An Application of Pattern Matching in Intrusion Detection*

Technical Report CSD–TR–94–013

Sandeep Kumar Eugene H. Spafford

The *COAST* Project
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
{kumar,spaf}@cs.purdue.edu

June 17, 1994

Abstract

This report examines and classifies the characteristics of signatures used in misuse intrusion detection. Efficient algorithms to match patterns in some of these classes are described. A generalized model for matching intrusion signatures based on Colored Petri Nets is presented, and some of its properties are derived.

1 Introduction

Computer break-ins and their misuse have become common features [Met87, Bos88, Sto88, Mar88, Rei87, SSH93]. The number, as well as sophistication, of attacks on computer systems is on the rise. Often, network intruders have easily overcome the password authentication mechanism designed to protect the system. With an increased understanding of how systems work, intruders have become skilled at determining their weaknesses and exploiting them to obtain unauthorized privileges. Intruders also use patterns of intrusion that are often difficult to trace and identify. They use several levels of indirection before breaking into target systems and rarely indulge in sudden bursts of suspicious or anomalous activity, for example in [Sto88]. If an account on a target system is compromised, intruders may carefully cover their tracks so as not to arouse suspicion, as in [Spa89]. Furthermore, threats like viruses and worms 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 is almost impossible to trace their origin or the extent of infection.

Tools are therefore necessary to monitor systems, to detect break-ins, and to respond *actively* to the attacks in real time. Most break-ins prevalent today exploit well known security holes in system software. One solution to these problems is to study the characteristics of intrusions and from these, to extrapolate intrusion characteristics of the future, devise means of representing intrusions in a computer so that the

*This work was funded by the Division of INFOSEC Computer Science, Department of Defense.

break-ins can be detected in real time. This report is an analysis of a computational framework designed for matching such signatures efficiently.

Intrusions as such are difficult to define in terms of behavior or actions, but are more easily understood, and thus defined, in terms of their effect on computer systems. Such a definition is given in section 2 along with a brief description of the categories of intrusion detection, and a justification of the need for intrusion detection systems [sec 2.2]. Several approaches to intrusion detection are known. These include statistical methods which monitor anomalies in the system [sec 3.1] and pattern matching/expert system techniques that look for well defined patterns of attack [sec 3.2]. Recent techniques have proposed inductive generalization as a means for abstracting patterns of behavior [sec 3.8]; neural networks for predicting future behavior from past behavior [sec 3.8], in which the prediction failure is indicative of anomalous activity; and model based techniques in which the system is continually updating a model to match the observed behavior [sec 3.10].

There are many intrusions that cannot be detected by statistical methods alone and must be specifically watched for. This technique of monitoring specific patterns of attack, or misuse intrusion detection, is difficult to implement in a practical system for a variety of reasons [sec 4]. To design a generic misuse intrusion detection system, its desirable characteristics should be defined at the outset. We propose such a set that includes `generality, portability, scalability` and `real time performance` [sec 4]. Because the system watches for well defined intrusion signatures we have studied the characteristics of existing signatures and classified them in terms of five basic characteristics, namely `linearity, unification, occurrence, beginning` and `duration` [sec 4.2]. A matching model is proposed thereafter that can be part of a generic misuse detector. This model is based on Colored Petri Nets and lends itself well to a graphical representation [sec 4.4]; the details and some of its properties are given in sec 5. Some forms of matching in this model are shown to be as easy as matching fixed strings and regular expressions, but matching with unification is shown to be hard. Some possible optimizations in this model of matching are outlined.

The proposed model of matching is compared with two of the desirable characteristics outlined earlier. The issue of scalability, or matching several patterns efficiently is discussed in sec 5.2. The applicability of compiler optimization techniques to speed up matching and the virtual machine which forms its basis are also described. A method of achieving portability of intrusion signatures across different audit trails is outlined in sec 5.3. The realization of other characteristics can be better determined experimentally and is not discussed in this report.

2 Intrusion Detection

An intrusion is defined [HLMS90] as

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

It is a violation of the security policy of the system. Any definition of an intrusion is, of necessity, imprecise, as security policy requirements do not always translate into a well defined set of actions. Intrusion Detection is the methodology by which intrusions are detected.

This methodology 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 `A` 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. A user `B` might always login outside of working hours through the company terminal server. A late night rlogin session from another host to his account might be considered unusual. This technique of detecting intrusions attempts to quantify the good or acceptable behavior and flags other irregular behavior as intrusive.

In contrast, the second, 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 [Spa88] would come under this category. This technique represents knowledge about the 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 was based on its method of detection. Another classification scheme, based on the intrusion types, presented in [Sma88] breaks intrusions into the following six types:

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

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

Penetration of the security control system.

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

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

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

2.1 Premise of Intrusion Detection Schemes

A main 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 legitimate 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 in itself, anomalous. Ideally, the set of anomalous activity coincides with that of intrusive activity resulting in a lack of false positives. However, intrusive behavior does not always coincide with anomalous behavior. There are four possibilities, each one with a non zero probability:

1. *intrusive* but *not anomalous*
2. *not intrusive* but *anomalous*
3. *not intrusive* and *not anomalous*
4. *intrusive* and *anomalous*

For a probabilistic basis of intrusion detection see [LV89, LV92].

Most intrusion detection systems built to date [BK88, Sma88, LJL+89, SSHW88, LV89, LJL+89] etc. use the audit trail generated by a C2[1] or higher rated computer, for input. Others, for example [HLMS90, HLM91] analyze intrusions by the network connections and the flow of information in a network. Input information appropriate to the domain of intrusive activity is needed to detect intrusions. Enough domain information is required to determine those variables that accurately reflect intrusive behavior.

2.2 Why are Intrusion Detection Systems Necessary?

Computer systems are being broken into regularly. Are these intrusions a result of their generic weaknesses or a result of accidental discovery and exploitation of flaws, resulting perhaps, from their complexity? In other words, does the protection model of these systems theoretically provide adequate security against intrusions? Will the use of validated software for the critical components then thwart all attempts to intrude?

Most computer systems provide an access control mechanism as their first line of defense. However, this only defines 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, chapter 5]. Access control does not model and thus cannot prevent illegal information flow through the system, because such flow can take place with legal accesses to the objects.

Information flow can be controlled to provide more security; for example, the Bell LaPadula model [BL73], to provide secrecy, or the Biba model [Bib77], to provide integrity. However, there is a tradeoff between security and convenience of use. Both models are conservative and restrict read and write operations to ensure that the secrecy or the integrity of the system can never be compromised. Consequently, if both models are jointly used, the resulting system will flag almost any useful operation as a breach of some security condition. Thus, a very secure system may not be useful.

Furthermore, access controls and protection models do not help in the case of insider threats or compromise of the authentication module. If a weak password is broken, access control measures can do little to prevent stealing or corruption of information legally accessible to the compromised user. In general static methods of assuring properties in a system are overly restrictive and simply insufficient in some cases. Dynamic methods, for example behavior tracking, are therefore needed to detect and perhaps prevent breaches in security.

3 Approaches to Detecting Intrusions

Most systems built to date do both anomaly and misuse intrusion detection. Some are based on advanced techniques of predicting future patterns based on input seen thus far (like a reactive keyboard) and some intrusion detection systems have been built using neural nets [sec. 3.8]. Some approaches, for example the statistical approach, have resulted in systems that have been used and tested extensively, while others are still in the research stage. Proposed methods for future intrusion detection systems include the model based approach [sec. 3.10], to be included in NIDES [JLA+93]. Promising areas of application for future systems are discussed, which include Bayesian clustering [sec. 3.7]. No intrusion detection system using this approach has been designed to date. The merits, as well as shortcomings, of current intrusion detection systems are also discussed.

[1] A DoD security classification requiring auditing and unavailability of encrypted passwords.

3.1 Statistical Approaches to Intrusion Detection (Anomaly Intrusion Detection)

The following, based on [LTG+92], serves to illustrate the generic process of anomaly detection, which is by and large, statistical in nature. The anomaly detector observes the activity of subjects and generates profiles for them that captures their behavior. These profiles do not require much space for storage and are efficient to update. They are updated regularly, with the older data appropriately aged. As input audit records are processed, the system periodically generates a value indicative of its abnormality. This value is a function of the abnormality values of the individual measures comprising the user profile. For example, 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

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

In general, the measures $M_1, M_2, \ldots, M_n$ may not be mutually independent, resulting in a more complex function for combining them.

