partial order semantics. This is discussed further in Section 4.2.3. A transition is said to be *enabled* if each of its input states contain at least one token.

Optional expressions or guards can be placed at transitions. These expressions permit assignment to the token local variables that flow past the transition. Examples of these expressions include assignment of event data fields to token local variables; evaluation of conditions involving $=$, $<$, or $>$; and calling built-in and user defined functions. Guards are boolean expressions that evaluate to *true* or *false*. `this` is a special operator that is instantiated to the most recent event. Event data can be accessed through the `this` operator. Expressions involving `this` use the array indexing operator [] to refer to data from the current event. In Figure 4.1, transition `t2`, which is labeled by the event `chmod`, accesses the `OBJ` field of the `CHMOD` event, which returns the pathname of the object being chmoded. Guards are evaluated in the context of the event which matches the transition label and the set of *consistent* tokens that enable the transition. Tokens are consistent when their variable bindings unify. The set of tokens are unified before being used in the guard expression for evaluation.

For example, for transition `t7` to fire, there must be at least one token in each of states `s3` and `s6`; the enabling pair of tokens (one from `s3`, the other from `s6`) must have consistently bound (unifiable) variables; and the unified token and the event of type `exec` together must evaluate the guard at `t7` to *true*. A transition *fires* if it is enabled and an event of the same type as its label occurs that satisfies the guard at the transition. When a transition fires, the set of consistent tokens are *unified* to one token, and copies of this unified token are placed in each *output state* of the transition. A state `s` is an output state of transition `t` if there is a directed arc from `t` to `s`. For example, in Figure 4.1, the only output state of `t2` is `s3`.

The process of unification resolves conflicts in bindings (i.e., ensures that token bindings, if present, are identical) between tokens to be unified and stores a complete description of the path that each token traversed in getting to the transition. Thus, a token not only represents binding, but also the *composite* path that it encountered on its path to the current state.

The event sequence matched by a CPA is the sequence of events (or other ordering) encountered at each transition by the token that has reached the final state.

The states of a CPA can also be associated with *actions*. These actions are performed for each token that reaches the state. Actions allow the specification of activity to be made before the entire pattern is matched. This allows the encoding of countermeasures when a partial signature is recognized. Actions can also be used to invoke built-in primitives, for example the recursive invocation of the same pattern, or resetting the pattern.

Invariants, or conjuncted negative specifications, are specified using their own graphs. These graphs are similar to pattern graphs and are matched in the same way. For each token that reaches the successor state of the CPA start state, its copy is placed in the start state of the invariant graph. A match of the invariant graph nullifies the pattern match. This is elaborated further in Section 4.2.1.

The model has a built-in operation `RESTART`. When executed, it removes all tokens from the CPA and its invariants and reinitializes the CPA by placing new tokens in its start states. This operation is useful when a pattern match collects data for a period of time, at the end of which matching is reinitiated.

4.2 An Example Simulation

As another example, consider the pattern shown in Figure 4.2.

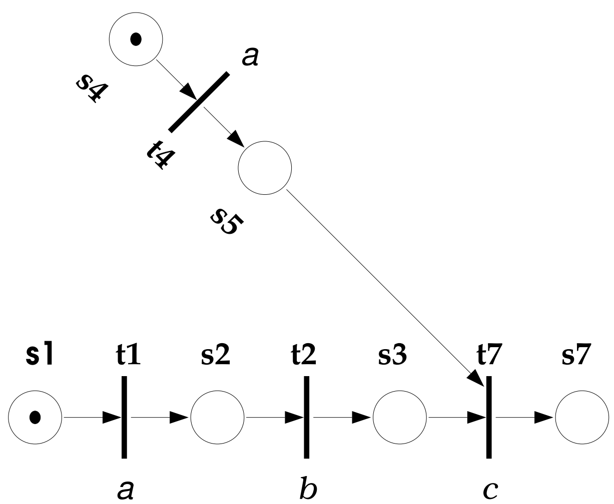

Figure 4.2 Simulation of a Pattern That Does Not Use Guard Expressions or Token Local Variables

Assume that we want to match the pattern against the event sequence *abac*. The steps in the non-deterministic match of the CPA against the event sequence are described in Table 4.1. Non-deterministic matching is used in the sense of computing using an oracle, similar to that used in matching non-deterministic finite automata as described by Aho, Hopcroft and Ullman [AHU74].

Step	Input	Token Configuration	Comment
1.	$.abac$	$\{s1, s4\}$	
2.	$a.bac$	$\{s2, s4\}$	The CPA non-deterministically chooses to move token $s1$.
3.	$ab.ac$	$\{s3, s4\}$	
4.	$aba.c$	$\{s3, s5\}$	
5.	$abac.$	$\{s7\}$	The two tokens are merged to one. From the token in *s7* we can reconstruct the path of each individual token from the initial marking.

Table 4.1 Non-deterministic Matching of a CPA

Merging of tokens $s3$ and $s5$ occurs in step four and this requires conflict resolution. Because tokens associated with this pattern do not have variables, tokens unify trivially. This example illustrates how the CPA non-deterministically matches the pattern. In non-deterministic matching, tokens are *removed* from one or more states and placed in others, and the matching procedure always makes the right choice in the selection of tokens and transitions to exercise.

In a deterministic, exhaustive search on the other hand, tokens are never *moved* from one state to another, they are instead duplicated and copies moved to other states. Because tokens are colored, i.e., they have data bindings associated with them, each token is in a sense unique. Therefore, tokens residing in the same state cannot be merged. Furthermore, it is not permissible to lose the binding of a token by *moving* it across a transition, instead the previous binding must be preserved for a later match, and a duplicate created and placed in the output state of the transition. A deterministic simulation of the same pattern is shown in Table deterministic-matching-of-cpa. The superscripts associated with states denote the number of tokens in the state. The simulation procedure is described in Section 6.4.2.

Step	Input Consumed	Token Configuration
1.	$.abac$	$\{s1, s4\}$
2.	$a.bac$	$\{s1, s2, s4, s5\}$
3.	$ab.ac$	$\{s1, s2, s3, s4, s5\}$
4.	$aba.c$	$\{s1, s2^2, s3, s4, s5^2\}$
5.	$abac.$	$\{s1, s2^2, s3, s4, s5^2, s7\}$

Table 4.2 Deterministic Matching of a CPA

4.2.1 The Semantics of Invariants

An invariant graph is similar to an ordinary pattern graph. If P denotes the pattern and I its invariant, then a negative invariant graph represents the condition $P \wedge \overline{I}$. That is, if P is matched by a sequence of events $e_1, \ldots, e_n$ (with the follows semantics), then $e_1, \ldots, e_n$ does *not* match the invariant graph. Negative invariants are usually used to specify when it is no longer useful to continue searching for a match. This way tokens can be destroyed to prevent build up of unnecessary tokens in a pattern graph. The rule for token destruction is:

> When any token in I reaches the final state, it is destroyed along with the tokens in the pattern and the invariant that have the same *root token*.

Two tokens have the same root token if both are the result of *duplicating* tokens that themselves have the same root token. A token is its own root token. See the appendix for implementation details on how this is done.

4.2.2 CPA Variable Semantics

As mentioned earlier, token local variables cannot change values once initialized. Global variables that need to be shared across CPAs must be provided by the external environment. These variables can be manipulated by either guard expressions or pattern actions. Concurrency control of accessing shared variables must be handled external to the model.

4.2.3 Partial Order or AND Matching Semantics

In choosing partial order semantics for matching a CPA against an event sequence, any event is restricted to exercise at most one transition. In choosing to match with the AND semantics, all transitions labeled with that event must be exercised.

When a CPA has at most one transition labeled with any given event, matching with either semantics yields the same matches. In our experience, intrusion patterns can either be naturally represented as AND patterns, or can be judiciously encoded in this manner. The simulation procedure for matching with the AND semantics is a straightforward extension of the simulation procedure for non-deterministic finite state machines. Thus, our choice of matching semantics for the CPA is that of AND semantics.

The basic CPA model of matching can be extended to match with partial order semantics if that is needed for intrusion signatures.

4.3 Formal Definition of a CPA

This section presents a formal description of Colored Petri automata that was informally introduced in the previous section. We define its operational semantics in terms of updates to its internal state as it performs the match against the event stream.

Let Σ be the finite set of event types over which matching is performed. The event stream consists of a sequence of zero or more events. Each event type $\sigma \in \Sigma$ has a fixed set of attributes associated with it [cf. Section 1.3]. Let these be labeled $a_1^\sigma \ldots a_l^\sigma$ where l depends on σ. The set of attributes may be different for each $\sigma \in \Sigma$. Each instance of an event of type σ in the event stream may have different values for its attributes. By Σ^* we denote a sequence of zero or more events. If e denotes an event, then by Label(e) we denote the type of e.

A Colored Petri automaton M that matches over an event sequence in Σ^* is the 11-tuple $(S, T, V, B, E, G, I, O, S_i, F, N)$ where:

- S is the set of states of M.
- T is the set of transitions of M. The set of states and transitions are disjoint, i.e., $S \cap T = \phi$.
- V is the set of CPA variables, also referred to as token local variables. These variables define the color of tokens associated with M.
- B is the global state that can be manipulated by M. The global state is a set of (variable, value) pair bindings.
- E is the labeling function $E : T \to \Sigma \cup \{\epsilon\} \cup \{\text{CLK}\}$ that labels each transition with an event type.
- G is the labeling function $G : T \to X_{\sigma,V}$ that maps each transition to a guard expression. $X_{\sigma,V}$ is the set of valid boolean expressions that only use the data attributes associated with the event σ, the CPA variables of M, and the global state B.
- I is the set of directed edges that connect states to transitions, i.e., $I : S \to 2^T$. 2^T denotes the power set of T, i.e., the set of all subsets of T.

- O is the set of directed edges that connect transitions to states, i.e., $O : T \rightarrow 2^S$. 2^S denotes the power set of S, i.e., the set of all subsets of S.
- $S_i \in S$ is the set of start states of M. No start state has an incoming edge, i.e., $\forall s \in S_i, I(s) = \phi$.
- $F \in S$ is the final state of M. The final state has no outgoing arc, i.e., $O(F) = \phi$.
- N is the invariant associated with M. N is a CPA graph similar to M except that (1) N does not have an invariant associated with it (2) there is exactly one start state in N and (3) all transitions in N have exactly one input state i.e., synchronization (direct specification of AND) is not permitted in invariants. N is represented by the 10-tuple (SN, TN, V, B, EN, GN, IN, ON, SN_s, FN). N shares the same variable space V and the global state B as M.

A token k associated with M (and N) is defined as the bindings over the set of CPA variables V of M. For each variable $v \in V$ associated with a token, v either has a value, or is *uninstantiated*, denoted here by the value ϕ. The vth variable binding of token k is denoted as k_v. Two or more tokens $k_1, \ldots, k_n$ are said to *unify* iff

$$\forall v \in V,\ \left(\forall_{p=1}^{n}\ \left(\forall_{q=1}^{n}\ \left(\left(k_p\right)_v = \left(k_q\right)_v\right)\right)\right)$$

A token variable value ϕ is equal to any value. The bindings of the unified token, denoted here as Unify$(k_1, \ldots, k_n)$ that is the result of unifying the tokens $k_1, \ldots, k_n$ is

$$\text{Unify}(k_1, \ldots, k_n)_v = \begin{cases} l_v & \text{if } \exists \text{ a token } l \,|\, l_v \neq \phi \\ \phi & \text{otherwise} \end{cases} \quad , v \in V$$

We denote by TE the bag (multiset) of all possible token values associated with M. That is, each element of TE is a token with an arbitrary binding for its variable set V.

A marking μ is the function $\mu : S \cup SN \rightarrow TE$ that assigns tokens to the states of the CPA M and its invariant N. The initial marking of M consists of exactly one token with no bindings, i.e., $\forall v \in V,\ k_v = \phi$, in each of its start states $s \in S_i$ and no tokens in any other state.

By the function $\text{In}(t)$, $t \in T \cup TN$, we denote the set of states $s \in S \cup SN \mid \exists$ a directed arc from s to t. Similarly, we use the function $\text{Out}(t)$, $t \in T \cup TN$ to denote the set of states $s \in S \cup SN \mid \exists$ a directed arc from t to s. The function $\text{NonZero}(s)$ is true iff the state $s \in S \cup SN$ has no tokens resident in it. The function $\text{Zero}(s)$ is the complement of the function NonZero. The function $\text{State}(k)$ returns the state $s \in S \cup SN$ in which the token k is resident.

The transition function $\delta : (\mu, e) \to \mu'$ takes a marking and an input event, and returns a new marking that represents the new state of M after it has been exercised with e. δ may *fire* a transition $t \in T \cup TN$ if:

- $E(t) = \text{Label}(e)$, and

- t is *enabled*, i.e., there is at least one token in every input state of t, i.e., $\forall s \in \text{In}(t)$, $\text{NonZero}(s)$, and

- If $x = |\text{In}(t)|$, then $\exists$ a set of tokens $k_1 \dots k_x \mid$

  $$(\text{State}(k_1) \neq \cdots \neq \text{State}(k_x)) \ \wedge \ \left(\forall_{p=1}^{x} \text{State}(k_p) \in \text{In}(t)\right) \ \wedge \ k_1 \dots k_x \text{ unify}$$

  Let $k = \text{Unify}(k_1, \dots, k_n)$.

- The unified token k satisfies the guard at t i.e., $G(t)$ evaluates *true* in the context of e, B, and k. By this we mean that all the CPA variable references in $G(t)$ are found in k, the event data are found in e and all the other references are found in B.

Note that use of the term "may fire" implies non-determinism. That is, the satisfaction of the conditions listed above are necessary, but not sufficient. The definition also implies concurrency, i.e., any subset of transitions that satisfy these conditions may fire, or that a transition may fire more than once.

Upon firing t, the tokens $k_1 \dots k_x$ are removed from their respective states and the unified token k is placed in all the output states of t, i.e., $\forall s \in \text{Out}(t)$ (when an invariant transition fires, tokens do not unify because $\text{In}(t) = 1 \ \forall t \in TN$.) These tokens may further transit non-deterministically across ϵ labeled transitions. For the special case when any input state of $t \in T$ is a start state of M, i.e., $\exists p \in \text{In}(t) \mid p \in S_i$, k is *duplicated* and its copy is placed in the start state of the invariant, i.e., in SN_s.

The transition function Δ is defined for a sequence of events as the successive compositions of δ. That is,

$$\Delta(\mu, e_1 \ldots e_n) = \Delta(\mu', e_2 \ldots e_n),\ \mu' = \delta(\mu, e_1)$$

A sequence of events $\alpha = e_1, \ldots, e_n$ is recognized by M iff NonZero(F) in the marking $\Delta(\alpha)$ but Zero(NF) in all possible $\Delta(\alpha)$. This realizes the conjuncted negation semantics of invariants.

4.4 Realizing the Intrusion Classification in this Model

This section describes how the abstract classification scheme presented in Section 3.1 fits the model of matching presented in this chapter. The category of RE patterns in the classification can be split further in our matching model. Other categories more or less directly correspond to particular structures in the CPA model. Because the CPA model permits side-effect operations through actions, the catch-all category of "other" patterns can easily be simulated through manipulations of global state via actions. The precise definition of a high-level event that was unspecified in Section 3.1 is defined here.

1. *Existence.* This is defined in our model by specifying a guard expression to be evaluated when the pattern is instantiated. Instantiating a pattern is implementation-dependent. Our implementation technique is presented in Chapter 6.

2. *Sequence.* The definition of a "thing" [cf. Section 3.1.2] is modeled in graphical terms as a DAG with two dominating states, one with no input arcs (referred to as the input dominating state) and one without any output arcs (referred to

as the output dominating state). Furthermore, the maximum number of input and output states of any transition in the "thing," including ϵ transitions, is constrained to one.

A sequence then is a concatenation of "things" where the input dominating state of all high-level events except the first is the same as the output dominating state of the previous high-level event.

For example, to specify the high-level action of writing to a file, one can look for all possible ways of *open*ing a file with the *write* flag specified to the *open* system call. The example in the sequence pattern of Figure 4.3 specifies the concatenation of two "things," a *read* followed by a *write*.

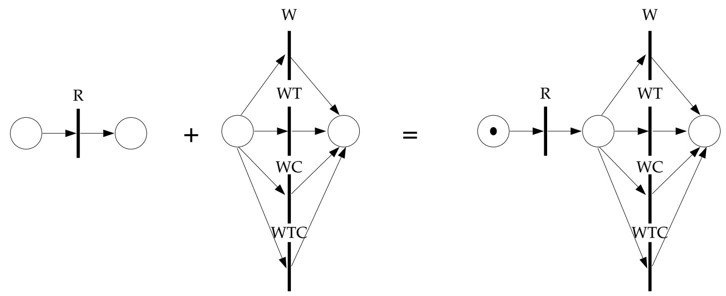

Figure 4.3 A Sequence Pattern of Read Followed by Write

A sequence is a concatenation of "things" with the output dominating state of one thing being the input dominating state of the next thing.

Because the amount of duplication of tokens in the pattern on exercising each event is at most two, the maximum number of tokens in the pattern after m events is 2^m. The number of tokens in the pattern is a measure of the upper bound time complexity of exercising the pattern with an event. Thus, the total time to exercise the pattern with m events is

$$1 + 2^1 + 2^2 + \cdots + 2^{m-1} = O(2^m)$$

RE Patterns. This category can be split further in our matching model into bounded and unbounded output nets.

Bounded Output Nets. These are general nets (need not be DAGS) without AND synchronization in which the maximum number of output states of any transition, including ϵ transitions, is bounded by a constant c, and the number of input states of all transitions is one. With these restrictions, the amount of duplication of tokens in the pattern is bounded by c for any event. Thus the maximum number of tokens in the pattern with n states after observing m events is c^m and the total time to exercise a bounded output net with m events is $O(c^m)$.

An example of this pattern is the result of trying to represent unauthorized transitions to root. This detects cases in which the user id of a process changes without an intervening authorized method (for example, in UNIX, a call to `setuid`) of changing the user id. The kernel long divide emulation code in some Sun operating systems that failed to check the address of the remainder may be detected using such a signature [CER, CA-92:15]. A simplied signature to handle this case is shown in Figure 4.4. When a process is started (the transition labeled with `exec`), its user id is stored in the pattern local variable UID. The signature remains unmatched as long as the process does not change its user id (the output state of the transition labeled `exec`). If the process changes its user id without calling `setuid`, `setgid`, or `exec`, the pattern is matched. If the process calls an authorized way of changing its user id, the invariant triggers the destruction of the partial matches.