There are several types of measures comprising a profile, for example

1. `Activity Intensity Measures` — These measure the rate at which activity is progressing. They are usually 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 1 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, I/O activity etc. over the entire system usage, for a particular user.

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

4. `Ordinal Measures` — These measure activity whose outcome is a numeric value, for example the amount of CPU and I/O used by a particular user.

The current user behavior is stored in a current profile. At regular intervals the current profile is atomically merged with the stored profile[2] . Anomalous behavior is determined by comparing the current profile with the stored profile.

3.1.1 Pros and Cons of Statistical ID

An advantage of anomaly intrusion detection is that statistical techniques have applicability here. For example, data points that lie beyond a factor of the standard deviation on either side of the average might be considered anomalous. Or the integral of the absolute difference of two functions over time might be an indicator of the deviation of one function over the other.

Statistical intrusion detection systems also have several disadvantages. Even if statistical measures could be defined to capture the computer usage patterns unique to every user, by their very nature, these measures

[2] This is true of [LTG+92], in some systems the profiles are static and do not change once determined.

are insensitive to the order of the occurrence of events. That is, they miss the sequential interrelationships between events. For intrusions reflected by such an ordering of patterns, a purely statistical intrusion detection system will miss these intrusions.

Purely statistical intrusion detection systems have the disadvantage that their statistical measures capturing user behavior can be trained gradually to a point where intrusive behavior is considered normal. Intruders who know that they are being so monitored can train such systems over a length of time. Thus most existing intrusion detection schemes combine both a statistical part to measure aberration of behavior, and a misuse part that watches for the occurrence of prespecified patterns of events.

It is also difficult to determine the right threshold above which an anomaly is to be considered intrusive. And, to apply statistical techniques to the formulation of anomalies, one has to assume that the underlying data comes from a quasi stationary process, an assumption that may not always hold.

3.2 Misuse Intrusion Detection

This 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 sufficient to detect all types of intrusions. Purely statistical methods cannot prevent users from altering their profiles gradually over a period of time to a point where activity previously considered anomalous is regarded normal. Furthermore, no step in an intrusion may, in itself, be anomalous but the aggregate may be.

Intrusion signatures are usually specified as a sequence of events and conditions that lead to a break-in. The conditions define the context within which the sequence becomes an intrusion. 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 simple enough to keep the matching computationally tractable. Often, these patterns are simple enough for the detection process to be automated.

The primary technique of misuse intrusion detection uses an expert system. This paper proposes another technique, that of using pattern matching as a viable option for misuse intrusion detection [sec 4]. An example of the former is [SS92] which 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. The primary disadvantage of using expert systems is that working memory elements (fact base) that match the left sides of productions to determine eligible rules for firing are essentially sequence-less. It is thus hard to specify a natural ordering of facts efficiently within the natural framework of expert system shells[3] .

No system directly using pattern matching has been reported to date. The objective is to frame the intrusion detection problem as a pattern matching one, and devise efficient algorithms for such matching. This also has the benefit of a leaner, more efficient solution than that of using expert systems, which were originally meant to solve problems in a different domain. To frame the matching problem, it is easier to first classify patterns in an increasing order of matching complexity (for example in [sec 4.3]) and ensure

[3]Even though facts are numbered consecutively in current expert system shells, introducing fact numbering constraints within rules to enforce an order makes the Rete match [For82] procedure very inefficient.

that most of them fall in a set of classes for which efficient algorithms are known or can be devised. As the Rete match procedure [For82] used in expert system shells can be incorporated in the solution, it can be no worse than using expert systems to detect misuse intrusions.

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 an intruded 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 often 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 [SG91].

The primary disadvantage of this approach is that it looks only for *known* vulnerabilities, and is of little use in detecting unknown future intrusions.

3.3 Feature Selection *or* Which Measures are Good for Detecting Intrusions?

Given a set of possible measures, chosen heuristically, that can have a bearing on detecting intrusions, how does one determine the subset that accurately predicts or classifies intrusions? The exercise of determining the right measures is complicated by the fact that the appropriate subset of measures depends on the types of intrusions being detected. One set of measures will likely not be adequate for all types of intrusions. Predefined notions of the relevance of particular measures to detecting 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 100 measures as potentially relevant to predicting intrusions. This results in 2^{100} possible subsets of measures, of which there are presumably only a few subsets that result in a high predictability of intrusions. It is clearly untractable to search through this large space exhaustively. [HLMS90] present a genetic approach to searching through this space for the right subset of metrics. Using a learning classifier scheme they generate an initial set of measures which is refined in the *rule evaluation* mode using genetic operators of crossover and mutation. Subsets of the measures under consideration having low predictability of intrusions are weeded out and replaced by applying genetic operators to yield stronger measure subsets. The method assumes that combining higher predictability measure subsets allows searching the metric space more efficiently than other heuristic techniques.

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

3.4 Combining Individual Anomaly Measures to Get a Composite Picture of the Intrusion?

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, for example, the amount of disk I/O activity, the number of page faults in the system etc. 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 is

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

This would require the joint probability distribution of the set of measures conditioned on I and $\neg I$, which would be a huge data base. To simplify the calculation, at the expense of accuracy, we might assume that each measure A_i depends only on I and is 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

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

that is, we can determine the odds 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 in the presence of intrusion.

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

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

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

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

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, relating 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 the evidence. For example, consider the trivial Bayesian network model of an intrusion shown in the figure below.

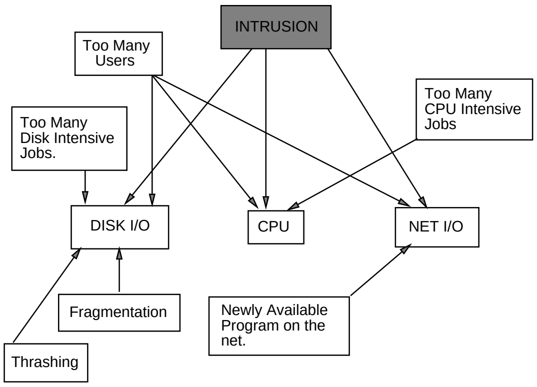


Figure 1: A Trivial Bayesian Network modeling intrusive activity

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 [Cha91].

3.5 The Conditional Probability Scheme of Predicting Misuse Intrusions

This method of predicting intrusions is the same as the one outlined in the preceding section, with the minor difference of the evidence now being a sequence of external events rather than anomaly measures. 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

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

Consider as the domain within which the conditional probability of intrusion is to be predicted, to be the campus network of a university. A security expert associated with the campus might be able to quantify the prior probability of the occurrence of an intrusion on the campus, or $P(\text{Intrusion})$, based on his experience. Further, if the intrusion reports from all over the campus are tabulated, one can determine, for each type of event sequence comprising an intrusion, $P(\text{Event Sequence}|\text{Intrusion})$, by finding the relative frequency of occurrence of the event sequence in the entire intrusion set. 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 LHS, from simple Bayesian arithmetic, for 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.

3.6 Expert Systems in Intrusion Detection

An expert system is defined in [Jac86] as a computing system capable of representing and reasoning about some knowledge-rich domain with a view to solving problems and giving advice. The main advantage here is the separation of control reasoning from the formulation of the problem solution. Some significant problems in the effective application of expert systems in intrusion detection are the voluminous amount of data to be handled and the inherent ordering of the audit trail. The chief applications of expert systems in intrusion detection can be classified into the following types.

1. To deduce symbolically the occurrence of an intrusion based on the given data (misuse intrusion detection). The chief problems in this use of expert systems are 1) No inbuilt or natural handling of sequential order of data 2) The expertise incorporated in the expert system is only as good as that of the security officer whose expertise is modeled, which may not be comprehensive [Lun93] 3) This technique can only detect known vulnerabilities and 4) there are software engineering concerns in the maintenance of the knowledge base [Lun93].

2. To combine various intrusion measures and put together a cohesive picture of the intrusion, in short, to do uncertainty reasoning. The limitations of expert systems in doing uncertainty reasoning are well known [Lun93, Pea88]. See also sec. 3.10, page 12 for a list of these limitations.