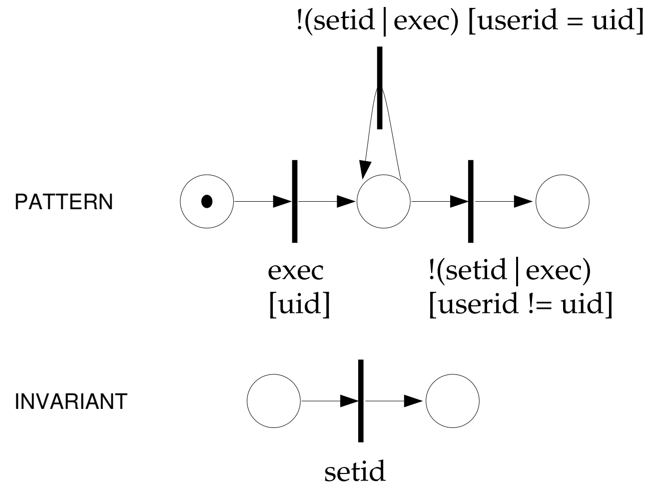

Figure 4.4 A Simlified Pattern to Detect Unauthorized Transitions to Root

Unbounded Output Nets. These are general nets without any constant bounds on the number of input or output states of a transition. Thus, the number of output states of a transition can be as large as n, the number of states in the pattern. The maximum number of tokens in the pattern after observing m events is no longer independent of n. Matching is with AND semantics, i.e. concurrency of exercising more than one transition labeled with the same event type is permitted.

Another category similar to unbounded output nets is that of partial order patterns. In these nets, matching is defined with partial order semantics. Partial order patterns overlap with AND patterns without subsuming them. AND patterns and partial order patterns are the same if no two transitions in a pattern are labeled with the same event. It is because of this direct correspondence that this category is mentioned here.

Other Patterns. The CPA model of representing and matching patterns can be used to model the moves of a Turing machine. A Turing machine has a finite control and an infinite tape on which symbols can be erased and written. A CPA can easily model the finite state of a Turing machine using its own graph, while the CPA global state that can be manipulated through actions can serve the function of the infinite tape. Each move of the Turing machine corresponds to the processing of an input event by the CPA. Thus, the CPA models a Turing machine by matching an event sequence of infinite length while the tape contents (both initial and intermediate) are manipulated through global state changes. By the Church-Turing hypothesis [HU79], the CPA model can compute any computable function.

4.5 Comparison with Other Models of Matching

In comparing our model with other models of matching such as regular expressions, context-free grammars, and attribute grammars, we assume that these models are meant to be used as intuitively and directly as possible in representing intrusion signatures. For example, while attribute grammars are powerful, using them to recognize a set of sentences whose underlying structure does not lend itself to being represented as a context free grammar is not very intuitive and, therefore, not very useful to the human responsible for writing the intrusion signatures.

Regular Expressions. Traditional matching with regular expressions is fast and well understood. Approximate pattern matching involving regular expressions is polynomial in the size of its input. However, regular expressions can only represent extremely simple attack scenarios not involving context and that is their biggest limitation.

Deterministic Context-Free Grammars. By themselves they are of limited use because they cannot handle context. There is no easy way to extend them to match with the "follows" semantics. Deterministic grammars such as LR and LALR are subsumed under their corresponding extensions that provide grammar attributes. The discussion and comparison of the CPA model of matching with attribute grammars, which is discussed below, also applies to deterministic context-free grammars.

Attribute Grammars provide a powerful representation mechanism but, to be useful to humans writing intrusion signatures, the underlying signature specification needs to be context-free. As with context-free grammars, there is no easy way to extend them to match with the "follows" semantics.

Other technical difficulties in directly providing our notion of "context" to attribute grammars are:

- It has not been shown in the literature how partial matches can be efficiently abandoned if the context cannot be satisfied in the traditional model of computing provided in attribute grammars. That is, even though it may be possible to represent a pattern to be matched as a corresponding attribute grammar with a special attribute whose value indicates a successful match, it is not clear how to discard partial matches when the value of the special attribute indicates that a successful match is not possible. The straightforward technique of advancing the input by one and retrying the match is too inefficient because of the maintenance of a stack to match context-free grammars.

- It may not be possible to easily evaluate "context" expressions while generating the parse tree for the grammar. For example, suppose that

$$A \rightarrow BCD;\ P \rightarrow QRS;$$

  are two productions in the grammar. Let B have an attribute x and Q have an attribute y. Assume that the context that must be satisfied for valid sentential forms specifies that $x = y$. Assume further, for simplicity, that P is reduced before A in all valid sentences and that all valid sentences require both P and A.

  When the pushdown automaton that corresponds to the grammar reduces B, it should verify that $x = y$. If $x \neq y$, the automaton should discard the match. But the location of P or Q as an offset from the top of the stack is variable at the time that B is reduced. Furthermore, P might have been reduced further and may require that y be propagated upwards in the parse tree. Building the entire parse tree and then verifying the contextual expressions is contrary to the goal of discarding partial matches as soon as it is discovered that they will not lead to successful matches.

Thus, using attribute grammars to represent and match intrusion signatures complicates the matching procedure considerably. Fixing these problems would change the traditional model of computing with attribute grammars substantially.

4.6 Summary

This chapter introduced our model of matching which is based on Colored Petri nets. Our model is simple and provides for a direct graphical representation of patterns. Externally, a language can be designed to represent signatures in a more programmer-natural framework, and programs in the language compiled to this internal representation for matching. The model is designed to be as limited in its expressive power as possible while still satisfying the requirements presented in Section 3.2 and representing the intrusion classification of Section 3.1. For example, embedded negation of the form $p_1\overline{p_2}p_3$ is not directly representable in the model but we have not needed to use this feature to represent intrusion patterns that we have studied.

5. THEORETICAL PROPERTIES OF THE MATCHING MODEL

In this chapter we present some important theoretical properties of the model shown in the previous chapter. Establishing theoretical bounds on matching in this model is important because it lends credence to the procedure of simulating nondeterministic CPAs, which is one of exhaustive search. Section 5.1 highlights this result by showing that the time complexity of matching in which event data can be *remembered* (for example, in token local variables) for later matching is NP Hard. This means that new algorithms that will reduce the matching time for this problem are unlikely to be found. Section 5.3 shows how some compiler optimization techniques can be applied to improve pattern matching in our model.

5.1 Complexity of Matching

By unification we mean that patterns can specify variables that can match any single event and remember the event they match. These variables, once instantiated, can be used later in the pattern to force a match against events occurring in the input stream. For example, one can specify the pattern $abXcdX$ which means event a, followed by event b, followed by any event, which is stored and made available through the variable X, followed by c, d and the same event that matched the first X.

Property 1: Pattern Matching with Unification is NP Complete.

We show that the problem is NP Complete by reducing the problem of vertex-cover in arbitrary graphs to the problem of pattern matching with unification. This proof is a recast of the proof presented by Aho in [Aho90, Section 6].

The vertex-cover problem for an arbitrary graph G and an integer k is to determine if there is a subset of vertices in G of cardinality at most k such that every edge in G is incident on at least one vertex in the selected subset.

We will construct an event stream E and a pattern P from G in polynomial time such that P matches E iff G has a vertex cover of size $\leq k$. Let # be a distinct marker symbol. The event stream E is a concatenation of two parts (1) A listing of all the vertices in G and (2) A listing of all the edges in G. The purpose of (1) is to force the pattern to initially make a choice of the nodes that comprise the vertex-cover. The purpose of (2) is to verify that the vertices indeed form a vertex-cover. If the vertices of G are labeled $n_1 \ldots n_f$, i.e., the number of vertices in G is f, then (1) is the string $n_1 \ldots n_f\#$ repeated k times. The pattern $V_1\# \ldots V_k\#$, where $V_1, \ldots, V_k$ are variables that are instantiated to the events they match against, will select k nodes in G, perhaps not all distinct when matched against this event stream

To verify that the set of vertices selected in (1) constitute a vertex-cover, we take each edge in G and list it as a pair of vertices, separated by #, in no particular order. For all edge descriptions $n_i n_j\#$, at least one of n_i, n_j must be $\in \{V_i \ldots V_k\}$. For any particular edge, this can be written as the pattern $(V_i|\ldots|V_k\#)$. To verify that each edge in G has at least one endpoint in the vertex-cover, the pattern is simply repeated m times, where m is the number of edges in G.

Concatenation of (1) and (2) makes the pattern P, of size $O(km)$, to be $V_1\# \ldots V_k\#$ $(V_i|\ldots|V_k\#)^m$ while the event stream E, of size $O(kf+m)$, against which P is matched is $(n_1 \ldots n_f\#)^k(\forall_{i=1}^{f}\forall_{j=1}^{f} n_i\, n_j\#)$, $n_i\, n_j$ is an edge in G. The correspondence between the pattern and the event stream is shown below. The # symbols in the pattern and the event stream match one-to-one.

	1	$\cdots$	k	1	$\cdots$	m	
Pattern	$V_1\#$	$\cdots$	$V_k\#$	$(V_i\|\ldots\|V_k\#)$	$\cdots$	$(V_i\|\ldots\|V_k\#)$	
Events	$n_1 \ldots n_f\#$	$\cdots$	$n_1 \ldots n_f\#$	$n_i n_j\#$	$\cdots$	$n_i n_j\#$	$\square$

Because the vertex-cover problem can be represented as a CPA pattern, it is unlikely that *any* CPA matching algorithm can solve it in less than exponential time. This establishes a lower bound for any matching algorithm for CPAs. This means that the worst case performance of any matching algorithm is no better than that of the brute force search of trying to match a CPA against every subsequence (possibly non-contiguous) of input.

Property 2: Remembering event associated data is as at least as difficult as remembering event types. That is, pattern matching with context evaluation is NP Hard.

We show this result by reducing the problem of matching with unification (property 1, denoted here by MU) to the problem of matching with context evaluation (denoted here by MC) in polynomial time. If the pattern in MU is matched over the input event set $e_1, \ldots, e_n$, its corresponding pattern in MC is matched over the single input event e_{mc}. The event e_{mc} has one data element, d, whose value ranges over the set of integers I. An event e_x in MU corresponds to the event e_{mc} in MC with the data value $d = x$.

For all uses of an ununified variable X in MU that matches the event e_x, remember the data value associated with e_{mc} in MC. Let the remembered value be denoted by d'. This data value associated with e_{mc} should be x. For all bound uses of X, use the data value associated with e_{mc} to force the exact match in MC. That is, match $e_{mc} \mid e_{mc}.d = d'$. This transformation establishes a direct correspondence between a problem in MU and a problem in MC. A pattern in MU can be transformed to a pattern in MC in time linear in the size of the pattern. Because MU is NP Complete, MC is NP Hard. □

A consequence of these results is that there are unlikely to exist algorithms that will solve the general problem of matching with unification or context evaluation faster than exponential time in the worst case, which is the time required to exhaustively try to match the pattern in every possible way against the input. In some special cases of the structure of the patterns and the guard expressions matching can be improved. We now investigate these constraints.

5.2 Some Engineering Solutions that Improve Matching

Observation: An exhaustive search may be avoided when matching, if some or all the guards in the pattern are monotone.

Let x_1 be a variable that takes on values over the lattice $(L_1, \leq_{L_1})$. Let the variables $x_2, \ldots, x_n$ be similarly defined over the lattices $L_2, \ldots, L_n$ respectively. Let $e(x_1, \ldots, x_n)$ be a function of the n variables $x_1, \ldots, x_n$ whose values range over the lattice L. e is *monotonic* if e is either non-decreasing or non-increasing with respect to $\leq_L$ as the values of any one or more x_i is monotonically increased or decreased with respect to $\leq_{L_i}$. We refer to a data field as being monotonic if its observed value is non-decreasing or non-increasing with respect to to its domain lattice. Logical expressions in a programming language sense that evaluate to `true` or `false` are defined to be monotonic over the lattice `false` $\leq$ `true` but that choice is arbitrary.

As an example that illustrates this observation, consider Figure 5.1. Without context, the pattern represents the condition $(a \wedge ce)fbd$. The transition β synchronizes the subpatterns a and ce which must both occur before the pattern is matched further. Consider the following guards placed at the following pattern transitions:

At α: `T1 = this[time]`

The successful evaluation of this guard stores the time at which α occurred in the pattern variable `T1`.

At β: `T2 = this[time] &&` $T2 - T1 \ \leq \ 5$

The successful evaluation of this guard stores the time at which β occurred in the pattern variable `T2` and specifies that β should occur within 5 units of time of the occurrence of α.

In the figure, states are numbered, while edges are labeled with alphabets.

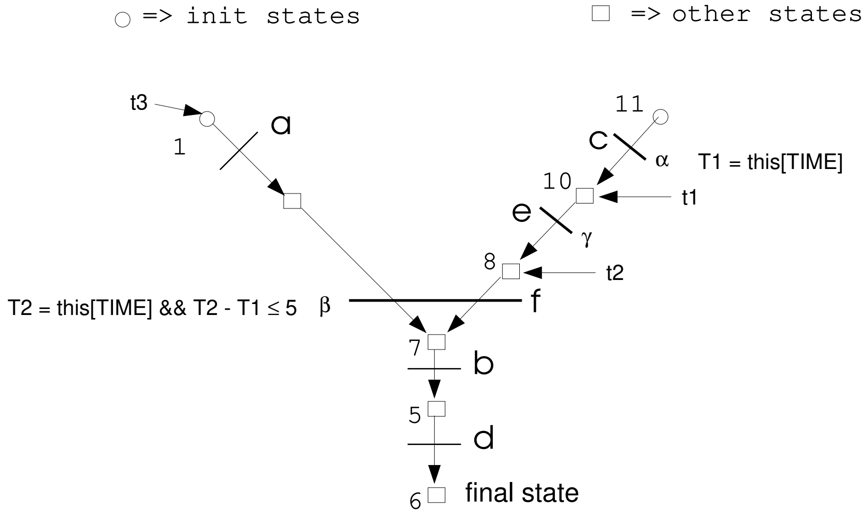

Figure 5.1 A Pattern with Monotonic Guard Expressions

During matching of this graph, consider that there is a token t_1 in state 10, duplicated to t_2 in state 8 and awaiting merging with t_3 in state 1 before transiting to state 7. Assume that the transition β can never fire for the combination of tokens $\{t_2, t_3\}$ because the guard $T2 - T1 \geq 5$ cannot be satisfied for this pair. This indicates that t_1 *need not be duplicated* any further, because any further duplication will only result in a larger value of $T2$, resulting in the continued failure of the guard $T2 - T1 \geq 5$. Finding a match of the pattern does not require exhaustive search because duplication of t_1 can be avoided. This conclusion can be made because the time stamps of successive events are non-decreasing and the boolean expression $e(x) = a + x \leq b$ is monotonic in the sense that if $e(x)$ is false for any $x = l$ then $e(x)$ is false $\forall x \geq l$. This observation is not applicable for non monotonic data fields.

This observation can be generalized as property 3:

Property 3: During matching, whenever a monotonic data field d is defined at state s for token t, t *may* be destroyed if there is a node p *dominating* the final state f on any path from s to f such that the monotonic expression involving d at p cannot be satisfied.

For example, consider the pattern shown graphically in Figure 5.2

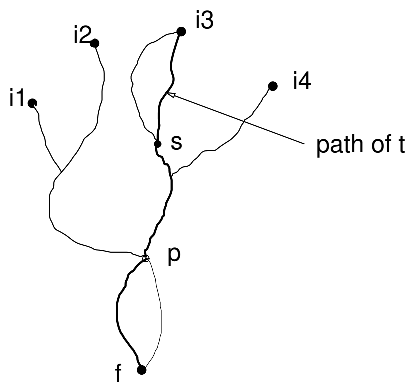

Figure 5.2 A General Pattern with Monotonic Guard Expressions

in which i_1, i_2, i_3, i_4 are initial states, f is the final state, p *dominates* f and the darkened path is the path of token t as it flows to f. The monotonic data field d (of the audit record) is bound to a pattern variable at state s. Because p dominates s, token t must pass through p before it reaches f. However, before reaching p, t might merge with other tokens at intermediate transitions. At each step of the movement of t towards p, copies of token t (and copies of copies etc.) are being moved to further states rather than t itself. If the monotonic condition involving d cannot be satisfied for the copy of t first reaching p, then the condition cannot be satisfied for any of its copies that occupy states in the path between s and p, including t itself (at state s) because of the monotonicity of the expression involving d. Future combinations of the set of tokens that resulted in t may be prohibited, because they will yield a non-increasing or non-decreasing value of d, and depending on the type of monotonicity of the expression at p, may continue to result in its failure. $\square$

This observation can be easily generalized to multiple monotonic fields and monotonic expressions involving only these fields.

Property 4: When any match of the pattern against the input will suffice, tokens can be *moved* instead of *duplicated* from a state s if:

- The only out transition of s is t and
- The only in state of t is s and
- t does not involve unification or make additional bindings to token variables.

Because of these conditions, no pattern variable associated with a token changes when it flows past t. Therefore, duplications do not alter its bindings. The result of expressions evaluated at later transitions are also not affected. Thus, because duplicated tokens traverse the same path, no new solutions are discovered.

It is not permissible to *move* tokens from states that have more than one outgoing transition because moving them involves making a choice of the transition over which they will flow. Thus, if the token is moved across the wrong transition it will be unavailable to match further events from its original state and one of them may be the correct choice. □

5.3 Common Subexpression Elimination in Guards

In this section we describe how we can improve the evaluation of guard expressions by exploiting the commonality in guard subexpressions across all transitions labeled with a particular type. The approach is to evaluate constant subexpressions involving event data only once and using these values when they are referenced again. However, because guard expressions can involve short-circuited expressions that cannot be compiled into a basic block [ASU86] we have modified the traditional notion of common subexpression elimination to work across basic blocks by using a virtual machine that understands the semantics of operations used in guard expressions. In particular, the virtual machine understands which operations cause side effects and which do not.

Consider, as an example, the following attack scenarios (see Figures 5.3 and 5.4) which can be encoded as two separate patterns to be matched simultaneously:

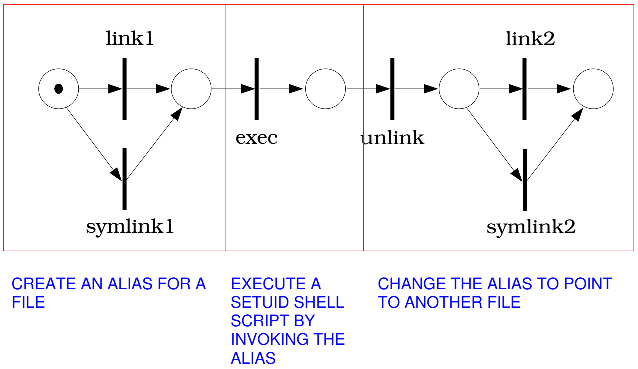

Figure 5.3 A Timing Attack Involving Setid Shell Scripts

```
1. a. ln setuid_shell_script -i
   b. -i

2. a. ln setuid_shell_script FOO
   b. FOO &
   c. rm FOO (200ms <= T.c - T.b <= 1s)
   d. ln your_favorite_shell_script FOO (T.d - T.b <= 1s)
```

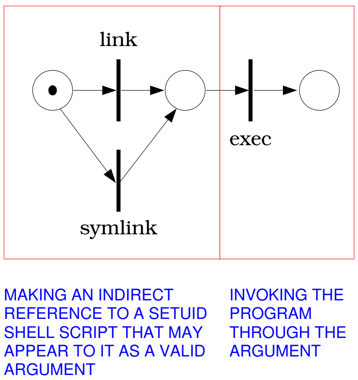

Figure 5.4 Exploiting Setid Shell Scripts

There is considerable similarity between the sub-signatures `1a` and `2a`. For any event of type `LINK`, once one of them is evaluated, the other may not need to be evaluated completely from the beginning. Consider the following decomposition of 1a and 2a.

5.3.1 Compilation of 1a

The `LINK` event provides information about the pathname of the existing file to which the link is being formed as well as the pathname of the newly created link. The guard at 1a ensures that the pathname of the newly created link (stored in FILE2) begins with the character '-' and that the existing file to which the link is made (stored in FILE1) is a setuid shell script file, i.e. the file has its setuid bit set, is executable by *group* or *other* and the first two characters of the file start with "`#!`". The guard also tests if the link is being formed to a file that is not owned by the process forming the link. Otherwise there is little advantage in exploiting this vulnerability. This is checked by testing `U` (stores the EUID of the process issuing the link command) and `T4` (stores the owner of the file to which the link is being formed) for inequality. The guard expression being compiled is:

```
FILE1 = this[SRC_FILE] && FILE2 = this[DEST_FILE] &&
SHELL_SCRIPT(FILE1) = 1 && OWNER(FILE1) != this[EUID] &&
basename(FILE2) = "-*" &&
(FPERM(FILE1) & XGRP = 1 || FPERM(FILE1) & XOTH = 1)
```

1.	THIS ← LINK	All references to the current event are made via THIS.
2.	TRANSITION ← 4	This transition is numbered 4 among all the transitions.
3.	T1^1 ← THIS[SRC_FILE]^1	Indexing may be considered a primitive, polymorphic operation.
4.	FILE1^1 ← T1^1	Global variables are assigned only through temporaries. FILE1 and FILE2 are variables global to the pattern.
5.	T2^2 ← THIS[DEST_FILE]^2	
6.	FILE2^2 ← T2^2	All temporary variables are named T<number>.
7.	T3^3 ← THIS[EUID]^3	
8.	U^3 ← T3^3	U is also global to the pattern.
9.	T4^4 ← OWNER(FILE1^1)^4	`Owner` may be considered a built-in function that returns the owner of a file.
10.	IFEQ T4, U, EXIT	If `T4` matches with `U`, jump to stmt labeled `EXIT`.
11.	T5^5 ← BASENAME(FILE2^1)^5	`Basename` may be considered as a built-in function that gives the filename portion of a full path name.
12.	IFM T5, "-*", EXIT	If `T5` matches "-*" then jump to stmt labeled `EXIT`. The regular expression used here is just illustrative.
13.	T6^6 ← SHELL_SCRIPT(FILE1^1)^6	May be regarded as a built-in function to test if a file is a shell script.
14.	IFEQ T6, 0, EXIT	If `T6` is 0, jump to stmt labeled `EXIT`.
15.	T7^7 ← FPERM(FILE1^1)^7	Built in function giving the permissions of a file.
16.	T8^8 ← AND T7^7, XGRP	XGRP is a constant used to determine if a file is group executable.
17.	IFEQ T8^8, 0, L1	
18.	RES ← 1	Signals a successful evaluation of the guard.
19.	RETURN	Return from this guard.
20.	L1:	
21.	T9^9 ← AND T7^7, XOTH	XOTH is a constant used to determine if a file is executable by others.
22.	IFEQ T9^9, 0, L2	
23.	RES ← 1	
24.	RETURN	
25.	EXIT: L2:	
26.	RES ← 0	Signals an unsuccessful evaluation of the guard.
27.	RETURN	

The functions `owner()`, `name()`, `shell_script()` may be considered mathematical functions in the sense that they return the same value for the same arguments. This assumption is made to illustrate better the common subexpression mechanism. `owner()`, for example, may not strictly be constant in that sense.

5.3.2 Compilation of 2a

The guard to be compiled is:

```
FILE1 = this[SRC_FILE] && FILE2 = this[DEST_FILE] &&
             SHELL_SCRIPT(FILE1) = 1 &&
(FPERM(FILE1) & XGRP = 1 || FPERM(FILE1) & XOTH = 1)
```

$\mathtt{THIS} \leftarrow \mathtt{LINK}$	
$\mathtt{TRANSITION} \leftarrow \mathtt{7}$	This transition is numbered 7 among all the transitions.
$\mathtt{T10}^{10} \leftarrow \mathtt{THIS[SRC\_FILE]}^{10}$	Temporary variable numbers are not reset.
$\mathtt{FILE1}^{10} \leftarrow \mathtt{T10}^{10}$	
$\mathtt{T11}^{11} \leftarrow \mathtt{THIS[DEST\_FILE]}^{11}$	
$\mathtt{FILE2}^{11} \leftarrow \mathtt{T11}^{11}$	
$\mathtt{T12}^{12} \leftarrow \mathtt{SHELL\_SCRIPT(FILE1}^{10}\mathtt{)}^{12}$	
$\mathtt{IFEQ\ T12}^{12}\mathtt{,\ 0,\ EXIT}$	
$\mathtt{T13}^{13} \leftarrow \mathtt{FPERM(FILE1}^{10}\mathtt{)}^{13}$	
$\mathtt{T14}^{14} \leftarrow \mathtt{AND\ T13}^{13}\mathtt{,\ XGRP}$	
$\mathtt{IFEQ\ T14}^{14}\mathtt{,\ 0,\ L3}$	
$\mathtt{RES} \leftarrow \mathtt{1}$	
$\mathtt{RETURN}$	
$\mathtt{L3:}$	
$\mathtt{T15}^{15} \leftarrow \mathtt{AND\ T13}^{13}\mathtt{,\ XOTH}$	
$\mathtt{IFFALSE\ T15}^{15}\mathtt{,\ L4}$	
$\mathtt{RES} \leftarrow \mathtt{1}$	
$\mathtt{RETURN}$	
$\mathtt{EXIT:\ L4:}$	
$\mathtt{RES} \leftarrow \mathtt{0}$	
$\mathtt{RETURN}$	

The superscripted numbers in the instructions above correspond to their value numbers as outlined in Cocke and Schwartz [CS70]. Associating a number with each expression, called its value number, allows the efficient determination of common subexpressions within an expression or in the three address code that corresponds to a basic block. These are compile-time optimizations. For example, to compute $b * b + b * b$ we can avoid computing $b * b$ twice by assigning a value number to $b * b$ when it is first encountered. When $b * b$ is seen again, we know that its value has already been evaluated because it has a value number associated with it. We can therefore use the precomputed value of $b * b$ instead of reevaluating it.

The expression `THIS[SRC_FILE]` is given a single value number because indexing may be regarded as a primitive operation in the virtual machine. Each guard expression begins with an instruction of the form

$$\texttt{THIS} \leftarrow \langle \texttt{type of audit record} \rangle$$

The variable `THIS` is a placeholder name for the audit record currently under analysis for a possible match. This instruction also serves to limit the types of audit records that are tried for a possible match with this instruction sequence. Only an audit record of type `LINK` can possibly evaluate the expressions associated with `1a` and `2a` successfully. We have used two special variables in the compilation: `RES`, whose value determines whether the guard has been evaluated successfully, and `TRANSITION`, which refers to the particular guard transition currently being compiled. `TRANSITION` may be used to index into a vector of transitions in which each element denotes whether the corresponding transition fires.

Doing Common Subexpression Elimination Across Basic Blocks

Combining the set of compiled instructions from all transitions labeled with the same event type is nontrivial. Each guard expression may involve `&&`s and `||`s, resulting in conditional jumps in the compiled code. This complicates static subexpression elimination across jumps, both within and across guards. Common subexpression elimination within a basic block is not useful here as the size of basic blocks is likely to be small, with little redundancy. The approach we are proposing is to always evaluate every basic block so that we can statically precompute all the available subexpressions regardless of the flow of execution of any particular run.

Another important decision is the method of combining the guard expressions. Guards can be combined in a chain with common subexpression elimination performed on the composite sequence, or it may be possible to organize them as a network (similar to Rete networks [For82]) to improve the running time of evaluating them by taking dynamic evaluation into account. When organizing a network, a good configuration needs to be determined as does the duplication and rearrangement of guards, perhaps based on runtime statistics of their evaluation outcome. This is similar to optimizations that use branch prediction to shorten the running time of programs. See Hennessy and Patterson [HP90] for a discussion of branch prediction techniques.

The approach we have taken is to combine the guards in a chain in an arbitrary order and do elimination *across* basic blocks and guards by introducing the notion of active and inactive basic blocks. We do this in a manner similar to how SIMD architectures [Fly66] control which of their processors are active and execute instructions, and which ignore them. SIMD machines broadcast instructions to all execution units, each of which can be disabled during a SIMD instruction [HP90]. In relation to the virtual machine definition, some instructions are treated differently depending on the type of basic block being executed. An active basic block (for a particular evaluation of the guard expression) is one that needs to be executed to determine the value of the guard expression. The virtual machine is said to be enabled when executing an active basic block. Active basic blocks cannot be determined statically because the evaluation of conditional expressions influences its boundaries. Expressions evaluated in an inactive basic block are termed `inactive`.

Evaluating Inactive Basic Blocks

Lack of loops and `goto`s in the guard expressions enable its translation to have forward jumps only. The virtual machine executing the composite code can then treat jumps specially. Instead of jumping to the specified label, it stores the label address and disables itself. When the virtual machine is disabled, certain types of instructions are not evaluated by it. Because all jumps are forward, the machine can be enabled correctly when the jump address is reached, at which point it can resume its normal operation and evaluate every instruction it encounters.

This artifice ensures that all expressions are *always* evaluated, and therefore, are available to expressions evaluated later. This happens whether expressions are active or inactive. All assignments to pattern variables (associated with each token) occur through temporaries and *assignment to non temporary variables is disabled in an inactive region.* This prevents undesired side effects while ensuring that all subexpressions are evaluated and reside in their appropriate temporary variables.

Following the procedure of common subexpression elimination outlined in Cocke and Schwartz [CS70], the code for *both* the guard expressions is as shown here:

```
 1.  THIS ← LINK
 2.  TRANSITION ← 4
2'.  if(!ENABLED_TRANSITIONS[TRANSITION]){
        set processor state disabled
        JUMP 28
```

This has no effect as the processor state is disabled, but serves to store the label at which the processor will be enabled.

```
     }
 3.  T1^1 ← THIS[SRC_FILE]^1
 4.  FILE1^1 ← T1 ^1
```

Assignment to global variables has no effect when the processor is disabled.

```
 5.  T2^2 ← THIS[DEST_FILE]^2
 6.  FILE2^2 ← T2^2
 7.  T3^3 ← THIS[EUID]^3
 8.  U^3 ← T3^3
 9.  T4^4 ← OWNER(T1^1)^4
```

10.	IFEQ T4, U, EXIT	Conditional jumps have no effect when the processor state is disabled.
11.	T5^{5} ← BASENAME(T2^{1})^{5}	
12.	IFM T5, "-*", EXIT	If `T5` matches "-*" then jump to `EXIT`.
13.	T6^{6} ← SHELL_SCRIPT(T1^{1})^{6}	
14.	IFEQ T6, 0, EXIT	
15.	T7^{7} ← FPERM(T1^{1})^{7}	
16.	T8^{8} ← AND T7^{7}, XGRP	
17.	IFEQ T8^{8}, 0, L1	
18.	FIRABLE_TRANSITIONS[TRANSITION] ← 1	This assignment has no effect when the processor is disabled.
19.	JUMP 28	Instead of RETURN. If the processor is enabled, disable it and continue. A JUMP has no effect otherwise.
20.	L1:	
21.	T9^{9} ← AND T7^{7}, XOTH	
22.	IFEQ T9^{9}, 0, L2	
23.	FIRABLE_TRANSITIONS[TRANSITION] ← 1	
24.	JUMP 28	Instead of RETURN.
25.	EXIT: L2:	
26.	FIRABLE_TRANSITIONS[TRANSITION] ← 0	
27.	JUMP 28	Instead of RETURN.
28.	THIS ← LINK	Compiled away.
29.	TRANSITION ← 7	This transition is numbered 7 among all the transitions.

29′.	if(!ENABLED_TRANSITIONS[TRANSITION]){
	   set processor state disabled
	   JUMP beginning_of_next_pattern
	}	

30.	T10^{10} ← THIS[SRC_FILE]^{10}	Compiled away because of value propagation. Same as `T1`.
31.	FILE1^{10} ← T10^{10}	*Not* compiled away because it refers to a pattern variable.

32.	T11^{11} $\leftarrow$ THIS[DEST_FILE]^{11}	Compiled away. Same value as T2.
33.	FILE2^{11} $\leftarrow$ T11^{11}	*Not* compiled away.
34.	T12^{12} $\leftarrow$ SHELL_SCRIPT(T10^{10})^{12}	Compiled away. Same value as T6.
35.	IFEQ T12^{12}, 0, EXIT	T6 value propagated to T12.
36.	T13^{13} $\leftarrow$ FPERM(T10^{10})^{13}	Compiled away. Same value as T7.
37.	T14^{14} $\leftarrow$ AND T13^{13}, XGRP	Compiled away. Same value as T8.
38.	IFEQ T14^{14}, 0, L3	T8 value propagated to T14.
39.	FIRABLE_TRANSITIONS[TRANSITION] $\leftarrow$ 1	
40.	JUMP next_pattern	Instead of RETURN.
41.	L3:	
42.	T15^{15} $\leftarrow$ AND T13^{13}, XOTH	Compiled away. Same value as T9.
43.	IFFALSE T15^{15}, L4	T9 value propagated to T15.
44.	FIRABLE_TRANSITIONS[TRANSITION] $\leftarrow$ 1	
45.	JUMP next_pattern	Instead of RETURN.
46.	EXIT: L4:	
47.	FIRABLE_TRANSITIONS[TRANSITION] $\leftarrow$ 0	
48.	JUMP next_pattern	Instead of RETURN.

`ENABLED_TRANSITIONS` is a vector, each element of which indicates if a particular transition is enabled. `FIRABLE_TRANSITIONS` is also a vector whose elements indicate if the corresponding transition is firable. The percentage reduction in the number of instructions is $\approx 10\%$. Out of 48 instructions, 7 were compiled away while 2 were added ($2'$ and $29'$). This is the case with two guards. Note that all the instructions compiled away are from the second expression. In the asymptotic case, the first few expressions will result in most other subexpression eliminations, and for our example, *may* asymptotically result in a reduction of 6 statements out of 21, which tends to $\approx 28\%$. The compilation of the second guard results in 21 instructions $(28 \ldots 48)$, out of which 7 are compiled away $(28, 30, 32, 34, 36, 37, 42)$ and one added $(29')$. The figures for the reduction in the number of instructions do not imply a corresponding decrease in the execution time of the code, for that depends on the runtime behavior of the conditionals. But, to simplify analysis, an assumption of uniform elimination in every basic block implies a corresponding decrease in the evaluation time of the guards.

Thus, to determine whether the tokens in the initial states of figures 5.3 and 5.4 need to be duplicated and moved across to the succeeding state, we need to evaluate the code for every audit record of type `LINK`.

Is Evaluating Every Basic Block of Every Guard Worthwhile?

This leads to the question of the efficiency of this approach. It is possible that only one guard is true, but this approach would require every expression in every guard to be evaluated. This approach might seem worse than that of evaluating every guard individually because in that case short circuiting might result in fewer expressions being evaluated. We believe that savings can be made with this approach, but the amount is dependent on the type of guard expressions and the commonality among them.

Because of the nature of the matching process, transitions, enabled once, usually continue to remain enabled[1]. Thus, if a guard is evaluated once for an event, it is likely to be evaluated from then on for all events of that type. The approach presented here provides a mechanism to improve matching that can be profitably used given the appropriate set of signatures. A system might incorporate both types of approaches, with and without subexpression elimination and, based on heuristics and runtime statistics, use one approach over the other.

In summary, the following properties are used to ensure the semantic consistency of the expressions or simplify the CSE on the generated code. For a treatment of these and other compiler optimization issues, see the book by Aho et al. [ASU86] or the book by Fischer and LeBlanc [FL88].

[1]Some states may be specified *nodup* [Section 6.3.1], but such states are rare.

1. Token variable values referenced in a guard expression but set outside it are not value-propagated across guards, but re-evaluated from the token the first time they are referenced in the guard.

2. All jumps in the compiled code are *forward*. This can always be arranged by the compiler as there are no loops in the guard expressions. This implies that the structure of a guard expression is a DAG with only *forward edges*. A benefit of this structure is that dead variables can be detected by simply examining the *rest of the code*.