3.7 Bayesian Classification in Intrusion Detection

This technique of unsupervised classification of data, and its implementation, Autoclass [CKS+88, CHS91] searches for classes in the given data using Bayesian statistical techniques. This technique attempts to determine the most likely process(es) that generated 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. Real 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 (in the probabilistic sense) number of classes, with users with similar profiles lumped together, thus yielding a natural classification of a group of users.

However, it is not clear (to us) how well Autoclass handles inherently sequential data like an audit trail, and how well the statistical distributions built into Autoclass will handle user generated audit trails. Lastly, we are not sure if this technique lends itself well to online data, i.e. whether Autoclass can do its classification on an incremental basis as new data becomes available, or whether it requires all the input data at once.

3.8 Predictive Pattern Generation in Intrusion Detection

The assumption here is that sequences of events are not random but follow a discernible pattern. The approach of time based inductive generalization [Che88, TCL90] uses inductive generated time-based rules that characterize the normal behavior patterns of users. The rules are modified dynamically during the learning phase and only "good" rules, i.e. rules with low entropy remain in the system. An example of a rule generated by TIM [TCL90] may be

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

where E1-E5 are security events. This rule says that if the pattern of observed events is E1 followed by E2 followed by E3 then the probability of seeing E4 is 95% and that of E5 is 5% (all this is based on previously observed data). TIM can generate more general rules that incorporate temporal relationships between events.

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

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

The strengths of this approach are:

1. Claims better handling of users with wide variances of behavior but strong sequential patterns.
2. Can focus on a few relevant security events rather than the entire login session that has been labeled suspicious.
3. Claims 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 meaning associated with rules.

3.9 Neural Networks in Intrusion Detection

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. 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. The use of the neural net conceptually looks like:

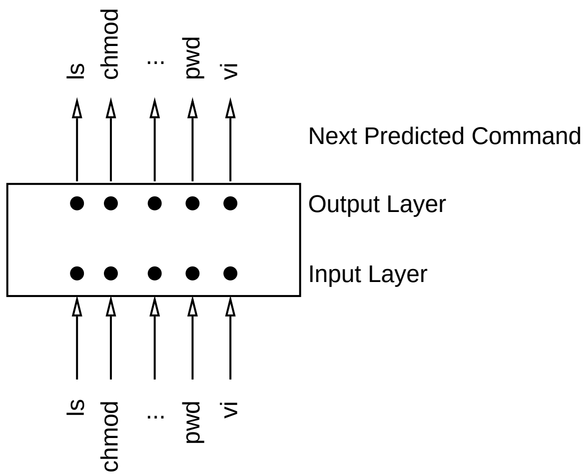


Figure 2: A Conceptual Use of Neural Nets in Intrusion Detection

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

Some of the drawbacks of this approach are:

1. The right 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. Too low and the net will do poorly, too high and it will suffer from irrelevant data.

3. This approach cannot induce patterns from samples. The neural net must be trained exhaustively on all pattern instantiations.

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 or fuzzy data.

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

3.10 Model Based Intrusion detection

This approach was proposed in [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 the system is considering a subset of these attack scenarios as likely ones under which the system might be currently under attack. It seeks to verify them by seeking information in the audit trail to substantiate or refute the attack scenario (*the anticipator*). The anticipator generates the next behavior to be verified in the audit trail, based on the current active models, and passes these behaviors to the *planner*. The planner determines how the hypothesized behavior will show up 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 that

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

must be high.

As evidence for some scenarios accumulates, while for others drops, the active models list is updated. The evidential reasoning calculus built into the system permits one to soundly 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 to dealing with uncertainty in which retraction of intermediate conclusions is not easy, as evidence to the contrary accumulates. Expert systems also have difficulty in explaining away conclusions once facts, contradicting earlier asserted facts, are made known. These problems can be avoided using a graphical approach such as evidential reasoning.

2. It can potentially reduce substantial amounts of processing required per audit record by monitoring for coarser grained events in the passive mode and actively monitoring 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 models to assign meaningful and accurate evidence numbers to various parts of the graph representing the model.
2. As proposed, the model seems to suggest the use of behaviors specifiable by a sequential chain of events but that does not seem to be a limitation of the model.
3. It is not clear from the model how behaviors can be compiled efficiently in the planner and the effect this will have on the run time efficiency of the detector. This, however, is not a weakness of the model per se.

It must be mentioned that model based intrusion detection does not replace the statistical anomaly portion of intrusion detection systems, but rather complements it. For a thorough treatment of reasoning in the presence of uncertainty see [Pea88].

3.11 Dorothy Denning's Generic Intrusion Detection Model

[Den87] established a model of intrusion detection independent of the system, type of input and the specific intrusions to be monitored. A brief description of the generic model will also help in relating specific examples of intrusion detection systems to it and viewing how these systems fit into or enhance it.

Fig. 3 describes how the generic intrusion detection system works. The event generator is generic, the actual events may come from audit records, network packets, or any other observable and monitored 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 Detection system. It contains variables that calculate the behavior of the system using predefined statistical measures. These variables are *smart* variables. Each variable is associated with a pattern specification which is matched against the generated event records and the matched records provide data to update their value appropriately. 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 knows how to update itself 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. It 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 contains a set of Expert System rules which can be triggered based on event records, anomaly records and time expirations. The rules fire as their antecedents are satisfied and make inferences which may trigger further rules and inferences. It interacts with the Activity Profile by updating it based on its rules.

There are variations on how the rules comprising the rule set are determined, whether the rule set is coded *a priori* or can adapt and modify itself depending on the type of intrusions, the nature of interaction between it and the Activity Profile, or the environment and a learning classifier. The basic theme, however,

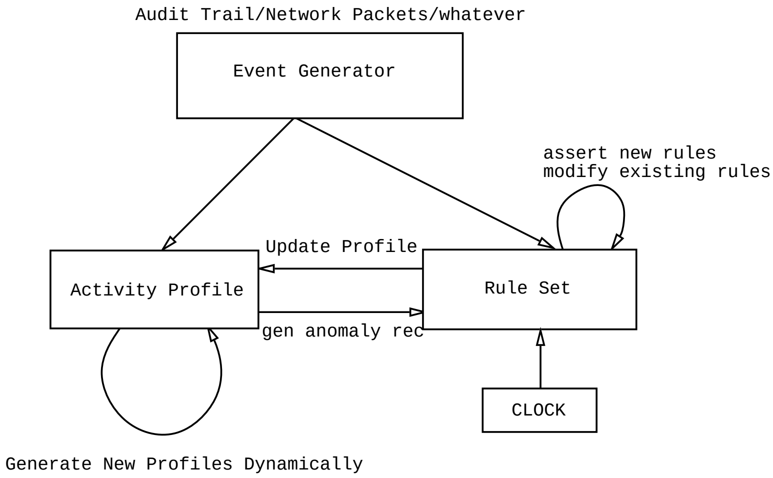


Figure 3: A Generic Intrusion Detection Model

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 does the anomaly detection, 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 as outlined in the paper. 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. It is also not clear which module TIM [TCL90] would be placed in. It 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, much like expert systems. Very recent approaches like model based approaches are too different to fit this framework directly.

3.12 Shortcomings of Current Intrusion Detection Systems

In general, the cost of building an intrusion detection system is immense. The specification of the rules of the expert system and the choice of the underlying statistical metrics must be made by one who is not only a security expert, but also familiar with the expert system rule specification language.

Intrusion Detection systems written for one environment are difficult to use in other environments that may have similar policies and concerns. This is because much of the system and its rule set must 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.

Lastly, there is no easy way to test existing Intrusion Detection systems. Potential attack scenarios are difficult to simulate and known attacks difficult to duplicate. Further, a lack of a common audit trail format hampers experimentation and comparison of the effectiveness of existing systems to common attack scenarios.