In this dissertation we have not precisely specified a virtual machine and the set of primitives that it understands. Such an attempt was made in [KS]. We have instead shown that the use of a virtual machine can simplify the evaluation of guard expressions in the context of detecting computer intrusions. The practical utility of common subexpression elimination can only be determined by an implementation that measures the overhead imposed by the virtual machine and the interpretation of the virtual machine instruction set.

5.4 Summary

In this chapter we studied the theoretical limitations of matching patterns of the type required for intrusion detection. Traditional pattern matching that does not involve the specification of context is not applicable to intrusion detection. The problem of matching with context, which is a basic requirement to represent intrusion patterns, is NP Hard. This means that an exhaustive search for the solution in the solution space may be required in the worst case. While the theoretical bounds on matching for intrusion detection are exponential, engineering optimizations are possible that may make an implementation more efficient in the usual case. In the context of the model in which patterns may be represented (described in Chapter 4) we described some of these heuristics. These heuristics exploit particular structures of the graphical representation of patterns, the non-decreasing value of time stamps associated with event sequences and the nature of the desired match. We also presented an artifice for doing common subexpression elimination when evaluating guard expressions. This exploits the peculiar nature of the problem domain that evaluates every expression associated with every link with a given label for every occurrence of that event.

6. IMPLEMENTATION ARCHITECTURE OF THE MODEL AND SIMULATION RESULTS

In this chapter we describe the architecture of the prototype we built based on the model described in Chapter 4. The model, together with the prototype presented here, are designed to provide the benefits listed in Section 3.2.1 and meet the system considerations listed in Section 3.2.3. The prototype serves as a proof of concept implementation of the model.

6.1 Introduction

We have used C++ [Str91] as the programming language for the implementation of the prototype. The prototype runs under the Solaris 2.3 operating system and uses the Sun BSM [Sun93b] audit trail as its input. The programming techniques and language features we have used for the implementation are applicable to other programming languages as well. The implementation is directed at providing a set of integrated classes that can be used in an application program to build a generic misuse intrusion detector. The implementation also suggests a possible way of structuring classes encapsulating generic functionality and the interrelationships between the classes to design any misuse detector. This chapter also describes that structure. Measurements of the time and space requirements of the implementation are also presented.

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

- The *free availability* of quality implementations of the language. Not only is this helpful for developing software, it is important for wide-spread acceptance if the implementation is distributed in source form for others to modify and adapt to their environments. We have used the SunPro[1] C++ compiler for our prototype.

- Our *familiarity with C++ and its development environment.* In the interest of building a working prototype quickly, we capitalized on our knowledge of the language and the development environment provided by the SPARCworks[2] workbench.

- The availability of a large collection of *ready-to-use libraries.* Because this is a prototype implementation, we were not unduly concerned with writing highly optimized code specifically tailored for intrusion detection. To perform common tasks including string manipulation, hash table generation, or binary file I/O, we preferred to use ready-made libraries unless these tasks proved to be a performance bottleneck. The availability of quality implementations of such libraries are very useful for rapid prototyping. We have used the Rogue Wave class library Tools.h++ [Sun93a] for our implementation.

- The *linguistic support* provided by C++ to write modular programs. C++ provides for data encapsulation and abstraction in the form of classes and overloaded functions, genericity in the form of templates, object orientation in the form of inheritance and virtual member functions. All of these features have been used extensively in our implementation.

- *Availability of support tools* in the form of grammar recognizer generators like yacc++ and lex++. Because our prototype parses descriptions of patterns into code that realizes the pattern, it was desirable to have parsing tools in the same programming language as the one in which the prototype was written. While it is possible to use a parsing mechanism in any language as a filter that is called as a subprocess, direct sharing of data and functions between the prototype and the parser enabled simplifications.

[1]Trademark of Sun Microsystems, Inc.
[2]Trademark of Sun Microsystems, Inc.

- Because it is easy to add new event streams to the prototype, we have also built a rudimentary matcher for IP datagrams. As a further step, we intended to build the entire matcher as a streams module that interfaces with the networking subsystem. If the prototype is built in a language close to C, the effort of converting it to a streams module [Rag93] would be less.

The set of integrated classes we have developed for misuse intrusion detection can be programmed in many other programming languages as well because no properties specific to C++ have been assumed or used. We only exploit the language's data encapsulation, data abstraction and object-oriented features to simplify the software engineering concerns of our implementation. We use the word *class* in a generic sense and the corresponding notion from many other languages can be substituted here.

6.2 Approach

The implementation of this model can be decomposed into the following subproblems:

1. The external representation of patterns: how the pattern writer encodes patterns for use in matching.
2. The interface to the event source. In our example it would be the interface to IP datagrams.
3. Dispatching the events to the patterns and the matching algorithms used for matching.

These issues are discussed in the next sections. In addition to solving these requirements, our implementation is designed to simplify the incorporation of the following:

- The ability to create patterns and to destroy them dynamically, as matching proceeds.
- The ability to partition and distribute patterns across different machines for improving performance.
- The ability to prioritize matching of some patterns over others.

- The ability to handle multiple event streams within the same detector without the need to coalesce the event streams into a single event stream.

We describe our design in the next section and show how the library classes embody the design. We have included a description of the application interface to the library and the description of patterns because they are important to formulate a comprehensive view of the library.

6.3 Overall Architecture

The library consists of several classes, each encapsulating a logically different functionality. An application program that uses the library includes appropriate header files and links in the library.

The external representation of patterns (sub-problem 1) is done using a straightforward representation syntax that directly reflects the structure of their graphs. These specifications can be stored in a file or maintained as program strings. When a pattern is needed to be matched in an application, a library-provided routine (a Server class member function) is called that compiles the pattern description to generate code that embodies the pattern. This code is then dynamically linked to the application program and the pattern matching for that pattern is initiated. This structure is explained using an example in Section 6.3.1.

The application also instantiates a server for each type of event stream used for matching. Events are totally encapsulated inside the server object (sub-problem 2) and are only used inside pattern descriptions. A pattern may only refer to events from one event stream. When a pattern description is compiled, it is added to the server queue that handles events of that type. The server accesses and dispatches events to the patterns on its queue in some policy specifiable order (sub-problem 3).

The application structure is explained below. Section 6.3.2 describes the structure of events. Section 6.3.3 explains the structure of the server itself in detail and its relationship to the patterns that are instantiated by the application.

6.3.1 Application Structure

As an example application structure, consider matching the pattern described in Figure 6.1. The pattern monitors `rlogin` connections and may be used to detect such connections on a fast gateway by examining each packet that passes through it. The example is chosen from a different event domain to illustrate that the model is independent of the nature of the underlying events.

A TCP connection (an `rlogin` connection is a TCP connection to a specific port) setup between the initiator `S` and the recipient `D` involves a three-way handshake [Com91]. The first segment of the handshake involves sending an IP datagram from `S` to `D` with the SYN bit set in the code field. In response to this SYN packet `D` sends a datagram that acknowledges the SYN packet and sets the SYN bit to continue the handshake. The final message is the acknowledgement of the second SYN and is sent from `S` to `D`.

Thus, to detect simplified TCP connections not involving retransmissions we can monitor for the sequence:

1. A SYN packet, from a source `S` to a destination `D`.
2. A SYN+ACK, from `D` back to `S`.
3. An ACK, from `S` to `D`.

Pictorially this is:

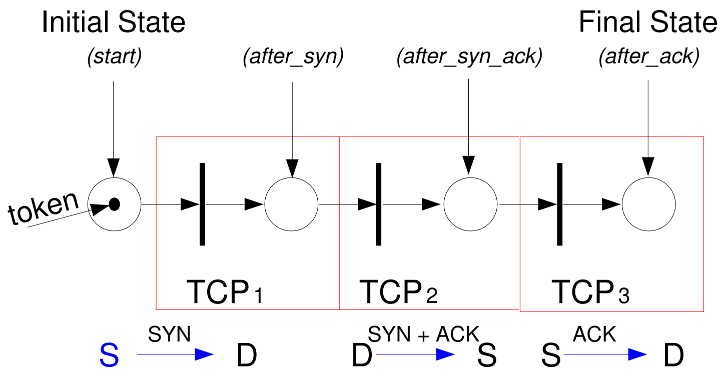

Figure 6.1 Matching a TCP Connection

To monitor `rlogin` connections, we match this pattern for destination ports equal to the rlogin port, which is 513. The application program makes use of an `IP_Server` object. The server object has built into it the layout of events and the event types that can be used in a pattern definition. `IP_Server` also has member functions to access events, in this case from the machine's network interface, and to dispatch them to the patterns that are registered with it. The server is also responsible for parsing pattern descriptions and can type-check the pattern specification because the data format of events are built into the server. The call to the server member function `parse_file` reads, compiles and registers a new pattern with the server object. When the server object is started with a call to `S.run()`, it starts accessing events and dispatching them. An example application code is shown in the boxed text in Figure 6.2.

This consumes one thread of control, as `S.run()` never returns. The server is responsible for implementing concurrency control methods to ensure that calls to its public member functions do not corrupt its internal state when there is an active thread in `run()`. Our implementation uses monitors as described by Hoare [Hoa74] to ensure this. The pattern description contained in file *patterns-ip* is:

```
//file patterns-ip

extern int RLOGIN_PORT_CLIENT, RLOGIN_PORT_SERV,
           print_tcp_conn(int, int);

pattern TCP_Conn_Mon "Monitor rlogin connections" priority 10
  int FROM_PORT, FROM_HOST;
  int TO_PORT, TO_HOST;
```

The variable declarations define the color of the tokens in the pattern. Each token has four integers that can be accessed through the syntax `this[FROM_PORT]`, `this[FROM_HOST]` and so on.

```
  state start;
  nodup state after_syn, after_syn_ack;
  state after_ack;
```

These are the states of the pattern. `after_syn` signifies the state after the initial SYN is observed, `after_syn_ack` signifies the observation of the initial SYN followed

```
//file application.C

#include "IP_Server.h"

int RLOGIN_PORT = 513;

int print_tcp_conn(int from_HOST, int to_HOST) //callback function
{
   cerr << "A TCP connection has been established between "
        << ((from_HOST >>24) &0xFF) << "." << ((from_HOST>>16) &0xFF)
        << "."
        << ((from_HOST >>8)  &0xFF) << "." << (from_HOST       &0xFF)
        << " and "
        << ((to_HOST   >>24) &0xFF) << "." << ((to_HOST>>16)   &0xFF)
        << "."
        << ((to_HOST   >>8)  &0xFF) << "." << (to_HOST         &0xFF)
        << endl;
   return 1;
}

int main()
{
   IP_Server S;

   //read pattern description from file "patterns-ip"
   IP_Pattern *p1 = S.parse_file("patterns-ip");

   /* dup thread of control if necessary. run() doesn't return */
   S.run();

   return(1);
}
```

Figure 6.2 An Example Application

by a response SYN. `nodup` indicates that tokens in this state will not be duplicated to other states, rather they will be moved to other states when the transition fires.

```
post_action { print_tcp_conn(FROM_HOST, TO_HOST); }
```

`print_tcp_conn` is called with token values corresponding to the token in the final state of the pattern.

```
neg invariant first_inv
  state inv_start, inv_final;

  trans rst(TCP)
    <- inv_start;
    -> inv_final;
    |_ { this[RST] = 1 &&  TO_HOST = this[FROM_HOST] &&
          this[TO_HOST] = FROM_HOST;
       }
  end rst;
end first_inv
```

The invariant specifies that no reset should be received during connection formation. An invariant specification can itself be a graph. Whenever a token is moved from the start state of the pattern, its copy is placed in the start state of the invariant. This token can have part of its color defined because the firing of a transition may change a token color.

```
trans tcp_1(TCP) /* TCP is the event type of the transition */
  <- start;
  -> after_syn;
  |_ { this[SYN] = 1 &&  this[ACK] = 0 &&
        FROM_PORT = this[FROM_PORT] &&
        this[TO_PORT] = RLOGIN_PORT_SERV &&
        FROM_HOST = this[FROM_HOST] && TO_HOST = this[TO_HOST];
     }
end tcp_1;
```

If this packet is a SYN packet destined to the RLOGIN port, store its source and destination host and source port in the token.

```
trans tcp_2(TCP)
  <- after_syn;
  -> after_syn_ack;
  |_ { this[SYN] = 1 && this[ACK] = 1 &&
        (this[FROM_PORT] = RLOGIN_PORT_SERV) &&
        (this[TO_PORT] = FROM_PORT) &&
        (this[FROM_HOST] = TO_HOST) && (this[TO_HOST] = FROM_HOST);
     }
end tcp_2;
```

If this packet is a SYN packet from the RLOGIN port of a host whose name matches that stored in the token, destined to the host and port corresponding to this token's variables FROM_HOST and FROM_PORT then fire the transition.

```
    trans tcp_3(TCP)
      <- after_syn_ack;
      -> after_ack;
      |_ { this[SYN] = 0 && this[ACK] = 1     &&
           (this[FROM_PORT] = FROM_PORT)      &&
           (this[TO_PORT] = RLOGIN_PORT_SERV) &&
           (this[FROM_HOST] = FROM_HOST)      &&
           (this[TO_HOST] = TO_HOST);
         }
    end tcp_3;
```

Any non SYN packet flows from (FROM_HOST, FROM_PORT) to (TO_HOST, TO_PORT). This defines the structure of the pattern graph.

```
  end TCP_Conn_Mon;
```

Listing 1: A Sample Pattern Description

Similarly, if an application needed to match patterns against a C2 audit trail it might have used a `C2_Server` instead of `IP_Server` or concurrently with it within the same application program.

6.3.2 Event Structure

Each event in the event stream is converted to an instance of an event class. For IP datagrams this class might be named `IP_event`. This class encapsulates all the attributes common to IP datagrams. Derived classes of `IP_event` may be used for specifying more specialized types of IP datagrams. For example, `TCP_event` and `UDP_event` may be derived to represent TCP and UDP datagrams. Each event object can identify its type through its `type()` member function. This is used by the server to dispatch the event to the appropriate patterns. All the data belonging to the event is made available through its member functions. This mechanism encapsulates the organization of data in the event, which may be system dependent in general. The description of all the event classes is what constitutes the back end of the system and is one of the few system dependent layers.

6.3.3 Server Structure

For each event, the server looks at its type and consults a dynamically-maintained table of patterns that have requested events of that type. It then calls the `Patproc` procedure of each such pattern. `Patproc` is a procedure associated with every pattern (its member function) that processes events for it. This approach to processing events is similar to the approach taken in Microsoft Windows [Pet92]. Events of interest are requested by patterns when they are instantiated by the server.

Events can be dispatched to patterns based on their priority. Patterns can be placed in queues at the appropriate priority level, and patterns serviced in each queue in a round-robin fashion. This ordering of patterns by priority assumes that on the average, an event can be dispatched to all the patterns requesting it in a time less than the average time of generation of an event. If this requirement is not met, patterns up to a certain level in priority may be perpetually starved. A mechanism can be added to age patterns that have not been exercised by any event for a long time by increasing their priority. Pictorially this is as shown in Figure 6.3.

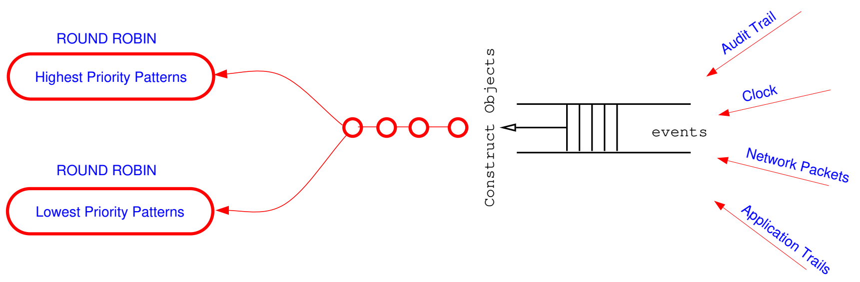

Figure 6.3 Server Structure

The prototype does not implement the priority structure of dispatching events to patterns. It treats every pattern to be of the same priority.

6.3.4 Summary

The use of an event stream requires the creation of two classes. An event class that is the root class of all events provided in the event stream and a server class that parses pattern descriptions, instantiates them, and manages them on its data structures. The server class interacts with the event class by converting raw events into objects of this class and dispatching them. The interrelationship between the various classes is shown in Figure 6.4. Class names bounded by dotted boxes are abstract classes. The functions identified within these boxes are the pure virtual functions of these classes.

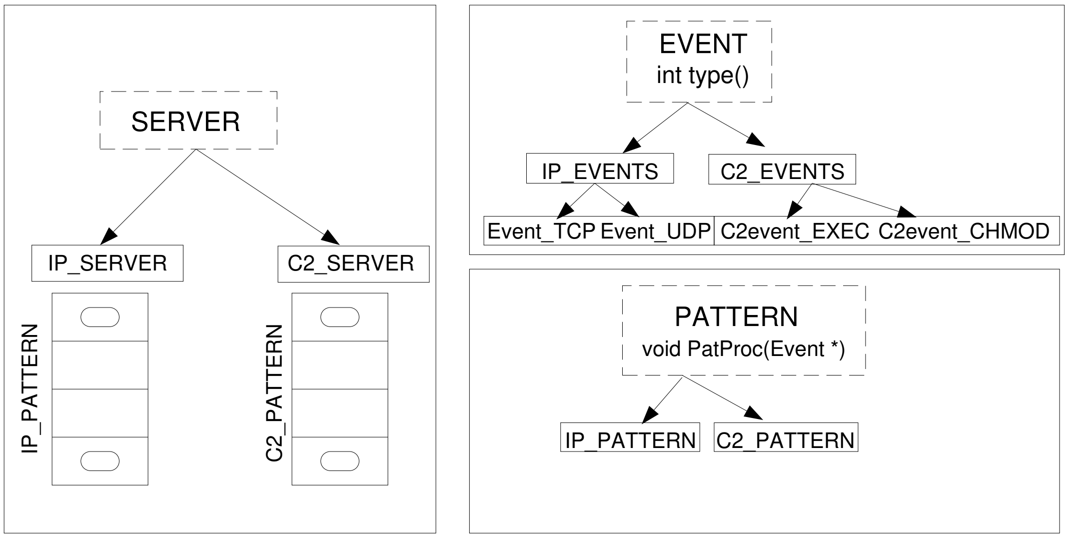

Figure 6.4 Interrelationship Among the Various Classes in the Detector

Use of writable application global variables that can be manipulated by pattern actions or guard expressions obviates parallelism in exercising several tokens simultaneously when several multi-processor threads are available. Several available threads can, however, simultaneously exercise tokens in different patterns. The order in which an event is dispatched by the server to the patterns is undefined. Application global variables should ideally be read-only so that concurrency of access to these variables is possible.