3.13 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 [Den87]. Some of them, for example [BK88, Sma88, LJL+89, SSHW88, LV89, LJL+89] use the audit trail generated by a C2 or higher rated computer, for input. Others, for example [HLMS90, HLM91] try to analyze intrusions by analyzing network connections and the flow of information in a network. Others still [SBD+91] (not elaborated in this report) have expanded the scope of detection by distributing anomaly detection across a heterogeneous network and centrally analyzing partial results of these distributed sources to piece together a picture of a potential intrusion which may be missed by the individual analysis of each source. Among non statistical approaches to ID is the work by Teng [TCL90] which analyzes individual user audit trails and attempts to infer the sequential relationships between the various events in the trail and the neural net modeling of behavior by Simonian et. al [FHRS90]. A promising approach for future intrusion detection systems might involve Bayesian classification [CKS+88, CHS91]. There are several issues that we have omitted, including audit trail reduction and browsing for intrusion detection [Wet93, Moi], non-parametric pattern recognition techniques [Lan92] etc. Audit trail reduction techniques permit the compression of audit data into coarser, higher abstraction events, which may be queried later by the security officer to retrieve information rapidly and efficiently. Non parametric techniques for anomaly detection have the advantage that they make no assumptions about the statistical distribution of the underlying data, and are useful when such assumptions do not hold. The interested reader is advised to follow up on these references for more information.

4 Intrusion Detection Using Pattern Matching

The previous section presented an overview of the various types of intrusion detectors commonly used today and the techniques they use for detecting intrusions. This section examines in depth, one particular method of detecting intrusions, namely misuse intrusion detection using pattern matching. We first examine the generic problems encountered in misuse intrusion detection via pattern matching, notably the amount of data to be matched and the generality of the signatures to be matched. We then state what we believe are important goals of any system that attempts to do misuse intrusion detection. Our attempt here is to lay down metrics for a misuse intrusion detector that we will use later, to measure the effectiveness of our model of misuse intrusion detection using pattern matching. We are not implying completeness of these criteria, but hope to have captured the most significant requirements of such a system. As we are focusing on pattern matching as the means of intrusion detection we include a brief overview of some relevant string matching results in section 4.1. Several results in approximate string matching are applicable to misuse intrusion detection, except that in some instances approximate string matching provides a more general solution than is needed for misuse intrusion detection.

To provide a generic framework for understanding the characteristics of intrusion signatures, we propose a set of orthogonal characteristics of signatures so that signatures can be studied in terms of these simpler basis properties. This partitioning is discussed in section 4.2 and section 4.3 illustrates the use of this basis set by classifying some attack classes in terms of this set. Finally, section 4.4 describes our proposed model for misuse intrusion detection.

Difficulties in Intrusion Detection using Pattern Matching

There are several difficulties in doing misuse intrusion detection using pattern matching. The generic problem, and, by far the most dominant one is the sheer rate at which data is generated by a modern processor. A typical C2 audit mechanism can fill a 1Gb disk in less than an hour on a multiuser system; a Sun SPARC 4-CPU system can saturate 2 CPUS and 100% of the disk channel logging the activities of the other two CPUS. The amount of data substantially increases in a networked environment.

The other major problem is the nature of the matching itself. An attacker may perform several actions under different user identities, and at different times, ultimately leading to a system compromise. Because an intrusion signature specification, by its nature, requires the possibility of an arbitrary number of intervening events between successive events of the signature, and because we are generally interested in the first (or all) occurrence(s) of the signature, there can be several partial matches of each signature at any given moment, requiring substantial overhead in time and space keeping track of each partial match. In some scenarios, there may be weeks between events. In others, different portions of an attack scenario can be executed over several login sessions and the system is then required to keep track of the partial matches over login sessions. Sometimes the signature may specify arbitrary permutations of sub-patterns comprising the pattern, which would be too many to enumerate separately, thus making the recognition problem much more difficult.

Before we attempt to solve any of these problems particular to misuse intrusion detection using pattern matching, we must abstract the problems and place them in the larger context of a generic solution to misuse intrusion detection. This will have the benefit, not only of providing a yardstick against which general solutions can be compared, but also serve to qualify the success of the solution that we propose, based on these criteria. To this end we propose the following to be the key design goals that the final system or matching detection model must possess in as large a measure as possible.

Generality. The model should capture all or almost all the known and hypothesized next generation attacks.

Efficiency of the common case. The model should be fast for the common case and should not unduly penalize the common case because of the uncommon case.

Real Time behavior, in as much as it can be achieved.

Portability of the signature representation across existing and future audit trail formats. The portability must be considered ascending the security rating as well as across implementations of the same security rating by different manufacturers.

Embeddability. This is a desirability, for embeddability can free the signature representation language of the clutter of providing general language constructs best left to general purpose languages within which it can be embedded.

Scalability of the model as an increasing number of signatures are added to the system.

Low Resource Overhead. The system should not consume inordinate amounts of memory/cpu to run effectively.

Our approach to designing a model for misuse intrusion detection is to start with a classification (resulting, perhaps, in a hierarchy) of intrusion scenarios and devise efficient algorithms to detect intrusions in each category. We have developed a matching model tailored for misuse detection and we show that the above mentioned goals of generality, efficiency of the common case, portability and scalability are realized in our model. The remaining characteristics of efficiency, real time behavior and low resource overhead will be determined by an implementation of our model and running tests on it. The results of the implementation will be presented in a later report. We partition the intrusion signatures along the orthogonal characteristics shown in section 4.2 and, to illustrate, we classify a few categories of attacks in terms of these characteristics in section 4.3.

A fundamental assumption made in the requirement of matching audit trails against intrusion pattern specifications is that the common case of representing intrusion scenarios weighs heavily or exclusively towards the specification of patterns embodying the "follows" semantics rather than the "immediately follows" semantics. For example, with the "follows" semantics the pattern ab specifies the occurrence of the event a *followed* by the occurrence of event b and not a *immediately followed* by b with no intervening event. In Unix regular expression syntax, this means that any two adjacent sub patterns within a pattern are separated by an implicit ".*". This assumption is reasonable under the current mechanism of audit trail generation and modern user interfaces which allow users to login simultaneously through several windows whereby audited events from multiple processes overlap in the audit trail.

Because of this overwhelming requirement of matching with an arbitrary number and type of intervening events between two specified events, the field of approximate pattern matching is relevant to intrusion detection. We outline some related results from this field in the next section. They are also used in later sections to compare against results obtained in our model.

4.1 A Brief Overview of Relevant Results in Pattern Matching

The class of pattern matching algorithms of interest in misuse intrusion detection is that of discrete approximate matching. This is because of the basic requirement of allowing an arbitrary number of events between adjacent sub-patterns. However, discrete approximate matching is too broad in scope for detecting intrusions, and its specialization of interest is termed matching with the "follows" semantics in this report. The longest common subsequence problem (see [WF74] for a fuller treatment), referred to as LCS, is used to illustrate the applicability of approximate matching to misuse intrusion detection. Consider the input to be the stream of events $a\ b\ c\ d\ d\ a\ b\ a\ c\ e\ b\ c$ to be matched against the pattern (intrusion signature) $a\ d\ a\ c$. The pattern does not occur in the input if an exact match ("immediately follows" semantics) is desired. However, if an approximate match is desired, the pattern matches the input in the following sense

$$\begin{bmatrix} a & b & c & d & d & a & b & a & c & e & b & c \\ a & \epsilon & \epsilon & d & \epsilon & a & \epsilon & \epsilon & c & \epsilon & \epsilon & \epsilon \end{bmatrix}$$

where ϵ indicates that the input event does not match any character in the pattern.

If the misuse intrusion detection question 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. That is, insertions and mismatches between the input and the pattern are disallowed. For the example problem there is a linear time algorithm as stated in observation 1, page 35. The general LCS problem can be solved using dynamic programming in $O(mn)$

time where m is the size of the pattern and n that of the input. This holds for both the off-line and the online versions of the match. In off-line matching, all the input is known in advance, in the online case matching proceeds as each input character becomes available.

Results in approximately matching various classes of patterns are summarized in the table below. These time bounds hold for arbitrary values of deletion, insertion and mismatch costs, and are not optimized for the special case of misuse intrusion detection. They are restricted to online matching because we are primarily concerned with real time intrusion detection. RE stands for regular expressions and *sequence* refers to the type of patterns used in the example above (same as fixed linear patterns defined on page 27). The column `match` denotes the type of match determined by the corresponding algorithm. An entry of "all endpts" denotes that the algorithm detects all positions in the input where a match with the pattern ends, but cannot reconstruct the match sequence, "all" denotes that the algorithm can also construct the match. However, finding all matches of a pattern in the input is an all paths source to sink problem and is computationally hard.

Pattern	Time	Space	Preproc	Match	Reference	Comment
Sequence	$O(mn)$	$O(m)$	$O(1)$	all endpts	[WF74]	Using dynamic programming.
Sequence	$O(mn)$	$O(mn)$	$O(1)$	all	[WF74]	Using dynamic programming[a] .
Sequence	$O(n)$	$O(m)$	$O(1)$	all endpts	[BYG89, WM91]	Pattern fits within a word of the computer. Small integer values of costs.
RE	$O(mn)$	$O(m)$	$O(m)$	all endpts	[MM89]	Using dynamic programming.
RE	$O(mn)$	$O(mn)$	$O(m)$	all	[MM89]	Using dynamic programming[a].

[a] Does not include the time for enumerating all matches which may be exponential

While approximate pattern matching is useful in misuse intrusion detection, the general problem in misuse intrusion detection cannot be reasonably solved by current pattern matching techniques. For example, it requires matching of partial orders, context free and context sensitive structures, and matching in the presence of time, a notion inherent in audit trail generation and very important in specifying intrusions. Traditional pattern matching has restricted itself to sequences, which is subsumed under the notion of time.

The results on pattern matching above are useful starting points for further investigation. For specialized classes of patterns, some of these results are directly applicable or can be improved [Appendix A]. It is our attempt to unify the disparate algorithms and pattern classes into one generalized model without sacrificing the efficiency of individual classes.

Because of the complexity of matching requirement for misuse intrusion detection, we partition intrusion signatures into a set of orthogonal characteristics in the next section, with the intention of classifying them in terms of these characteristics.

4.2 Orthogonal Characteristics of Patterns Used to Model Attacks

Linearity. By linearity of the patterns we mean that the specified sequence of events comprising the signature pattern is a strict chain of one event following another, with no conjunction, disjunction, * (0 or more) etc. Examples: `foo, aXbcX, aXbcY(Y.time - X.time` $\leq$ `5)`. Capitalized variables denote arbitrary events except that later occurrences of a capital variable refer to the same value that was bound in the first occurrence.

By non linearity, on the other hand, we mean that patterns can specify partial orders. One can thus say that b follows a and d follows c, but there is no connection between ab and cd. For example, the attack scenario [Ilg92]

```
1. cp /bin/sh /usr/spool/mail/root
2. chmod 4755 /usr/spool/mail/root
3. touch x
4. mail root < x
5. /usr/spool/mail/root
```

can be specified by the partial order

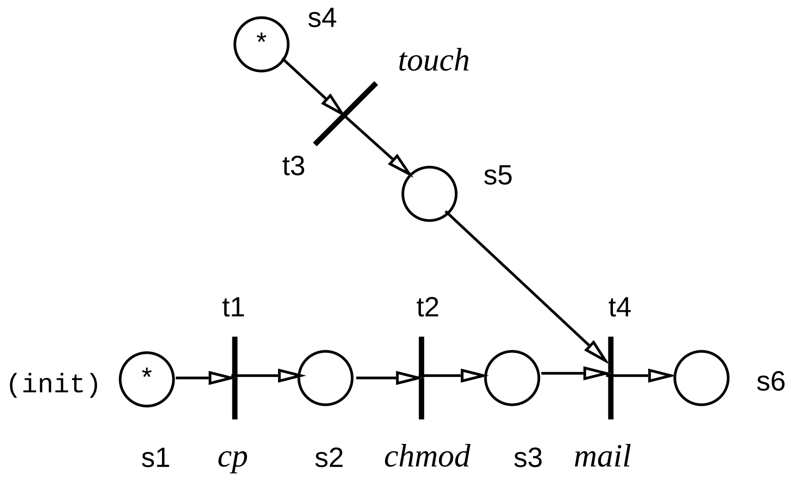


Figure 4: Representing a Partial Order of Events

as the only dependency of `touch` is that it occur before `mail`. States marked with a '*' indicate initial states (will be elaborated on later) of the partial order.

Unification. This is a characteristic of patterns which instantiate variables to earlier events and match these events to later occurring events. The instantiation is meant to be over an infinite space, otherwise the pattern can be written as a set of ORs representing all the possibilities of the instantiation. For example one can specify the pattern $abXcd$ 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, followed by d. We refer to this style of intrusion signature specification and matching, *pattern matching with unification.* For example attacks of the type

```
1. ln setuid_shell_script -x
2. -x
```

can be generalized into the pattern [Ilg92]

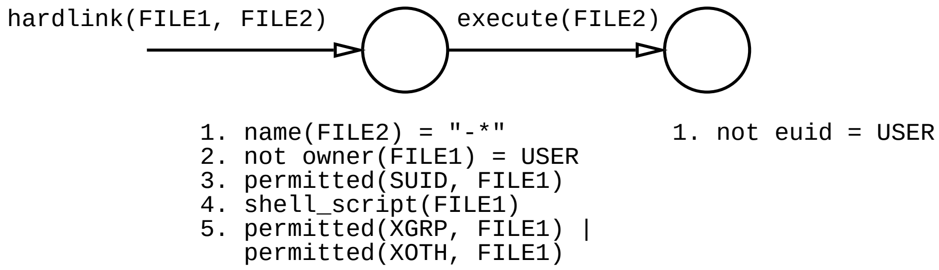


Figure 5: A pattern illustrating unification

All the capitalized identifiers, like `FILE1, FILE2` etc. are variables, and take on values corresponding to every audit record against which the pattern is matched. Bound identifiers can be referenced later to retrieve the values bound to them. Once instantiated, identifier values do not change.

Occurrence. This characteristic specifies the relative placement in time of an event with respect to the *previous* events. For example, in figure 4 there might be an occurrence specification that the event at transition $t2$ occur within 5s of transition $t1$. An event can have many previous events and the placement constraint can be specified with respect to any such event. For example, event `mail` at transition $t4$ has two previous events, `chmod` at $t2$ and `touch` at $t3$. A complex constraint might specify that $T_{t4} - T_{t2} \in [0, 2]$ and $T_{t4} - T_{t3} \in [2, 7]$. The total time within which a pattern must be matched can be easily specified by imposing a constraint on the last event with respect to the first. This constraint is an interval within which the specified event must begin. That is, for the specification $T_{t4} - T_{t3} \in [2, 7]$ to be satisfied, the event at $t3$ must begin at least 2 units before the event at $t4$, but no earlier than 7 units before.

Beginning. This specifies the absolute time of match of the beginning of a pattern. It can be significant if the detection of certain signatures is more meaningful during certain times. For example, invalid logins at night might be more significant than during daytime. This constraint permits use of ranges.

Duration. This characteristic places constraints on the time duration for which the event must be active. This is also specified as a range.

Three more characteristics determine significantly the kinds of theoretical bounds that can be placed on the matching solution.

Dynamic Input. In this case the input is not known in advance before matching begins, but input is generated dynamically. This is also referred to in string matching literature as *online* matching, as opposed to *off-line* matching in which the input is known in advance. If the algorithm for the static case is efficient while that of the dynamic case intractable, a window of previous input may be considered for the application of the static algorithm, with a concomitant loss of accuracy in detection.

Dynamic Patterns. Here patterns can be added or deleted as matching proceeds. This might be useful in model based intrusion detection.

All or One Match. This is the requirement of finding all matches of a pattern in a given input stream, or simply finding whether the pattern occurs in the stream.

In order to show the relevance of these pattern characteristics and the resulting simplification on examining intrusion signatures in terms of these simpler characteristics, we take a few categories of attack and show how each can be represented by the orthogonal characteristics that we have mentioned.

4.3 Some Examples of Attack Scenario Classification

The previous section examined characteristics of intrusion patterns by breaking them into simpler, independent properties. These properties are used to build a classification of signatures in this section. This classification is not strictly a hierarchy as characteristics easy to match at lower levels of the classification become intractable at upper levels. Thus, some of the characteristics present in lower levels are dropped in higher levels. The classification, meant to be illustrative, is outlined below and summarized in the following table.

1. **Existence.** The fact that something(s) ever existed is sufficient to detect the intrusion attempt. Simple existence can often be found by static scanning of the file system, ala the COPS [FS91] and Tiger [SSH93] systems. This means looking for changed permissions or certain special files.