6.4 Building the Server

This section describes how a server class (e.g., IP_Server) is implemented in our library. The event class associated with the server class is completely encapsulated in the server class and is not visible to the application. The heart of the server class is the member function that translates a pattern description into C++ code that implements the pattern [Section 6.4.1]. Because our language for describing patterns is a straightforward representation of the pattern structure, translation into an automaton is direct. Syntactic structures introduced in the language often translate directly into functions that are invoked to perform the operation. Section 6.4.2 describes what the translated automaton looks like, particularly the procedure that accepts incoming events from the server and exercises the automaton with it.

6.4.1 Server::parse()

The server class associated with each event stream is responsible for translating patterns specific to the event stream. For each pattern (each pattern name is unique), the translation performs the following actions:

1. It generates a C++ class representing the pattern (`IP_TCP_Conn_Mon` in our example) with all the pattern global variables as static data members of the class (none in our example).
2. It generates a token class (`IP_TCP_Conn_Mon_Tok`) that represents tokens associated with that pattern). The token class has private data members corresponding to each pattern local variable and corresponding public functions to

access them. In our example these are `FROM_PORT`, `FROM_HOST`, `TO_PORT` and `TO_HOST`. These were declared in lines 5 and 6 of listing 1.

3. Each guard expression associated with a transition is re-written with several syntactic changes:
   - Pattern local variable references are substituted by calls to token member functions.
   - Certain operations are syntactically changed to library calls. For example, the pattern matching operator `=~` is changed to a call to a regular expression matching routine.
   - Calls of the form `this[...]` are changed to member function calls to the appropriate event object. See for example line 24 of listing 1.
4. A `PatProc` procedure is generated for the pattern to handle events for the pattern, in our example its signature would be `IP_TCP_Conn_Mon::PatProc(IP_Event *)`.

6.4.2 Pseudo-code for the Generated PatProc

The heart of a pattern is its `PatProc`, which exercises its automaton on each event that the pattern has requested. Figure 6.5 shows the pseudo-code of a sample PatProc. For each incoming event, all transitions labeled with that type are tested to see if they fire. This requires testing whether the event and the unified token formed by unifying tokens drawn from each input state of the transition satisfy the guard at the transition. All tokens residing in *nodup* states that comprise the unified token are marked for later deletion. Tokens that are added to output states of a transition as a result of its successful firing wait to be added to the states until all transitions have been tried. Then the tokens are added into all the states. When an invariant is satisfied, i.e., a token reaches the final state of the invariant, all the tokens related to the token are destroyed.

```
IP_TCP_Conn_Mon::PatProc(Event *e)
{
   for(all transitions in pattern and invariants of type e->type())
   {
      for(all token sets formed by taking one token from each
                                          input of this transition)
      {
         if(the token set does not unify)continue;
         if(the token set fails the guard)continue;
         mark all tokens in this set belonging to nodup states
                                                    for deletion;
         put a copy of this token in each successor of this transition;
         if(one of the input states of this transition is a pattern
                                                          start state)
            put a copy of this token in the start state of each invariant;

      }
   }

   clock the states to merge tokens waiting at its input
                        with tokens already in the state;

   eval post actions for all tokens in the final state and free them;

   delete all marked tokens from all nodup states;
   process all invariant final states;
}
```

Figure 6.5 Pseudo-code of a Sample PatProc

6.5 Design Choices

By far the most significant consideration guiding the design was the runtime efficiency of the detector. For misuse detection using a C2 generated audit trail one might reasonably expect to process events (audit records) at the rate of 50K-500K/user/day [Sma95]. Furthermore, any computer resource required for matching patterns reduces the availability of these resources for general use. We therefore decided not to interpret the pattern automata by using table lookups to determine the pattern structure but instead, to compile the pattern description into an automaton. This also has the benefit of compile-time optimizations of guard expressions present in the pattern.

We tried to make the generated code realizing the automaton efficient by using functions as little as possible to avoid function call overhead in cases where functions could not be inlined. This often meant that data structures manipulated by the various pieces of the generated automaton were not encapsulated and were manipulated directly by these pieces. This has not resulted in code that is complex and difficult to understand. The routines that generate this "program" are structured and the generated program logic can be deciphered by following the structure and logic of the generating routine.

The overriding constraint of efficiency combined with the requirement to dynamically create and destroy patterns meant that automaton descriptions be compiled and dynamically linked for matching. An additional benefit of the dynamic creation of patterns is that new patterns can be created within an executing program based on its logic and execution flow. For example, it might be desirable to instantiate specific patterns for matching based on the type and degree of suspicious activity observed. Such patterns may depend on the particular user and other specifics of the suspicious activity.

Our design, which is based on the model of dispatching events to patterns lends itself naturally for distribution. In a distributed design, the event sources (audit trails) may be generated on different machines and their processing on another machine. That is, the patterns, the server and the event sources may all reside on physically different machines. The server can then retrieve events by using any of several well-known techniques such as remote procedure calls [BN84] or distributed objects [Par90] and dispatch them to patterns. Although our current implementation is single host based, a distributed implementation should be straightforward.

Our current implementation requires that patterns be exercised sequentially on events. It does not permit more than one event to be exercised concurrently within a pattern. We do not consider that to be a significant limitation because concurrency can be exploited by exercising more than one pattern on the same event. In a system where the expected number of patterns are of the order of a hundred, this does not seem to be a stringent limitation.

A limitation of the current design is that patterns cannot directly use more than one event source. To use more than one event source, the disparate sources need to be canonicalized to one event stream and used in the patterns. Many modern audit trails, for example the Sun BSM mechanism [Sun93b], allow the creation of user defined event types and applications can generate their own specific audit records through an API.

6.6 Performance

The experiments described below were done on a Sun SPARCstation 5 with 32MB of memory running Solaris 2.3 under light load. The audit file was generated separately by enabling auditing and simulating exploitations manually and under program control. Auditing was enabled with the default configuration, which logs all successful as well as failed events. The pattern descriptions were translated into C++ code and compiled separately. The running times mentioned below represent the reading of the audit file, conversion of each audit record into an object, and dispatching the event to all the patterns that request that event. It does *not* include the time for the matcher to load and begin execution, nor does it include the time to dynamically link the patterns.

The following graphs show performance figures when the patterns are exercised in the system.

6.6.1 Timing Results

Figure 6.6 shows how much time it took to match each pattern against an audit file of approximate size 400KB[3]. Each sample point in the figure is the mean value

[3]KB in this section means 1000 octets.

of 200 runs. The circle at the end of each vertical bar serves to highlight the end of the bar. This is the value of the point being plotted. The little horizontal lines on either side of this point represent the standard deviation of the value over 200 runs.

The audit file contained 2514 events. The sample point $(0, 5.17)$ in the figure represents that the application took 5.17 seconds to create all the event objects and destroy them. The mean time for the creation and deletion of an audit trail event is then $5.17/2514 = 2.1$ milliseconds. This is the fixed cost per event for the system. The point $(1, 5.45)$ means that pattern numbered one (numbered arbitrarily) took 5.45 seconds when exercised by the 2514 events. Some patterns take little time, slightly more than what it took to run with no patterns. The reason for this is that the type of events used in the pattern occurred so infrequently in the event stream that the cost of exercising the pattern on those events was negligible.

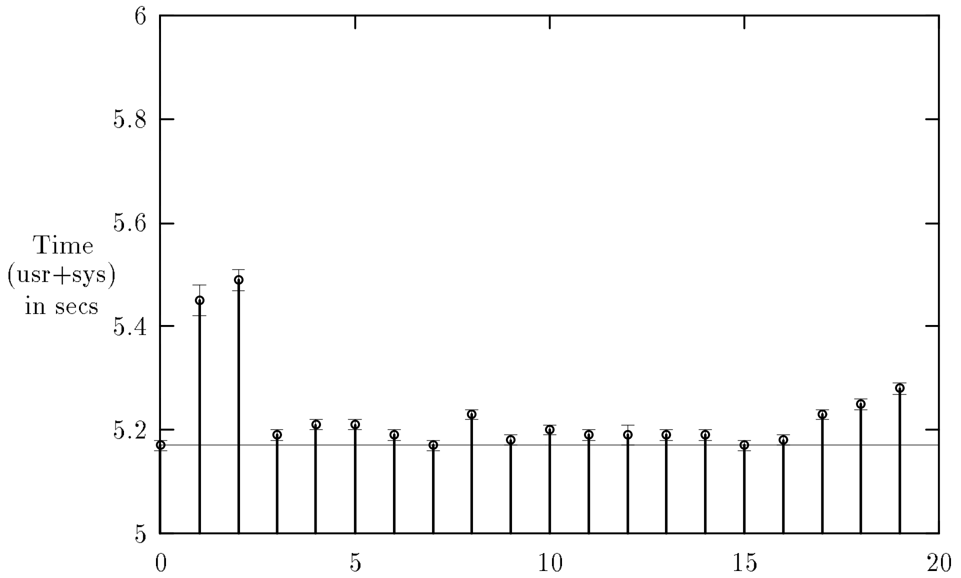

Figure 6.6 Time for Matching Each Pattern for a 400K Audit File

Figure 6.7 shows the simulation time when more than one pattern was matched simultaneously in the detector. The event stream and the pattern numbers are the same as in the previous simulation. In the figure, the data point $(3, 5.74)$ shows that it took 5.74s to exercise the three patterns $1, 2, 3$ together in the system.

The simulation to determine the cost/event/pattern of running multiple patterns together in the detector are shown in Figure 6.7. The fixed overhead cost of reading the audit file and converting each audit record into an object is the same as above, the varying cost that takes the multiplicity of patterns into account is:

$$\text{variable cost/event/pattern} = (5.91 - 5.17)/(2514 * 19) = 15\mu s$$

This uses the data point $(19, 5.91)$ which indicates that the detector took 5.91s to exercise 19 patterns together against an audit trail that consisted of 2514 events.

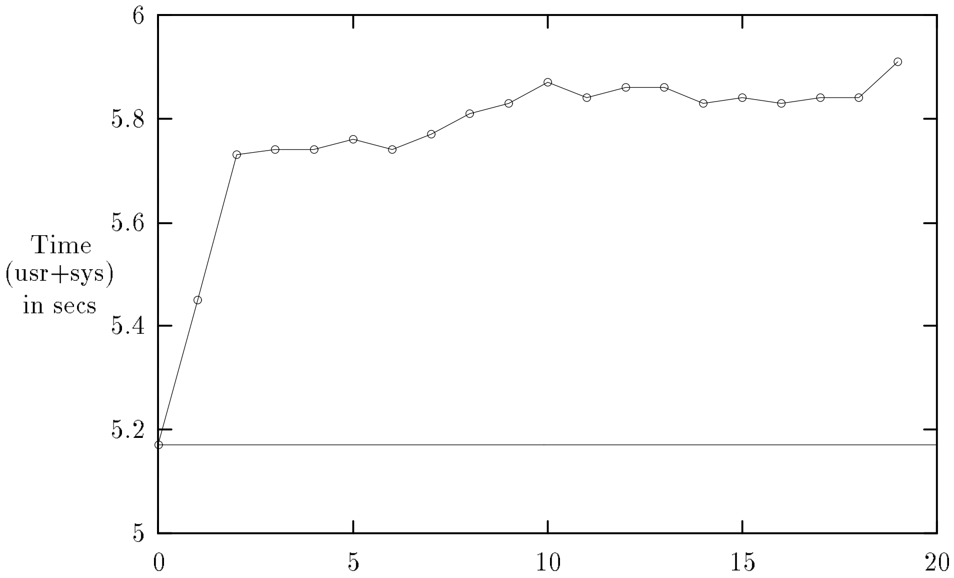

Figure 6.7 Time for Matching Multiple Patterns for a 400K Audit File

Consider the extrapolation of these results to estimate the performance of the detector in a real setting. When running a set of programs in sequence that saturated the CPU, the Sun auditing subsystem generated about 1MB every 10 minutes on a single-user workstation. This extrapolates to 6MB per hour, or $2514 \times 6/.4 \approx 38$K events per hour. Consider that there are 100 patterns in the detector. Then, for one hour of intense CPU activity, the detector might require the following time to process the generated audit data:

Fixed overhead	=	$5.17/2514 \times 38000$s	= 78.15s
Variable overhead	=	$15\mu s \times 100 \times 38000$	= 57s
Total time	=		= 135.15s

Table 6.1 Extrapolating Timing Results to Match 100 Patterns

Thus, for every hour of intense activity, the detector requires $\approx 135s$ to match 100 patterns. This fraction is $135/3600 \times 100 = 3.75\% \approx 4\%$ of the hourly activity. These results correspond to an unoptimized version of the detector.

6.6.1.1 Analysis

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

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

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

For our experiments this is not a factor because 1MB every 10 minutes is $\approx$ 2KB/s. However, this effect can be taken into consideration in cases where it is true.

Better Tuned Auditing. The experiments reported in the previous section were done using an audit trail that logged all events. If the audit events are selected so that only events referenced in the patterns are logged, then it is conceivable that the average time to exercise each event against all the patterns will increase. If, in the untuned case, the disk logging capacity was being saturated, then it is conceivable that the rate of audit data logging remains the same with a more finely tuned auditing. This implies a performance degradation in going from the untuned to the tuned case. However, this also means that the system is being better utilized.

6.6.2 Space Requirements

The space requirement of each pattern is depicted in Figure 6.8. The mean size of the patterns is 17KB.

There are several factors involved in this mean pattern size. The most significant reason is that the pattern structure is not saved in memory to be used by a common pattern simulation routine. Instead, the pattern is compiled into its structural description, which makes each transition responsible for evaluating its guard and moving tokens from its input states to output states. This results in substantial duplication of code, once for each transition.

The other reason is that support structure for the implementation of each pattern included dynamically expanding tables and linked lists. Because each pattern

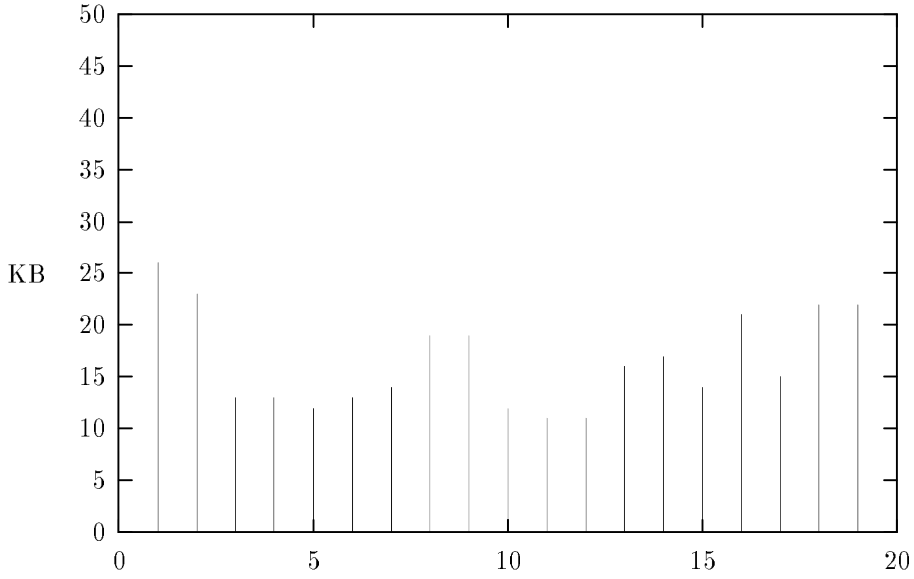

Figure 6.8 The size in KB of Each Compiled Pattern

is logically different, with a different definition of a "token," these structures were replicated in each pattern definition.

These problems are purely a manifestation of our non-optimized implementation, and not a limitation of the model. To reduce the space required per pattern, all the common support code comprising tables, singly and doubly-linked intrusive and non-intrusive lists can be collected in abstract base classes and specific classes derived for each pattern dynamically at runtime. This provides type safety because of derivation, while reusing the common code across all derivations. The same strategy can be applied to reduce the amount of code needed to exercise transitions. Instead of compiling the evaluation of each transition and its guard separately, transitions can store their input and output states in their member data. Then, a common procedure may be used to exercise any transition the same way. Thus, significant space benefits can result from "simulating" the pattern.

6.7 Summary

This chapter described a possible architecture for structuring a misuse intrusion detector based on pattern matching. The structure is client-server[4] based in which the server obtains events and dispatches them to clients (patterns) that implement the matching procedure specific to their structure. Implementing this structure as a library permits embedding this type of matching within application programs. The prototype allows the dynamic creation of patterns. These patterns are translated from a description language into C++ code that realizes the pattern and dynamically links that code into the application.

[4]We use this term in a slightly different way than is used in networking. In networking, it usually denotes a lock-step query/response in which the client makes queries to a server and the server responds with a reply. In our structure, the client (pattern) makes an initial request to the server to notify it of the event types that it needs to exercise the pattern, and then receives those events indefinitely from the server.

7. SUMMARY, CONCLUSIONS AND FUTURE WORK

This dissertation presents a solution to the problem of representing and detecting computer intrusions. The representation problem is one of encoding intrusions in a generic, alias-free manner. The basic goal of our work has been to adequately represent and efficiently detect the majority of intrusions commonly reported rather than attempt to represent and detect every conceivable intrusion scenario.

We have been successful in achieving these goals. We have been able to represent $\approx 88\%$ of the intrusions described in [Bis83, CER, FS90] that have occurred in UNIX systems. We could not detect $\approx 12\%$ of those vulnerabilities but we argue that signature detection is not a good choice for such vulnerabilities. Some require more sophisticated anomaly detection while others, such as passive wiretapping, do not exhibit a detectable signature [cf. Section 1.2.1].

Based on a prototype implementation of our approach, we derived performance results that indicate that our approach of using pattern matching to represent and detect computer intrusions is practical. Intrusion detection systems have not gained widespread acceptance predominantly because of their space requirements and the performance impact that is incurred while running them with regular system activity. We have shown that it is practical to run an intrusion detection system based on pattern matching, concurrently with other user activity on single-user machines, without undue degradation in performance. Extrapolation of our experimental results show that the overhead of matching a significant number of patterns simultaneously against an audit trail that is generated under heavy load on a typical workstation should be under 5% of the system CPU performance.

Furthermore, using pattern matching libraries to monitor intrusion signatures provides a simple, embeddable, and elegant mechanism for intrusion detection. It provides a more natural interface to the representation of signatures than other techniques that need knowledge of specialized tools such as expert system shells. Because of the low overhead imposed by our approach, it is practical to debug patterns in a real environment by adding them to the detector while it is running. the ability to incrementally add patterns also helps in the maintenace of the intrusion detector. Maintenance of signatures is also easier because they can be maintained in text files and as programming language strings. This allows them to be manipulated using familiar tools such as text editors.

7.1 Experiences

We learned several lessons from this effort, which are summarized under three categories below:

7.1.1 Using Pattern Matching for Intrusion Detection

Extended[1] *regular expressions augmented to provide context matching and follows semantics are adequate to detect a majority of commonly occurring intrusions.*

This is a surprisingly simple yet powerful result. However, because the problem of matching with context is NP Hard, there are no known solutions that solve the general version of the problem efficiently. In practice, however, an exhaustive search works well for the patterns that are needed to detect intrusions.

Pattern matching may be inappropriate for representing ill-defined intrusions.

Pattern matching provides an efficient mechanism for the detection of well-defined patterns. To deduce abstractions indicative of intrusions from uncertain information, other mechanisms such as expert systems may be more appropriate. For example, if an intrusion is indicated only when activities occurring at various locations and times are correlated, it may not be easy to specify all the correlations in a compact way using pattern matching alone.

[1]Those that permit the use of AND directly.

7.1.2 Writing Intrusion Patterns

Distilling an incident report into a pattern is involved.

It is nontrivial to translate advisories, for example CERT advisories [CER], into patterns that can reliably detect those and similar incidents. The process requires a good understanding of the key essentials of the exploitation to enable the problem to be abstracted and represented in a generic, alias-free manner.

Writing efficient signatures may require knowledge of the underlying matching model.

Once a vulnerability is clearly understood it must be written as a pattern description. Often, there are several ways of writing the same pattern that can result in different matching efficiencies. The most important fact to bear in mind is the correct use of invariants [Section 3.2.2] that can delete unneeded tokens from the pattern graph. The presence of unneeded tokens can degrade runtime performance because the matching procedure is one of exhaustive search.

Temporally ordered event sequences may make pattern representation much simpler.

Consider as an example the audit trail generated by older versions of SunOS without the BSM patch. The `EXECVE` record that indicated the start of a new process could be recorded in this trail *after* the new process began executing and had generated part of its own log. Thus, if a pattern was intended to monitor a specific activity of a certain *program* running with particular privileges there was no efficient way to ascertain if the monitoring conditions had been satisfied, without looking ahead to retrieve the `EXECVE` record and its associated data. Patterns were written to always match for the desired conditions and then ascertain based on the `EXECVE` record if the match was to be kept or discarded.

In contrast, if the log was temporally sequenced, a pattern to monitor the conditions could be much simpler. Pre-processing the audit trail can be done to make it temporally ordered, but this requires extra overhead and may not be feasible to do at current audit generation rates without an impact on the real-time performance.

7.1.3 Using Audit Trails

Application-level auditing is important.

Because audit logs only provide the events that are executed by programs and not the information manipulated as a result of performing those events, it is not always possible to deduce the actions of a program. Furthermore, it is extremely difficult to invert a low level audit log into higher level program abstractions which are often application dependent. Because intrusions are defined with respect to policies that are tied closely to application-provided abstractions, it is extremely difficult to determine if the policy is violated unless application abstractions can be deduced and used to base these decisions.

Even if the inversion was possible, it is likely to be computationally expensive and perhaps needless. Because applications often provide features that correspond directly to user-level abstractions, they are usually the best place to generate the audit events.

Auditing must provide a reliable means of detecting higher level events.

For example, in some audit trails it may not be possible to reliably detect when a process has exited. This may result in many "garbage" tokens in the matcher that may remain uncollected. This can result in poorer runtime performance.

7.2 Future Work

Our work can be pursued further in one or more of the following directions:

7.2.1 Optimize the Current Implementation.

Interpret patterns. Our prototype implementation described in Chapter 4 compiles a CPA directly into C++ code that realizes the automaton. The implementation does not store the pattern graphical structure in memory to use for pattern matching. Instead, the pattern is compiled into its structural description, which makes each transition responsible for evaluating its guard and moving tokens from its input states to output states. This results in considerable duplication of code, once for each transition because each transition has a different guard expression and a different set of input and output states.

It might be interesting to store the structure of the patterns in memory and interpret the guard expressions to simulate the behavior of the compiled CPA. This would permit a single common simulation procedure to simulate all patterns resulting in a much smaller code size of CPAs and the matching subsystem. It would be interesting to investigate this time/space tradeoff.

Determine Good Order of Combination of Guards. When the guards corresponding to all transitions with the same event label need to be evaluated, it is not clear what is the best order in which to enumerate, and thus evaluate them. There are no semantics to the expressions forming each guard in our model. Theoretically, all the expressions in all the guards need to be combined in a specific order to achieve maximum overlap between the expressions. This overlap is dependent on the probability of occurrence of each expression. Runtime statistics based on the history of evaluations can be maintained to estimate these probabilities. Rete network generation may have applicability to this problem and can perhaps be used to compile the guards in a particular order to achieve better performance.

Investigate a Token Replacement Policy. The Colored Petri net model of matching described in Chapter 4 uses states and transitions to describe the match. In this model, states may have an arbitrary number of tokens resident within them. It may not always be possible to permit this in practice because of memory limitations. It might be possible that matching is not adversely affected if the capacity of states to hold tokens is restricted and some replacement scheme put into effect that determines and discards tokens when adding a new one. It might be that common intrusions of interest follow a locality of attack rule with respect to the tokens, thus resulting in more efficient match procedures.

Determine Practical Benefits of Optimizations. It might be worthwhile investigating through experimentation if the engineering shortcuts presented in Chapter 5 have a significant benefit on the performance of a system structured around them. It may be that these shortcuts yield good performance benefits for the kinds of patterns that are currently needed to detect intrusions.

7.2.2 Add Other Features To The Implementation.

Investigate Feasibility of Kernel matching. It might be possible to embed a highly optimized matcher inside the kernel. This means that system call invocations no longer need to write audit data to disk. Instead, the detector can be exercised in the kernel at the point of the call. This results in an intrusion detector that can be truly real-time because kernel matching removes the latency between the occurrence of an event and its notification to the detector.

Provide a Friendly Interface to Help Develop Patterns. It has been our experience that encoding patterns into our description language requires expertise and experience. It would be beneficial to provide a GUI interface to assist users in specifying and editing patterns.

7.2.3 Apply the Pattern Matching Approach to Other Problems.

Investigate Applicability to Distributed Intrusion Detection. How applicable is the pattern matching approach to detecting intrusions that can only be detected by correlating and analyzing information from several sources? For example, our approach does not bind any semantic meaning to events or the data associated with events. If event data is augmented to provide a host name and other machine related information fields, can patterns be devised that only treat them as syntactic entities yet work well and efficiently? Can the fundamental problem of tracking changed identities across `rlogin/rsh/telnet` be done using patterns without operating system support? How can heterogeneity be handled using patterns? Does clock skew across machines complicate the description of patterns hopelessly?

All these questions and more need to be addressed before the viability of the pattern matching approach to the distributed case can be established.

Investigate Applicability to Specifying Application Level Security Policies. Many security experts believe that low level read-write based access control policies on system objects are inadequate to meet complex application security policy requirements. That is, applications cannot always specify correct subject-object access behavior using permissible read and write requiremenets on system objects. Before applications can specify policies that may be monitored on their behalf by the kernel, there must be a framework to specify these policies in a generic way. It may be possible to use the framework of pattern specification developed in this dissertation for that purpose.

7.3 Conclusions

We believe that this dissertation has advanced current knowledge in intrusion detection by providing insights into the representation and detection issues of a solution using pattern matching. Some of these insights form the crux of Section 3.2 that presents key requirements for any pattern matching solution. The other major contribution of the thesis is the hierarchy of intrusion signatures presented in Section 3.1. This hierarchy is new in that researchers have thus far focused on the classification of vulnerabilities rather than the characteristics of observable events that provide detection capability of these vulnerability exploitations. With this hierarchy one can refer to intrusion signatures as belonging to a particular class in the hierarchy, which suggests the runtime detection characteristics of the signature. The hierarchy also offers a different way of viewing intrusion detection, namely in terms of the types of patterns that can be used to detect intrusions, instead of the generic "anomaly" and "misuse" approaches.

Pattern matching has yielded an efficient mechanism for the detection of intrusions of common interest as evidenced by our experimental results in Chapter 6. Our implementation also suggests a new way of structuring intrusion detection systems, namely as libraries that can be embedded in applications and that use a call-back mechanism to invoke application functions. We believe that this thesis has provided a new approach to intrusion detection and hope that it will spur further work in this direction.

BIBLIOGRAPHY

BIBLIOGRAPHY

[8lg] 8lgm electronic mailing list. Can be retrieved from fileserv@bagpuss.demon.co.uk.

[A+76] R. P. Abbott et al. Security Analysis and Enhancements of Computer Operating Systems. Technical Report NBSIR 76-1041, Institute for Computer Science and Technology, National Bureau of Standards, 1976.

[Aho90] Alfred V. Aho. Algorithms for Finding Patterns in Strings. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science – VOL A*, Chapter 5, pages 256–300. Elsevier Science Publishers, 1990.

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Massachusetts, 1974.

[And80] J. P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P Anderson Co., Fort Washington, Pennsylvania, April 1980.

[Asl95] Taimur Aslam. A Taxonomy of Security Faults in the Unix Operating System. Master's Thesis, Purdue University, Department of Computer Sciences, August 1995.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, 1986.

[Bez83] Boris Bezier. *Software Testing Techniques*. Electrical Engineering/Computer Science and Engineering Series. Van Nostrand Reinhold, 1983.

[Bib77] K. J. Biba. Integrity Constraints for Secure Computer Systems. Technical Report ESD-TR-76-372, USAF Electronic Systems Division, Bedford, Massachussetts, April 1977.

[Bis83] Matthew Bishop. Security Problems with the UNIX Operating System. Confidential Technical Memo, Department of Computer Sciences, Purdue University, January 1983.

[Bis95] Mathew Bishop. UNIX Security: Threats and Solutions. Invited talk given at the 1995 System Administration, Networking, and Security Conference, April 24–29, 1995.

[BK88] David S. Bauer and Michael E. Koblentz. NIDX – An Expert System for Real-Time Network Intrusion Detection. In *Proceedings of the Computer Networking Symposium*, pages 98–106. IEEE, New York, New York, April 1988.

[BK89] Morris I. Bolsky and David G. Korn. *The KornShell Command and Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[BL73] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, The MITRE Corporation, Bedford, Massachussetts, May 1973.

[BN84] Andrew D. Birrell and Bruce Jay Nelson. Implementing Remote Procedure Calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bug] Bugtraq electronic mailing list. Issued electronically from bugtraq@crimelab.com.

[BYG89] R. A. Baeza-Yates and G. H. Gonnet. A New Approach to Text Searching. In *Proceedings of the 12th Annual ACM-SIGIR Conference on Information Retrieval*, pages 168–175, Cambridge, Massachusetts, June 1989.

[CER] CERT Advisories. Available by anonymous ftp from cert.sei.cmu.edu:/pub/cert_advisories.

[Cha91] Eugene Charniak. Bayesian Networks Without Tears. *AI Magazine*, pages 50–63, Winter 1991.

[Che88] K. Chen. *An Inductive Engine for the Acquisition of Temporal Knowledge.* Ph.D. Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1988.

[CHS91] Peter Cheeseman, Robin Hanson, and John Stutz. Bayesian Classification with Correlation and Inheritance. In *12th International Joint Conference on Artificial Intelligence*, August 1991.

[CKS+88] Peter Cheeseman, James Kelly, Matthew Self, John Stutz, Will Taylor, and Don Freeman. Autoclass: A Bayesian Classification System. In *Proceedings of the Fifth International Conference on Machine Learning*, pages 54–64. Morgan Kaufmann, June 1988.

[Coh87] Fred Cohen. Computer Viruses – Theory and Experiments. *Computers & Security*, 6:22–35, 1987.

[Com91] Douglas E. Comer. *Internetworking with TCP/IP*, Volume I. Prentice Hall, Englewood Cliffs, New Jersey, Second edition, 1991.

[CS70] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version.* Courant Institute of Mathematical Sciences, New York, 1970.

[CW89] David D. Clark and David A. Wilson. Evolution of a Model for Computer Integrity. *Report of the Invitational Workshop on Data Integrity*, September 1989.

[Den82] Dorothy E. Denning. *Cryptography and Data Security.* Addison-Wesley, Reading, Massachusetts, 1982.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. In *IEEE Transactions on Software Engineering*, Number 2, page 222, February 1987.

[Doa92] Justin Doak. Intrusion Detection: The Application of Feature Selection – A Comparison of Algorithms, and the Application of a Wide Area Network Analyzer. Master's Thesis, Department of Computer Science, University of California, Davis, 1992.

[FHRS90] Kevin L. Fox, Ronda R. Henning, Jonathan H. Reed, and Richard Simonian. A Neural Network Approach Towards Intrusion Detection. In *Proceedings of the 13th National Computer Security Conference*, pages 125–134, Washington, DC, October 1990.

[FL88] Charles N. Fischer and Richard J. LeBlanc. *Crafting a Compiler.* Benjamin/Cummings, Menlo Park, California, 1988.

[Fly66] M. J. Flynn. Very High-Speed Computing Systems. *Proceedings of the IEEE*, 54(12), December 1966.

[For82] Charles L. Forgy. RETE: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem. In *Artificial Intelligence*, Volume 19. 1982.

[FS90] Daniel Farmer and Eugene H. Spafford. The COPS Security Checker System. In *Proceedings of the Summer Usenix Conference*, pages 165–170, June 1990.

[Gia92] Joseph C. Giarratano. *Clips Version 5.1 User's Guide.* NASA, Lyndon B. Johnson Space Center, Information Systems Directorate, Software Technology Branch, March 1992.

[GL91] T. D. Garvey and T. F. Lunt. Model based Intrusion Detection. In *Proceedings of the 14th National Computer Security Conference*, pages 372–385, October 1991.

[GS91] Simson Garfinkel and Gene Spafford. *Practical Unix Security.* O'Reilly and Associates, Sebastopol, California, 1991.

[HCMM92] Naji Habra, B. Le Charlier, A. Mounji, and I. Mathieu. ASAX: Software Architecture and Rule-based Language for Universal Audit Trail Analysis. In *Proceedings of ESORICS 92*, Toulouse, France, November 1992.

[HLM91] L. T. Heberlein, K. N. Levitt, and B. Mukherjee. A Method To Detect Intrusive Activity in a Networked Environment. In *Proceedings of the 14th National Computer Security Conference*, pages 362–371, October 1991.

[HLMS90] R. Heady, G. Luger, A. Maccabe, and M. Servilla. The Architecture of a Network Level Intrusion Detection System. Technical report, Department of Computer Science, University of New Mexico, August 1990.

[Hoa74] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Communications of the ACM*, 17(10):549–557, 1974.

[HP90] John L. Hennessy and David Patterson. *Computer Architecture – A Quantitative Approach.* Morgan Kaufman Publishers, Inc., San Mateo, California, 1990.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison Wesley, Reading, Massachusetts, 1979.

[Ilg92] Koral Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. Master's Thesis, Computer Science Department, University of California, Santa Barbara, July 1992.

[Jen92] Kurt Jensen. *Coloured Petri Nets – Basic Concepts I.* Springer Verlag, New York, 1992.

[Kni93] James Robert Knight. *Discrete Pattern Matching Over Sequences and Interval Sets.* Ph.D. Thesis, Department of Computer Science, University of Arizona, August 1993.

[Koz92] John Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection.* MIT Press, Cambridge, Massachusetts, 1992.

[KS] Sandeep Kumar and Eugene Spafford. A Taxonomy of Common Computer Security Vulnerabilities Based on their Method of Detection. (in preparation).

[KS94] Sandeep Kumar and Eugene Spafford. An Application of Pattern Matching in Intrusion Detection. Technical Report 94-013, Department of Computer Sciences, Purdue University, March 1994.

[KS95] Sandeep Kumar and Eugene H. Spafford. A Software Architecture to Support Misuse Intrusion Detection. Technical Report CSD–TR–95–009, Department of Computer Sciences, Purdue University, March 1995.

[Lam69] B. W. Lampson. Dynamic Protection Structures. In *Proceedings of the AFIPS Fall Joint Computer Conference*, pages 27–38, 1969.

[Lam71] B. W. Lampson. Protection. In *Proceedings of the Fifth Annual Princeton Conference on Information Science Systems*, pages 437–443, 1971. Reprinted in *Operating System Review*, Volume 8, Number 1 (January 1974), pages 18–24.

[Lan92] Linda Lankewicz. *A Non-Parametric Pattern Recognition to Anomaly Detection*. Ph.D. Thesis, Tulane University, Department of Computer Science, 1992.

[LBMC93] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws, with Examples. Technical Report NRL/FR/5542–93–9591, Naval Research Laboratory, Washington, DC 20375–5320, November 1993.

[Lin75] Richard R. Linde. Operating System Penetration. In *National Computer Conference*, pages 361–368, 1975.

[LJL+89] Teresa F. Lunt, R. Jagannathan, Rosanna Lee, Alan Whitehurst, and Sherry Listgarten. Knowledge based Intrusion Detection. In *Proceedings of the Annual AI Systems in Government Conference*, Washington, DC, March 1989.

[LS87] Dennis Longley and Michael Shain. *Data and Computer Security: Dictionary of Standards, Concepts, and Terms*. Stockton Press, New York, New York, 1987.