2. **Sequence.** The fact that several things happened in strict sequence (with the "immediately follows" or "follows" semantics) is sufficient to specify the intrusion. The vast majority of known intrusion patterns fall into categories 1 & 2.

3. **Partial order.** Several events are defined in a partial order, for example as in fig. 4.

4. **Duration.** This requires that something(s) existed or happened for not more than or no less than a certain interval of time.

5. **Interval.** Things happened an exact (plus or minus clock accuracy) interval apart. This is specified by the conditions that an event occur no earlier and no later than x units of time after another event.

In the table below, Y stands for yes and N for no. Under the linearity column LIN, L stands for linear and NL for non linear. $\rightsquigarrow 0$ means that existence is a limiting case of linearity in which the sequence length tends to 0. Also note that the choice of particular values under LIN, UNIF etc. corresponding to the types of attack is arbitrary, and reflects our belief in the most promising direction of investigation for efficient algorithms.

#	Type of Attack	LIN	UNIF	OCC	BEG	DUR
1.	Existence	$\rightsquigarrow 0$	Y/N	$[0, 0]$	Y/N	N
2.	Sequence	L	Y/N	$[0, \infty]$	Y/N	N
3.	Partial Order	NL	Y/N	$[0, \infty]$	N	N
4.	Duration	L	N	$[0, \infty]$	N	Y
5.	Interval	L	Y/N	$[x, x]$	N	N

Table 1: Classification of some attack types in terms of simpler pattern characteristics.

4.4 The Model of Pattern Matching — An Overview

This section introduces the pattern matching model proposed for matching intrusion signatures. It is based on Colored Petri Nets. Each intrusion signature is represented as a Petri net and the notion of start states and final state is used to define matching in this model. The notion of `linearity/non linearity` is easily subsumed in this model because of its generality, `unification` is introduced through the use of globally visible variables, and `occurrence, beginning & duration` constraints are introduced through the use of guard expressions in the net. In the subsequent section we analyze the model to see how well it does against the criteria specified earlier for an ideal misuse detector.

The following computation models were considered for their suitability in detecting intrusions via pattern matching. The premise in investigating these models is that they are to be used as intuitively and directly as possible in representing intrusion signatures. For example, attribute grammars are known to be Turing complete, so a program in any model of computation can be mapped to an equivalent attribute grammar. However, using an AG to recognize a set of sentences whose underlying structure does not lend itself to being represented as a CFG is not very intuitive and, therefore would not be very useful for the human responsible for writing the intrusion signatures.

1. **Regular Expressions.** Traditional matching with regular expressions is fast (linear in the input with the "immediately follows" semantics) and well understood. Approximate matching is polynomial in its input. However, regular expressions can represent only very simple attack scenarios not including non linearity, unification or duration.

2. **Deterministic Context Free Grammars** like LR, LALR are easily subsumed under Attribute Grammars. By themselves they are of limited use because they cannot handle unification, occurrence or duration. There is no easy way to extend them to match with the "follows" semantics.

3. **Attribute Grammars** provide a very powerful representation mechanism but do not provide a natural human understandable graphical model for their representation. Graphical models can be imposed on them which would make them very similar to CP nets. Furthermore, to be useful to humans writing intrusion signatures as attribute grammars, the underlying signature specification would need to be context free, and for efficient matching, deterministic context free. Like CFG's there is no easy way to extend them to match with the "follows" semantics.

4. **Colored Petri Nets.** The main advantage here is that the model itself is conceptually so general that extensions to it do not change it substantially. They are also naturally represented as graphs and have well defined semantics.

We have adapted the CP net model [Jen92] for pattern matching because of its generality, conceptual simplicity and graphical representability. This model represents the form in which patterns are internally stored and matched. 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. Work in this direction is planned at COAST once the underlying matching model is validated and thoroughly tested.

To recall briefly, our attempt is to match incoming events (audit trail records or higher abstractions) to the patterns representing intrusion scenarios. Consider, as an example, the representation of the attack scenario on page 19, as represented in figure 4. `s1` and `s4` are the initial states of the net and `s6`, its final state. Any net in our model requires the specification of ≥ 1 initial states (to represent partial orders of events) and exactly 1 final state. The circles represent states and the thick bars, the transitions. At the start of the match, a *token* is placed in each initial state. This is called the **initial marking** of the net. Each state may contain an arbitrary number of tokens. An arbitrary distribution of tokens in the net is referred to as its **marking**.

The net also has associated with it, a set of variables, in the Prolog sense, i.e. assignment to the variables is tantamount to unification, which are *globally visible to the pattern.* This is unlike variables in CP-nets which are local to transitions. Variables local to a transition can be simulated by using the global variable name space. Allowing only global variables does not dilute the expressiveness of the model. We feel that the choice between local/global variables is a user interface issue and rests in the domain of language design, if a language were to be designed as an alternative representation of this model. Local variable specifications in such a language can easily be translated to the above described model by choosing a separate part of the global variable name space, much like is done in imperative language compilation. For related work which uses a language front-end for an underlying matching model in intrusion detection see [HCMM92, WB90].

Each token maintains its own local copy of these globally visible variables (depending on the pattern the token is associated with), the reason being that each token can make its own variable "bindings" as it flows towards the final state. In CP-net terminology, each token is colored, and its color can be thought of as the cross product of the variable types associated with the pattern.

The net also contains a set of directed arcs which connect places to other places and transitions in the net. The arcs which connect places to other places are ϵ transitions in which tokens can flow nondeterministically from one place to another without being triggered by an event. Each transition is associated with the type of event, called its **label**, which must occur before the transition will **fire**. For example in figure 4, transition `t1` is labeled with the event *cp*, `t3` is labeled with the event *touch* and so on. Nondeterminism can be specified by the labels given to the transitions by labeling more than one outgoing transition of a state by the same label. There is, however, no concurrency in the net, an event can fire at most 1 transition. A transition is said to be **enabled** if all its input states contain at least one token.

Optional expressions, called **guards**, can be placed at each transition. These expressions permit assignment to the global variables of the pattern, for example the values of matched event fields; variable testing for equality, $<$, $>$ etc.; calling built in functions on the variables etc. Guards are boolean expressions which evaluate to *true* or *false*. Compare this with [Ilg92, PK92] in which guards are placed on states rather than transitions. Guards are evaluated in the context of the event which matches the transition label and the set of *consistent* tokens which enable the transition. For example in figure 7, in order for transition `t4` to fire, there must be at least 1 token in each of states `s3` and `s5`, and the enabling pair of tokens (one from `s3`, the other from `s5`) must have consistently bound (unifiable) pattern variables, and the unified token and the event of type `c` together must satisfy the guard at `t4`. 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, all the input tokens that have caused the transition to fire are *merged* to one token, and copies of this unified token are placed in each output place of the transition.

The process of merging resolves conflicts in bindings (i.e. unifies or rejects the combination of tokens) between the tokens to be merged 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 the token encountered along its way to this state.

As an example consider the following signature which raises an alarm if the number of unsuccessful login attempts for user `kumar` exceeds 3 within a minute. The example below only serves to illustrate the idea, and has no bearing on the signature representation format.

```
login(user = kumar, exception = YES, time = T1)  #failed login due to exception
login(user = kumar, exception = YES)
login(user = kumar, exception = YES)
login(user = kumar, exception = YES, time = T2)
```
if $T2 - T1 < 180s$

Its corresponding CP-net is

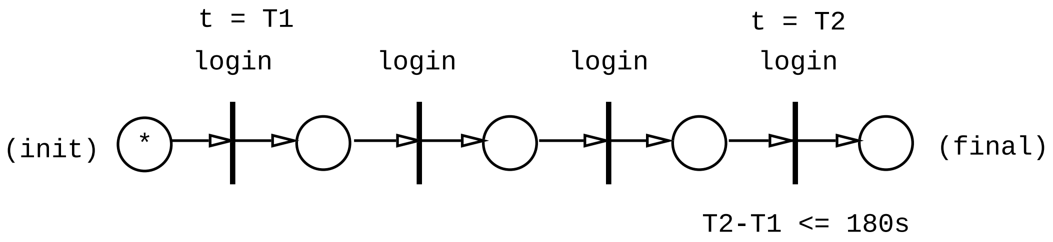


Figure 6: Four failed login attempts within a minute.

$T1$ and $T2$ are global variables associated with the pattern.

As another example, consider the fixed partial order pattern in figure 7

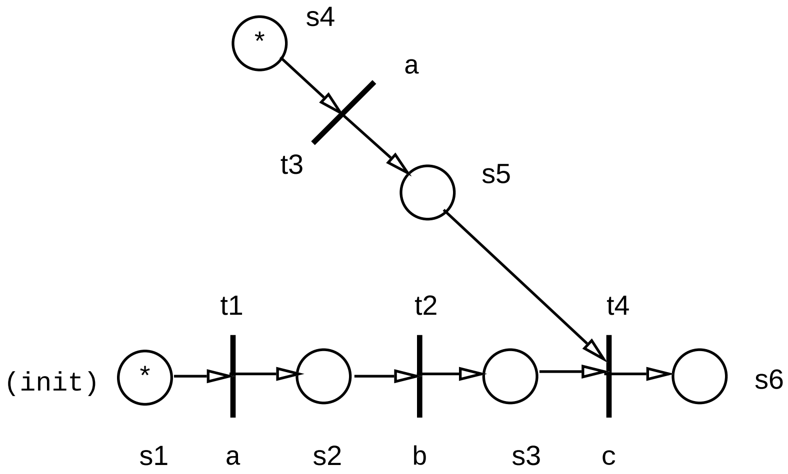


Figure 7: Fixed Pattern Simulation

and the event sequence *abac*. The way the pattern is matched is:

Pattern consumed	TokenConfig	Comment
$.abac$	$\{s1, s4\}$	
$a.bac$	$\{s2, s4\}$	Only 1 transition can fire, no concurrency. Nondeterminism is allowed.
$ab.ac$	$\{s3, s4\}$	
$aba.c$	$\{s3, s5\}$	
$abac.$	$\{s6\}$	The two tokens are merged to one. From the token in `s6` we can reconstruct the path of each individual token from the **initial marking**.

Note that merging of tokens occurs in step 4 and so does conflict resolution. This example illustrates a nondeterministic search of the pattern, in which tokens are *removed* from one or more states and placed in others, and tokens always make the right choice in their transitions. 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. In the case of matching in the absence of guards, this duplication is probably not required, but because events are tagged with data fields, this in a sense makes every event unique, and because tokens carry bindings with them, 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 future match, and a duplicate created for the current match.

The guards allowed at the transitions are grouped into the following categories:

File Test Operations for example, testing for a file being readable, writable, executable by effective/real user id, existence of a file, testing the size of a file etc.

Set Manipulation Operations like adding, deleting, testing for the presence of an element in a set etc.

System Interaction Functions like system(), raising and lowering audit levels, interacting with processes etc.

this is instantiated to the most recent transition (signature action). It may be empty, i.e. NULL. It provides a hook into the event (audit record) matched at this transition.

last takes a hook into an event (transition) as input and returns the (set of) events (audit records) matched at the immediately previous event. In this specification of expressions, one cannot refer to values in other tokens yet.

Logical, Arithmetic and Matching Operators like `&&, ||, =~` etc.

Note: The process that does this matching must not itself generate audit records in response to its computation. Further, for the sake of efficiency, the average time to exercise each token (over all the patterns) must be much less than the average time between the generation of successive events in the system.

The Nature of Events

Events in general are tagged with data. In particular, the time stamp at which the event "occurred" is of special importance because of the monotonicity properties of time. The events can have an arbitrary number (though usually a small number) of tag fields. The exact number and nature of the fields is dependent on the type of the event. Mathematically one can think of the events as being tuples with a special field indicating the type of event. For example, one can think of the event a occurring at time t to be the tuple (a, t), where a denotes the type of the event.

5 Analysis of Our Matching Model

The previous section described our matching model in some detail, this section presents some of its theoretical properties. The complexity of matching in this model increases rapidly with increasing complexity of the patterns. At the simplest end are linear patterns without guards, for which well known algorithms from discrete approximate matching [MM89, WF74, BYG89, WM91] are applicable. Such matching can be done deterministically and efficiently, without requiring much preprocessing of the patterns.

If the pattern is `fixed` but has regular or partial order features, preprocessing it to yield an equivalent deterministic automaton is very expensive. It takes exponential time in the worst case to convert an *NFA* to a *DFA* [ASU86, pp. 117–121]. Converting a nondeterministic partial order pattern to an equivalent deterministic automaton is even more expensive [Appendix A]. This forbids preprocessing of any realistic set of partial order patterns before matching. Adding unification to fixed linear patterns makes the problem of matching NP Complete [claim 5, Appendix A].

The expense of preprocessing the pattern types mentioned above necessitates a simulation of the non deterministic pattern representation. Structures in the input may also be exploited to improve matching in special cases. Two such structures are outlined [observation 8, claim 3; Appendix A]. Both reduce an exhaustive search of the input for matching under certain conditions. Observation 8 exploits the monotonicity of audit record fields, for example its time stamp, which is non decreasing. A monotonic expression that fails for a monotonic field of a particular audit record, fails for all subsequent records as well (depending on the directions of the monotonicities of the expression and the field). Claim 3 is based on the observation that a single match for the pattern ab (with the follows semantics) is satisfied for any choice of a and a subsequent b. Other less significant results are also presented in Appendix A.

Sections 5.2 and 5.3 consider the two important, practical aspects of scalability and portability in the model. The measure of scalability is the lack of degradation in performance with an increased number of patterns. Ideally, the degradation is sub linear with a linear increase in the number of patterns, albeit with a linear increase in the preprocessing time. Common subexpression elimination techniques can be used to exploit the commonality of guard subexpressions placed at the various pattern transitions to reduce this degradation [Appendix B, sec 5.2].

Portability, considered in sec 5.3 is concerned with ensuring that intrusion signatures can be moved across sites without rewriting to accommodate fine differences in each vendor's implementation of the same security rating. It also ensures their transparent movement to higher security rated systems. An abstract audit record definition and a standard definition of a virtual machine to represent guards, ensures that patterns precompiled to an intermediate representation can be moved across systems with minimal overhead

5.1 Theoretical Results

Definitions

`Fixed Linear Pattern Matching`: This denotes matching in the KMP [KMP77] or Boyer Moore [BM77] sense, i.e. the pattern consists of a sequence of known events (symbols in KMP & BM) forming a chain of one event following another, with no conjunction, disjunction or other "regular" features. Examples: `abc, abracadabra, foo`. This is in contrast with patterns in which all events are fixed (i.e. prespecified), but the pattern is not linear and may have regular or partial order features. For example, the pattern specification in figure 8 below

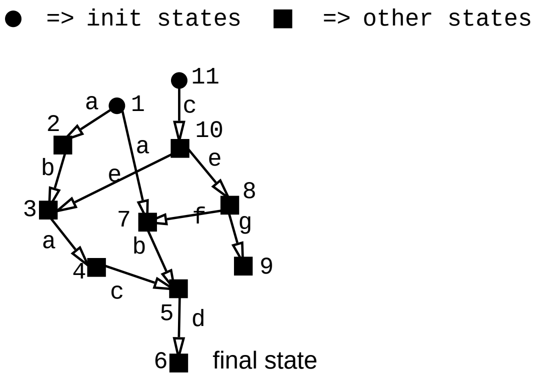


Figure 8: A non linear fixed pattern

is fixed but not linear. The numbers represent states (vertices in the directed graph) and alphabets, representing events, label the edges. Such patterns are denoted `fixed`.

`Linear Pattern Matching`: This denotes matching in patterns in which the specified sequence of events comprising the pattern may not be fixed (allowing for unification [sec 4.2], for example), but the pattern is a strict chain of one event following another, with no conjunction, disjunction, 0 or more etc. For example `aXbcX, aXbcY(Y.time - X.time` $\leq$ `5).`

The key result to note is that the complexity of matching in this model, with the exception of generalized partial orders, is exponential [obs. 10, pg. 44, Appendix A]. Matching results for specialized classes are summarized here. The derivation details can be found in Appendix A. The attack classification listed below, and its properties in terms of pattern characteristics, is similar to that in table 1. m is the size of the pattern, and n that of the input.