[LTG+92] T. F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P. G. Neumann, H. S. Javitz, A. Valdes, and T. D. Garvey. *A Real-Time Intrusion Detection Expert System (IDES) – Final Technical Report*. Computer Science Laboratory, SRI International, Menlo Park, California, February 1992.

[Lun93] Teresa F Lunt. A Survey of Intrusion Detection Techniques. *Computers & Security*, 12(4):405–418, June 1993.

[LV89] G. E. Liepins and H. S. Vaccaro. Anomaly Detection: Purpose and Framework. In *Proceedings of the 12th National Computer Security Conference*, pages 495–504, October 1989.

[MM89] Eugene W. Myers and Webb Miller. Approximate Matching of Regular Expressions. In *Bulletin of Mathematical Biology*, Volume 51, pages 5–37, 1989.

[MMA] Arthur B. Maccabe, Ruth McDonald, and Vinay Anand. Learning How to Characterize Normal Behavior in Local Area Networks.

[Moi] Abha Moitra. Real-Time Audit Log Viewer And Analyzer.

[oDS85] Department of Defense Standard. *Department of Defense Trusted Computer System Evaluation Criteria.* Number DOD 5200.28-STD. U.S. Government Printing Office, December 1985.

[Par90] Graham D. Parrington. Reliable Distributed Programming in C++: The Arjuna Approach. In *USENIX 1990 C++ Conference Proceedings*, pages 37–50, 1990.

[Pea88] Judea Pearl. *Probabilistic Reasoning in Expert Systems.* Morgan Kaufman, San Mateo, California, 1988.

[Pet92] Charles Petzold. *Programming Windows 3.1.* Microsoft Press, Redmond, Washington, 1992.

[PK92] Phillip A. Porras and Richard A. Kemmerer. Penetration State Transition Analysis – A Rule-Based Intrusion Detection Approach. In *Eighth Annual Computer Security Applications Conference*, pages 220–229. IEEE Computer Society press, IEEE Computer Society press, November 30 – December 4, 1992.

[Pow95] Richard Power. Current and Future Danger. Computer Security Institute, San Francisco, California, 1995.

[Pro94] Paul Proctor. Audit Reduction and Computer Misuse Detection. Talk given at the Sixth Annual Computer Security Incident Handling Workshop, 1994.

[Rag93] Stephen A. Rago. *UNIX System V Network Programming.* Addison-Wesley, Reading, Massachusetts, 1993.

[RS91] Deborah Russell and G. T. Gangemi Sr. *Computer Security Basics.* O'Reilly & Associates, Inc., Sebastopol, California, December 1991.

[SBD+91] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and an Early Prototype. In *Proceedings of the 14th National Computer Security Conference*, pages 167–176, October 1991.

[SG91] Shiuhpyng Winston Shieh and Virgil D. Gligor. A Pattern Oriented Intrusion Model and its Applications. In *Proceedings of the 1991 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 327–342, May 1991.

[SG94] Abraham Silberschatz and Peter B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, Massachusetts, Fourth edition, 1994.

[SH82] John F. Schoch and Jon A. Hupp. The "Worm" Programs — Early Experience with a Distributed Computation. *Communications of the ACM*, 25(3):172–180, March 1982.

[Sma88] Stephen E. Smaha. Haystack: An Intrusion Detection System. In *Fourth Aerospace Computer Security Applications Conference*, pages 37–44, Tracor Applied Science Inc., Austin, Texas, December 1988.

[Sma92] Steve Smaha. Questions about CMAD. In *Proceedings of the Workshop on Future Directions in Computer Misuse and Anomaly Detection*, pages 17–21, Davis, California, March 1992.

[Sma95] Steve Smaha. Talk given at the third Computer Misuse and Anomaly Detection Workshop (CMAD III) in Sonoma, California, January 1995.

[Spa89] Eugene Spafford. Crisis and Aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[SS92] Steven R. Snapp and Stephen E. Smaha. Signature Analysis Model Definition and Formalism. In *Proceedings of the Fourth Workshop on Computer Security Incident Handling*, Denver, Colorado, August 1992.

[SSH93] David R. Safford, Douglas L. Schales, and David K. Hess. The TAMU Security Package: An Outgoing Response to Internet Intruders in an Academic Environment. In *Proceedings of the Fourth USENIX Security Symposium*, pages 91–118, Santa Clara, California, 1993.

[SSHW88] M. Sebring, E. Shellhouse, M. Hanna, and R. Whitehurst. Expert Systems in Intrusion Detection: A Case Study. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

[Sto88] Clifford Stoll. Stalking the Wily Hacker. *Communications of the ACM*, 31(5):484–497, May 1988.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, Second edition, 1991.

[Sun93a] SunPro, Mountain View, California. *SPARCompiler C++ 4.0 Tools.h++ Class Library*, December 1993. Part No: 801-4317-10.

[Sun93b] SunSoft, Mountain View, California. *Solaris SHIELD Basic Security Module Revision A*, October 1993. Part No: 801-5285-10.

[TCL90] Henry S. Teng, Kaihu Chen, and Stephen C Lu. Security Audit Trail Analysis Using Inductively Generated Predictive Rules. In *Proceedings of the Sixth Conference on Artificial Intelligence Applications*, pages 24–29, Piscataway, New Jersey, March 1990. IEEE.

[Tho87] Ken Thompson. Reflections on Trusting Trust. *Communications of the ACM*, 1(3):21–31, July 1987.

[Wet93] Bradford R. Wetmore. Paradigms for the Reduction of Audit Trails. Master's Thesis, University of California, Davis, 1993.

[WF74] Robert A. Wagner and Michael J. Fischer. The String-to-String Correction Problem. In *Journal of the ACM*, Volume 21, pages 168–178, January 1974.

[Win92] Patrick Henry Winston. *Artificial Intelligence*. Addison Wesley, Reading, Massachusetts, Third edition, 1992.

[WM91] Sun Wu and Udi Manber. Fast Text Searching With Errors. Technical Report TR 91-11, Department of Computer Science, University of Arizona, 1991.

APPENDIX

APPENDIX
SOME EXAMPLE INTRUSION PATTERNS

Here we describe some signature patterns that we used while deriving performance results for the prototype implementation. These signatures are translated into C++ code that do the matching. We have included the translated C++ code for the first pattern. The translation of other patterns is similar and is omitted for brevity.

1. Representing Clarke Wilson monitoring triples [CW89]. The purpose of these triples is described in Section 3.2.1. Figure 3.3 is a pictorial representation of the signature.

```
pattern CW "Clarke Wilson Monitoring Triples" priority 10
  int PID, EUID; /* pattern local vars, may be initialized. */
  str PROG, FILE;
```

PROG is a token local variable that stores the program name corresponding to the process id PID, FILE stores the file name that PROG opens for writing. EUID stores the effective user id of PROG.

```
  state start, after_exec, violation;
  post_action {
   printf("CWilson violated for file %s, PID %d, EUID %d\n",
     FILE, PID, EUID);
  }
```

The post action is code that is executed when the pattern is successfully matched.

```
  neg invariant first_inv
    state start_inv, final;

    trans exit(EXIT)
      <- start_inv;
```

```
        -> final;
        |_ { PID = this[PID]; }
      end exit;
    end first_inv;
```

The invariant specifies the garbage collection of partial matches once the process has exited. What follows is the pattern description. The pattern matches all `EXECVE` records to monitor the creation of all processes in the system. Once a process is created, the pattern attempts to match all possible ways that the process could modify a file. These could be:

- Open a file to read and create it if it doesn't exist. This is handled in transition `mod1`.
- Open a file to read and truncate if it exists. Create the file if it doesn't exist. This is handled in transition `mod2`.
- ....and so on for all the other valid audit record types involving an `open` that might change the file.
- Delete a file. This is handled in transition `mod12`.

```
    trans exec(EXECVE) /* EXECVE is the event type */
      <- start;
      -> after_exec;
      |_ { this[ERR] = 0 && PID = this[PID] && PROG = this[PROG] &&
           EUID = this[EUID]; }
    end exec;

    trans mod1(OPEN_RC)
      <- after_exec;
      -> violation;
      |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
           disallowed(EUID, PROG, FILE); }
    end mod1;

    trans mod2(OPEN_RTC)
      <- after_exec;
      -> violation;
      |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
           disallowed(EUID, PROG, FILE); }
```

```
end mod2;

trans mod3(OPEN_RT)
  <- after_exec;
  -> violation;
  |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
       disallowed(EUID, PROG, FILE); }
end mod3;

trans mod4(OPEN_RW)
  <- after_exec;
  -> violation;
  |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
       disallowed(EUID, PROG, FILE); }
end mod4;

trans mod5(OPEN_RWC)
  <- after_exec;
  -> violation;
  |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
       disallowed(EUID, PROG, FILE); }
end mod5;

trans mod6(OPEN_RWTC)
  <- after_exec;
  -> violation;
  |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
       disallowed(EUID, PROG, FILE); }
end mod6;

trans mod7(OPEN_RWT)
  <- after_exec;
  -> violation;
  |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
       disallowed(EUID, PROG, FILE); }
end mod7;

trans mod8(OPEN_W)
  <- after_exec;
  -> violation;
  |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
       disallowed(EUID, PROG, FILE); }
end mod8;

trans mod9(OPEN_WC)
  <- after_exec;
```

```
      -> violation;
      |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
           disallowed(EUID, PROG, FILE); }
    end mod9;

    trans mod10(OPEN_WTC)
      <- after_exec;
      -> violation;
      |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
           disallowed(EUID, PROG, FILE); }
    end mod10;

    trans mod11(OPEN_WT)
      <- after_exec;
      -> violation;
      |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
           disallowed(EUID, PROG, FILE); }
    end mod11;

    trans mod12(UNLINK)
      <- after_exec;
      -> violation;
      |_ { this[ERR] = 0 && PID = this[PID] && FILE = this[OBJ] &&
           disallowed(EUID, PROG, FILE); }
    end mod12;
  end CW;
```

The translation of the signature into C++ code by the prototype results in the following:

```
#include <stream.h>   // -*- Mode: c++; truncate-lines: t; -*-

#include <assert.h>
#include <fstream.h>
#include <stdlib.h>
#include "utils.h"
#include "C2_Server.h"
#include <stdarg.h> // va_list, va_start...
```

The token local variables of the pattern become private data members of the class `C2_CW_Token`. The class name `C2_CW_Token` is a concatenation of three terms, `C2`, which signifies that the pattern matches against a C2 audit trail, `CW`, for the name of the pattern, and `Token` as a mnemonic. An application can use several instantiations of the library simultaneously such as an instantiation for IP datagrams and an instantiation for C2 audit trails. The naming scheme we have used allows the pattern name space to be local to the event stream the pattern is matched against. This allows the specification of the same pattern name for a pattern that matches against different event streams.

The enum `field_names` allows token local variables to be referred to symbolically.

```
class C2_CW_Token
{
   int PID;
   int EUID;
   Str PROG;
   Str FILE;
   enum field_names
   {                      /* enum values must start at 1 (?) */
      field_PID = 1,
      field_EUID = 2,
      field_PROG = 3,
      field_FILE = 4
   };
```

Because the semantics of token local variables distinguish between uninstantiated and instantiated variable states, a bit vector is maintained that stores this value for each token local variable. A value of 1 for the bit indicates that the variable is instantiated. The following mapping between `field_num` and the array index in `instantiated_field` helps to understand the expression used to index it. As `field_num` varies from `1..32,33..64` the index into the array `instantiated_field` varies as `0..0,1..1`. This suggests the indexing expression $(x \,\%\, 32 == 0)\ ?\ (x/32-1) : (x/32)$. The corresponding bit within the index is then $field\_num - 1 \,\%\, 32$.

```
   unsigned int instantiated[1]; // array size to store 4 fields
   int instantiated_field(int field_num)
   {
```

```
      assert(field_num <= 4);
      return instantiated[field_num % (sizeof(int) * 8) == 0 ?
                          field_num / (sizeof(int) * 8) - 1 :
                          field_num / (sizeof(int) * 8)] &
             (1 << (field_num - 1) % (sizeof(int) * 8));
   }

   void set_instantiated_field(int field_num)
   {
      assert(field_num <= 4);
      return instantiated[field_num % (sizeof(int) * 8) == 0 ?
                          field_num / (sizeof(int) * 8) - 1 :
                          field_num / (sizeof(int) * 8)] |=
             (1 << (field_num - 1) % (sizeof(int) * 8));
   }
 public:
```

Each token in the pattern is linked on two doubly-linked lists. One list groups all the tokens resident in a pattern state. This corresponds to the member data fields `next_state` and `prev_state`. The other links all the tokens derived from the root token that was duplicated from the start state. When the invariant is satisfied, all these tokens are destroyed. This is called the sibling list and corresponds to the member data fields `next_sib` and `prev_sib`. Both the lists are intrusive [Str91, Chapter 8].

```
   C2_CW_Token *next_sib, *prev_sib;

   /* doubly linked list of tokens resident in a state */
   C2_CW_Token *next_state, *prev_state;

   static C2_CW_Token *unify_toks(int num_toks,C2_CW_Token*...);

   C2_CW_Token()
   {
      int i;
      for (i = 0; i < 1; i++)
           instantiated[i] = 0;
        next_sib = prev_sib = NULL;
        next_state = prev_state = NULL;
   }
```

Assignment to token local variables involves determining if the variable has already been instantiated. If the variable has not already been instantiated, assignment is normal. If the variable has been instantiated, assignment is the same as equality. Assignment in our model is implemented as unification.

```
int assign_PID(int val)
{
   return instantiated_field(field_PID) ? (PID == val) :
           (PID = val, set_instantiated_field(field_PID), 1);
}

int get_PID()
{
   return PID;
}

int assign_EUID(int val)
{
   return instantiated_field(field_EUID) ?
          (EUID == val) :
          (EUID = val, set_instantiated_field(field_EUID), 1);
}

int get_EUID()
{
   return EUID;
}

int assign_PROG(Str val)
{
   return instantiated_field(field_PROG) ?
      (strcmp(PROG, val) == 0) :
      (PROG = crcp(val),
         set_instantiated_field(field_PROG), 1);
}

Str get_PROG()
{
   return PROG;
}

int assign_FILE(Str val)
{
   return instantiated_field(field_FILE) ?
      (strcmp(FILE, val) == 0) :
```

```
      (FILE = crcp(val),
         set_instantiated_field(field_FILE), 1);
}

Str get_FILE()
{
   return FILE;
}
```

When a token is duplicated, it is not placed in any list by default. That is, it is placed neither in any state, nor is it associated with any root token.

```
C2_CW_Token *dup()
{
     C2_CW_Token *t = new C2_CW_Token;
     assert(t != NULL);
     *t = *this;
     t->next_sib = t->prev_sib = NULL;
     t->next_state = t->prev_state = NULL;
     return t;
}
```

This function duplicates a token *tok* and puts the duplicate in the same sibling queue as *tok*.

```
C2_CW_Token *dup_n_link_after(C2_CW_Token * tok)
{
     C2_CW_Token *t = dup();
     t->next_sib = tok->next_sib;
     t->prev_sib = tok;
     tok->next_sib = t;
     t->next_state = t->prev_state = NULL;
     if (t->next_sib != NULL)
        t->next_sib->prev_sib = t;
     return t;
}

~C2_CW_Token()
{
}
```

This function deletes a token from its state list. Each token knows which state it belongs to (not strictly true) and can therefore delete itself from that list.

```
int delete_from_state_list()
{
   /* there's always a dummy element in the list, therefore
      there's always an element before this one in the
      linked list */
   prev_state->next_state = next_state;
   if (next_state != NULL)
      next_state->prev_state = prev_state;

   return 1;
}
```

This deletes a token from its sibling list.

```
int delete_from_sibling_list()
{
   /* this is a pure doubly linked list, therefore there's not
      always an element before or after this one in the
      linked list */
   if (prev_sib != NULL)
      prev_sib->next_sib = next_sib;
   if (next_sib != NULL)
      next_sib->prev_sib = prev_sib;

   return 1;
}

int del()
{
   delete_from_state_list();
   delete_from_sibling_list();
   delete this;

   return 1;
}

int delete_all_siblings()
{
   /* as above + walk down the sibling chain & delete every
      token. These tokens might also be in state lists from
      which they must be removed before being deleted */
   C2_CW_Token *curr, *prev;

   /* go to one end of the linked list */
   for (prev = this; prev->next_sib != NULL;
                           prev = prev->next_sib);
```

```
   for (curr = prev->prev_sib; curr != NULL;
               prev = curr, curr = curr->prev_sib)
   {
      prev->delete_from_state_list();
      delete prev;
   }

   prev->delete_from_state_list();
   delete prev;

   return 1;
}

int dbg()
{
   cerr << "PID = ";
   if (instantiated_field(field_PID))
      cerr << PID;
   else
      cerr << "(unknown)";
   cerr << ".";

   cerr << "EUID = ";
   if (instantiated_field(field_EUID))
      cerr << EUID;
   else
      cerr << "(unknown)";
   cerr << ".";

   cerr << "PROG = ";
   if (instantiated_field(field_PROG))
      cerr << PROG;
   else
      cerr << "(unknown)";
   cerr << ".";

   cerr << "FILE = ";
   if (instantiated_field(field_FILE))
      cerr << FILE;
   else
      cerr << "(unknown)";
   cerr << ".";

   cerr << endl;
   return 1;
}
```

```

};
```

A state contains two linked lists, one to store the tokens currently present in it (`toktab`), the other, (`incoming_toks`), to store the tokens that will enter the state when the pattern is clocked. The simulation of the pattern takes place in discrete steps. At each step, all the enabled transitions are tested against the incoming event to determine if some tokens satisfy the transition guard and need to be duplicated and moved to successor states of the transition. Then all such successful tokens across all the transitions are moved into successor states together. This moving is done by function `clock()` for each state.

```
class C2_CW_State : public State
{
 public:
   PDL_Ilist <C2_CW_Token> toktab; /* incorporate token replace-
                                      ment policy here */
   PDL_Ilist <C2_CW_Token> incoming_toks;
   ~C2_CW_State()
   {
   }

   int clock()
   {
      if (incoming_toks.empty())
         return 1;

      /* Move all the tokens in incoming_toks to toktab
         in one fell swoop */
      toktab.push_chain(incoming_toks.chain());
      incoming_toks.reset();
      return 1;
   }

   int dbg()
   {
      C2_CW_Token *t;
        cerr << "Tokens in state are:\n";
      for(t = toktab.first(); t != NULL; t = t->next_state)
         t->dbg();
      cerr << "\nTokens awaiting entry into the state are:\n";
      for (t = incoming_toks.first(); t != NULL;
```

```
                                     t = t->next_state) t->dbg();
        return 1;
    }

};
```

This class implements the structure of the pattern. The pattern has five states, including invariant states: `start`, `after_exec`, `violation`, `start_inv` and `final`. The data member `serv` points to the server which dispatches events to it. When the pattern is created, it requests the server to queue it on each event queue that is the label of some transition in the pattern. This tells the server to dispatch those events to the pattern. The linking on the server queues is done by means of intrusive lists, i.e. the links are provided by the pattern. This pattern requests for the events `EXECVE`, `OPEN_RC`, `OPEN_RTC`, `OPEN_RT`, `OPEN_RW`, `OPEN_RWC`, `OPEN_RWTC`, `OPEN_RWT`, `OPEN_W`, `OPEN_WC`, `OPEN_WTC`, `OPEN_WT`, `UNLINK`, `EXIT`.

```
class C2_CW : public C2_Pattern
{
    C2_CW_State start+, after_exec, violation, start_inv, final;

    C2_Pattern *next_EXECVE, *next_OPEN_RC, *next_OPEN_RTC,
               *next_OPEN_RT, *next_OPEN_RW, *next_OPEN_RWC,
               *next_OPEN_RWTC, *next_OPEN_RWT, *next_OPEN_W,
               *next_OPEN_WC, *next_OPEN_WTC, *next_OPEN_WT,
               *next_UNLINK, *next_EXIT;
    C2_Server *serv;

 public:
    void print_dbg()
    {
        cerr << "I am in pattern C2_CW" << endl;
        cerr << "State start:" << endl;
        start.dbg();
        cerr << "State after_exec:" << endl;
        after_exec.dbg();
        cerr << "State violation:" << endl;
        violation.dbg();
        cerr << "State start_inv:" << endl;
        start_inv.dbg();
```

```
        cerr << "State final:" << endl;
        final.dbg();

    }

    void PatProc(Event * e);
    void restart(void);
    int num_toks();
    char *name()
    {
        return "CW";
    }

    C2_CW(C2_Server * S)
    {
        serv = S;
        start.toktab.push(new C2_CW_Token);
        S->thread_on_events(this, 14, C2event_EXECVE,
            C2event_OPEN_RC, C2event_OPEN_RTC, C2event_OPEN_RT,
            C2event_OPEN_RW, C2event_OPEN_RWC, C2event_OPEN_RWTC,
            C2event_OPEN_RWT, C2event_OPEN_W, C2event_OPEN_WC,
            C2event_OPEN_WTC, C2event_OPEN_WT, C2event_UNLINK,
                                                   C2event_EXIT);
    }
};
```

This function takes an arbitrary number of tokens and unifies them to get a new "unified" token. Let the token have m variables $v_1 \ldots v_m$. By unification (denoted here by $\cap$) of tokens $t_1 \ldots t_n$ we mean that $\forall v_i$, if any v_i has been instantiated to a value, then all t_is must have the same value for v_i. The unified token has that value for v_i.

```
C2_CW_Token *C2_CW_Token::unify_toks(int num_toks,
                                     C2_CW_Token *tok1...)
{
    typedef C2_CW_Token *C2_CW_TokenP;
    static C2_CW_Token **tokarr = new C2_CW_TokenP[9];
    static int tokarr_sz = 9;
    int i, j;

    if (num_toks > tokarr_sz)
    {
        /* resize the static array */
```

```
      C2_CW_Token **t = new C2_CW_TokenP[num_toks];
      tokarr_sz = num_toks;
      delete[] tokarr;
      tokarr = t;
   }

   // extract the varargs into tokarr
   va_list ap;
   va_start(ap, tok1);
   for (i = 0; i < num_toks; i++)
   {
      if (i == 0)
         tokarr[i] = tok1;
      else
         tokarr[i] = va_arg(ap, C2_CW_TokenP);
   }
   va_end(ap);

   C2_CW_Token *newtok = new C2_CW_Token;
   // try to unify the token local var symtab[i]->symname()
   for (i = 0; i < num_toks; i++)
   {
      /*
       * find the first token with the instantiated local var
       * symtab[i]->symname()
       */
      if (tokarr[i]->instantiated_field(field_PID))
         break;
   }
   if (i < num_toks)
   {
      for (j = i + 1; j < num_toks; j++)
         if (tokarr[j]->instantiated_field(field_PID) &&
                              tokarr[i]->PID != tokarr[j]->PID)
         {
            delete newtok;
            return 0;
         }
   }
   else
   {
      /* this field is already marked uninstantiated by the
         class constructor. It unifies successfully across all
         tokens */
   }

```

```
// try to unify the token local var symtab[i]->symname()
for (i = 0; i < num_toks; i++)
{
   /*
    * find the first token with the instantiated local var
    * symtab[i]->symname()
    */
   if (tokarr[i]->instantiated_field(field_EUID))
      break;
}
if (i < num_toks)
{
   for (j = i + 1; j < num_toks; j++)
      if (tokarr[j]->instantiated_field(field_EUID) &&
                         tokarr[i]->EUID != tokarr[j]->EUID)
      {
         delete newtok;
         return 0;
      }
}
else
{
   /* this field is already marked uninstantiated by the
      class constructor. It unifies successfully across all
      tokens */
}

// try to unify the token local var symtab[i]->symname()
for (i = 0; i < num_toks; i++)
{
   /*
    * find the first token with the instantiated local var
    * symtab[i]->symname()
    */
   if (tokarr[i]->instantiated_field(field_PROG))
      break;
}
if (i < num_toks)
{
   for (j = i + 1; j < num_toks; j++)
      if (tokarr[j]->instantiated_field(field_PROG) &&
                         tokarr[i]->PROG != tokarr[j]->PROG)
      {
         delete newtok;
         return 0;
      }
```

```
    }
    else
    {
       /* this field is already marked uninstantiated by the
          class constructor. It unifies successfully across all
          tokens */
    }

    // try to unify the token local var symtab[i]->symname()
    for (i = 0; i < num_toks; i++)
    {
       /*
        * find the first token with the instantiated local var
        * symtab[i]->symname()
        */
       if (tokarr[i]->instantiated_field(field_FILE))
          break;
    }
    if (i < num_toks)
    {
       for (j = i + 1; j < num_toks; j++)
          if (tokarr[j]->instantiated_field(field_FILE) &&
                            tokarr[i]->FILE != tokarr[j]->FILE)
          {
             delete newtok;
             return 0;
          }
    }
    else
    {
       /* this field is already marked uninstantiated by the
          class constructor. It unifies successfully across all
          tokens */
    }

    return newtok;
 }

 void C2_CW::PatProc(Event * e)
 {
    int i, j, succ;

    C2_CW_Token *unified_tok, *unified_evaled_tok, *tok;
```

This table contains pointers to all the tokens that reside in *nodup* states and that successfully participated in a transition firing. These tokens must be destroyed before exiting the function.

```
static PTable <C2_CW_Token *> toks_in_nodup_states;
C2_CW_Token *i0, *i1, *i2, *i3, *i4, *i5, *i6, *i7,
            *i8, *i9, *i10;
switch (e->type())
{
```

All the transitions in the pattern of type `EXECVE` will be exercised in this switch case. The pattern has only one such transition, named `exec`. There's a pointer downcast from `e`, of type Event, to `eve` of type C2event_EXECVE.

```
 case C2event_EXECVE:
   {
      C2Event_EXECVE *eve = (C2Event_EXECVE *) e;

```

The input state of transition `exec` is `start`. The transition has only one input state.

```
      // Transition exec
      for(i0 = start.toktab.first(); i0 != NULL;
                                      i0 = i0->next_state)
      {
         unified_tok = i0->dup();

         /* eval this guard for this token and event */
         succ = ((((eve->ERR() == 0) &&
                   unified_tok->assign_PID(eve->PID())) &&
                  unified_tok->assign_PROG(eve->PROG())) &&
                 unified_tok->assign_EUID(eve->EUID()));

         if (!succ)
         {
            delete unified_tok;
            continue;
         }
         else
            unified_evaled_tok = unified_tok;

```

```
        /* put a copy of a succ token in every out state of
           this transition */
        after_exec.incoming_toks.push(
           unified_evaled_tok->dup_n_link_after(
                                 unified_evaled_tok));
```

Because this transition indicates the beginning of a match, duplicates of its successful token are also placed in the start state of each invariant.

```
        /* this transition has one of its inputs from a
           start state */
        start_inv.incoming_toks.push(
           unified_evaled_tok->dup_n_link_after(
                                 unified_evaled_tok));
        unified_evaled_tok->delete_from_sibling_list();
        delete unified_evaled_tok;

     }

     toks_in_nodup_states.del_objs_uniquely();
     after_exec.clock();
     start_inv.clock();
     while ((i0 = final.toktab.first()) != NULL)
     {
        i0->delete_all_siblings();
     }
  }
  break;
```

Similarly for all transitions labeled with the event OPEN_RC.

```
case C2event_OPEN_RC:
  {
     C2Event_OPEN_RC *eve = (C2Event_OPEN_RC *) e;

     // Transition mod1
     for (i0 = after_exec.toktab.first(); i0 != NULL;
                                     i0 = i0->next_state)
     {
        unified_tok = i0->dup();

        /* eval this guard for this token and event */
        succ = ((((eve->ERR() == 0) &&
                   unified_tok->assign_PID(eve->PID()))  &&
```

```
                   unified_tok->assign_FILE(eve->OBJ())) &&
                   disallowed(unified_tok->get_EUID(),
                              unified_tok->get_PROG(),
                              unified_tok->get_FILE()));

        if (!succ)
        {
           delete unified_tok;
           continue;
        }
        else
           unified_evaled_tok = unified_tok;

        /* put a copy of a succ token in every out state
           of this transition */
        violation.incoming_toks.push(
           unified_evaled_tok->dup_n_link_after(i0));

        delete unified_evaled_tok;
     }

     toks_in_nodup_states.del_objs_uniquely();
     violation.clock();
     while ((i0 = final.toktab.first()) != NULL)
     {
        i0->delete_all_siblings();
     }
  }
  break;
```

And so on for the other transitions. We skip them for brevity.

```
case C2event_OPEN_RTC:
  {
    <similarly>
  }
  break;

case C2event_OPEN_RT:
  {
    <similarly>
  }
  break;

case C2event_OPEN_RW:
  {
```

```
        <similarly>
      }
      break;

    case C2event_OPEN_RWC:
      {
        <similarly>
      }
      break;

    case C2event_OPEN_RWTC:
      {
        <similarly>
      }
      break;

    case C2event_OPEN_RWT:
      {
        <similarly>
      }
      break;

    case C2event_OPEN_W:
      {
        <similarly>
      }
      break;

    case C2event_OPEN_WC:
      {
        <similarly>
      }
      break;

    case C2event_OPEN_WTC:
      {
        <similarly>
      }
      break;

    case C2event_OPEN_WT:
      {
        <similarly>
      }
      break;

```

```
    case C2event_UNLINK:
      {
        <similarly>
      }
      break;
```

This transition is part of the pattern invariant. Its code is similar to that of the other transitions except that tokens that are placed in the output place of this transition end up in the invariant final state. A token reaching the invariant final state signifies the successful matching of the invariant. This means that for that particular token, all its "siblings" must be destroyed. The siblings of this token is the equivalence class of all the tokens that descended (result of duplicating a token, duplicating its duplicate and so on) from the root token.

```
    case C2event_EXIT:
      {
         C2Event_EXIT *eve = (C2Event_EXIT *) e;

         // Transition exit
         for (i0 = start_inv.toktab.first(); i0 != NULL;
                                             i0 = i0->next_state)
         {
            unified_tok = i0->dup();

            /* eval this guard for this token and event */
            succ = unified_tok->assign_PID(eve->PID());

            if (!succ)
            {
               delete unified_tok;
               continue;
            }
            else
               unified_evaled_tok = unified_tok;

            /* put a copy of a succ token in every out state
               of this transition */
            final.incoming_toks.push(
               unified_evaled_tok->dup_n_link_after(i0));

            delete unified_evaled_tok;
         }
```

```

            toks_in_nodup_states.del_objs_uniquely();
            final.clock();
```

The while loop implements what was described in the previous paragraph.

```
            while ((i0 = final.toktab.first()) != NULL)
            {
               i0->delete_all_siblings();
            }
         }
         break;

      }                            /* end switch */
```

This is the post action. At the end of every simulation step, the final state is checked for tokens. This signals a pattern match. For each such token, the post action is executed.

```
      // Post Action
      for (i0 = violation.toktab.first(); i0 != NULL;
                                          i0 = i0->next_state)
      {
         unified_tok = i0;
         printf("CWilson violated for file %s, PID %d, EUID %d\n",
                unified_tok->get_FILE(), unified_tok->get_PID(),
                unified_tok->get_EUID());

         // destroy the token and rm from lists
         i0->del();
      }
   }
```

We define the routine `create_pattern` to be of external C linkage. This provides a fixed function name to call to create a pattern. When a pattern needs to be created from the application program, the the event server, for example, `C2_Server`, is given a file name that contains the pattern description. The server then parses the pattern description into C++ code (all this code), compiles the C++ code, and dynamically links it into the application. At this point the server creates a pattern object by looking for the function symbol `create_pattern` in the shared library just linked, and calls it. If `create_pattern` was not defined to be of external C linkage, its name would be mangled and it would be more tedious to create a new pattern object.

```
extern "C"
{
   C2_CW *create_pattern(C2_Server * S);
}

C2_CW *create_pattern(C2_Server * S)
{
   cerr << "Inside create pattern." << endl;
   return new C2_CW(S);
}
```

2. Privileged programs may not be permitted to follow symbolic links on opening files for reading/writing. This signature may indirectly detect the following exploitations:

   - `lpr` is made to dump a privileged file to the printer through a symlink.
   - `/bin/mail` and other programs are fooled into writing to arbitrary places because the name of the temporary file they create for internal use can be guessed. A link with this temporary name can be created that points to strange places.

   This pattern is very similar in structure to the previous one. The function `islink` tests to see if a pathname is a link. In this definition, a pathname is a link if it is a symbolic link or if it is a regular file and the link count of that file inode is > 1.

```
pattern Dont_Follow_Symlinks "" priority 7
  state start, after_open;
  int PID;
  str FILE;
  post_action
  {
    printf("Privileged process %d opened link %s.\n", PID, FILE);
  }
```

For any process with an effective uid of 0, flag all successful `opens` to pathnames that are links. A successful operation is denoted by the condition `this[ERR] = 0`. Each of the transitions below is a different way of opening a file. The difference between them is the set of arguments passed to the open system call.

```
trans open5(OPEN_R)
  <- start;
  -> after_open;
  |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
       FILE = this[OBJ] && PID = this[PID]; }
end open5;

trans open1(OPEN_RC)
  <- start;
  -> after_open;
  |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
       FILE = this[OBJ] && PID = this[PID]; }
end open1;

trans open3(OPEN_RT)
  <- start;
  -> after_open;
  |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
       FILE = this[OBJ] && PID = this[PID]; }
end open3;

trans open2(OPEN_RTC)
  <- start;
  -> after_open;
  |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
       FILE = this[OBJ] && PID = this[PID]; }
end open2;

trans open4(OPEN_RW)
  <- start;
  -> after_open;
  |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
       FILE = this[OBJ] && PID = this[PID]; }
end open4;

trans open6(OPEN_RWC)
  <- start;
  -> after_open;
```

```
    |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
         FILE = this[OBJ] && PID = this[PID]; }
  end open6;

  trans open7(OPEN_RWT)
    <- start;
    -> after_open;
    |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
         FILE = this[OBJ] && PID = this[PID]; }
  end open7;

  trans open8(OPEN_RWTC)
    <- start;
    -> after_open;
    |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
         FILE = this[OBJ] && PID = this[PID]; }
  end open8;

  trans open9(OPEN_W)
    <- start;
    -> after_open;
    |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
         FILE = this[OBJ] && PID = this[PID]; }
  end open9;

  trans open10(OPEN_WC)
    <- start;
    -> after_open;
    |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
         FILE = this[OBJ] && PID = this[PID]; }
  end open10;

  trans open11(OPEN_WT)
    <- start;
    -> after_open;
    |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
         FILE = this[OBJ] && PID = this[PID]; }
  end open11;

  trans open12(OPEN_WTC)
    <- start;
    -> after_open;
    |_ { this[ERR] = 0 && this[EUID] = 0 && islink(this[OBJ]) &&
         FILE = this[OBJ] && PID = this[PID]; }
  end open12;
```

```
end Dont_Follow_Symlinks;
```

3. Executing a link to a setuid shell script through a link that appears to the shell as an argument.

   `Basename` is an external function defined in the application that returns the file part of a pathname.

```
extern str Basename(str);
pattern Shell_Script_Attack "ln setid_script -x; -x" priority 7
  state start, after_exec;
  int RUID;
  str PROG;

  post_action {
    printf("User id %d has executed a wierd shell script(%s).\n",
     RUID, PROG);
  }
```

   The signature monitors all successful (`this[ERR] = 0`) execs of a pathname (`PROG`) whose name begins with a '`-`' (i.e. it matches the regular expression `^-`. This is the condition `Basename(this[PROG]) =~ "^-"`.) and which is a link to a shell script (the first two characters of the file are #! and the file is executable by one of user, group or other).

```
  trans exec(EXECVE)
    <- start;
    -> after_exec;
    |_ {
         this[ERR] = 0 && RUID = this[RUID] && PROG = this[PROG]
         && islink(this[PROG]) && shell_script(this[PROG]) &&
         (Basename(this[PROG]) =~ "^-");
       }
  end exec;
end Shell_Script_Attack;
```

VITA

VITA

Sandeep Kumar was born in India in 1963. He completed his Bachelor of Technology in electrical engineering from the Indian Institute of Technology, New Delhi in 1985. In the fall of 1985, he entered the University of Tennessee, Knoxville and received an M.S. in computer science in 1987. Before attending Purdue in 1990, he worked as a consultant in New Jersey. His research interests include computer security, intrusion detection, operating systems, and networking.