Attack	LIN	UN	OCC	BEG	DR	Time	Space	Match	Ref	Cmnt
Existence	$\rightsquigarrow 0$	N	$[0, 0]$	Y/N	N	$O(m)$	$O(m)$	one	–	A
Sequence I	L	N	$[0, \infty]$	Y/N	N	$O(n)$	$O(m)$	one	obs. 1, pg. 35	
Sequence II	L	Y	$[0, \infty]$	Y/N	N	c^n	$O(m)$	one	cl. 1, pg. 40	
Reg. Expr.	NL	N	$[0, \infty]$	Y/N	N	$O(mn)$	$O(m)$	one	obs. 3 pg. 36	
Partial Order	NL	N	$[0, \infty]$	Y/N	N	$O(nm^i)$	$O(m)$	one	pg. 39	B
Duration	L	N	$[0, \infty]$	N	Y	c^n	$O(m)$	one		C
Interval	L	N	$[x, x]$	Y/N	N	$O(r \log n)$	$O(n + m)$	one	[AA93]	D

Table 2: A summary of matching algorithms for different attack classes.

`Legend:`

`DR` is the duration characteristic associated with the events comprising the pattern.

`Match` refers to all or one match.

`Ref` lists the reference explaining the result in more detail.

`Comment:`

A. Existence patterns may have an associated clock event to specify evaluation of the condition at specified intervals.

B. i is the number of initial states of the partial order. It is rarely desirable to convert this type of pattern to a deterministic matching automaton.

C. A general linear pattern can be matched in exponential time by trying all possible sequences of the input. We are unaware of better bounds for this case.

D. $r = \Sigma_a P(a) \times T(a)$, $P(a) = \#$ of occurrences of the symbol a in the pattern, $T(a) = \#$ of occurrences of a in the input, which yields an off-line upper bound of $O(mn \log n)$.

Intrusions that can be determined by testing a condition are classified in the class `existence`. This class has no use of unification because instantiated variables cannot be referenced at later transitions, as there are none. m includes the representation of the condition. Evaluating the condition requires $O(m)$ time, in direct proportion to its size because conditions are loop free (see the example in Appendix C).

When the pattern is specified as a sequence of events, it is classified as a `sequence`. Matching a sequence without conditions or unification can be done efficiently in $O(n)$ time when only a single match is desired. If all matches are desired, it takes $O(mn)$ space and exponential time. Finding all matches can be transformed to finding all distinct paths between a source node and a set of sink nodes and there may be an exponential number of such distinct paths [Appendix A]. If unification is desired in a sequence, the problem of matching is NP Complete. This makes it as difficult to match linear patterns that require unification as matching general linear patterns [obs. 10, pg. 44]. A majority of intrusion scenarios common today belong to either the `existence` or the `sequence` class. The NP Completeness bound, however, holds for arbitrary patterns and input. In practice, there are several factors that can significantly reduce the bound. They are:

- Arbitrary general unification is rarely desired. For example, unification over the event type is seldom required. Unification is most often done over audit record fields of a given audit record type, which significantly reduces the computation required for matching.

- Unification is often not over an infinite space, and patterns can be written to use alternation to significantly reduce its matching complexity. This might result in a linear pattern becoming non linear, but if there is little non determinism in the resulting pattern, matching takes $\Theta(n)$[4] time rather than general result of $O(mn)$ time.

- Patterns are most often deterministic, including regular expression and partial orders, and the number of initial states of the partial order is small. Therefore, practically, matching regular expressions can be done in $O(n)$ time, and partial orders in much less time than the worst case bound.
- Multiple instances of the same intrusion pattern rarely overlap, or can be carefully written to ensure that they do not. This means that on a successful match, all tokens that are ancestors of the matching combination can be destroyed. That is, we need not look for overlapping matches.

The rest of the categories and the associated time and space bounds are self explanatory. Further optimizations can be made in matching several of these categories. They are outlined in Appendix A. Matching in the general case can also be improved by exploiting the monotonicity of audit record fields. Two such optimizations are detailed in Appendix A. In a practical system, in order to control space requirements, tokens that will never lead to a match should be garbage collected. For example, tokens may have variables bound to filenames, process numbers and other system objects. If these objects are destroyed the corresponding tokens may become garbage and can be collected. The machinery required to do this is not presented here as it does not add conceptually to our model of matching. The machinery also obviates the need to specify such explicit exclusion of events in the pattern, resulting in simpler pattern specifications.

It is now shown how multiple patterns can be matched in our model.

5.2 Scalability *or* Matching Many Patterns Efficiently

The problem of scalability is controlling the complexity of matching as the number of patterns to be matched increases. This section investigates approaches to matching several patterns in the audit trail. Matching several *fixed linear* patterns takes as much time as a regular expression formed by the alternation of the patterns in the worst case. The case of several regular expressions is subsumed under matching a single regular expression. Matching several partial orders is discussed under matching several general patterns. If a virtual machine and its instruction set is defined to represent and evaluate guards, common subexpression elimination techniques from compiler design [ASU86] can be used to improve their evaluation. For an example of this see Appendix C which shows a 25% asymptotic runtime reduction in multiple guard evaluation.

Scalability results derived in our model are summarized below. The classes of attack are the same as in Table 2. k is the number of patterns to be matched simultaneously. Each of the k patterns is of size $m_1, \ldots, m_k$ respectively and $M = m_1 + \cdots + m_k$ is the total pattern size. n is the size of the input and m is the largest m_i. Details can be found in Appendix B.

[4] A function is $O(f)$ if it grows no faster than f, it is $\Theta(f)$ if it grows at the same rate as f.

Attack	LIN	UNIF	OCC	BEG	DUR	Time	Space	Mat	Ref
Existence	$\rightsquigarrow 0$	N	$[0,0]$	Y/N	N	$O(M)$	$O(M)$	one	–
Sequence I	L	N	$[0,\infty]$	Y/N	N	$O(Mn)$	$O(M)$	one	obs. 11, pg. 46
Sequence II	L	Y	$[0,\infty]$	Y/N	N	exp	$O(M)$	one	–
RE	NL	N	$[0,\infty]$	Y/N	N	$O(Mn)$	$O(M)$	one	obs. 3
Partial Order	NL	N	$[0,\infty]$	Y/N	N	$O(knmi^n)$	$O(M)$	one	–
Duration	L	N	$[0,\infty]$	N	Y	exp	$O(M)$	all	–
Interval	L	N	$[x,x]$	Y/N	N	$O(kr \log n)$	$O(n+M)$	one	obs. 6, [AA93]

Table 3: Summary of matching multiple patterns in the same attack category.

In most classes above, the approach to matching several patterns is to match each independent of the other. The following observation for matching general patterns reduces the complexity of matching.

> *For any input audit event, only transitions labeled with that event type need be exercised. If the event types are uniformly distributed over the pattern and the input, substantial savings may result.*

The table above did not take into account the presence of guards at the transitions. One approach to matching in the presence of guards might be to ignore them during matching, and on success, to verify that they are also satisfied. The chief difficulty with this approach is that techniques for space efficient approximate matching do not store the entire dynamic programming matrix, but only the current and the previous columns (each column equals the size of the pattern). This means that all positions in the input that approximately match the pattern can be determined, but not all the matches. Thus there is no way to verify that guards are also satisfied without determining the exact match because guard evaluation is dependent on audit record fields.

Approaches to Matching Multiple General Patterns

For the most general case (each pattern being a CP-net) involving multiple patterns, there are several approaches, the simplest is to *exercise* every pattern against every input event. Pictorially, this looks like:

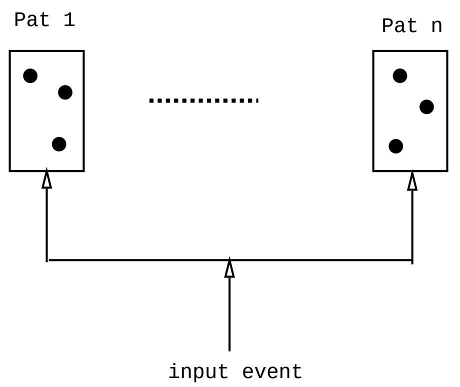


Figure 9: One approach to handling multiple patterns

The filled dots represent tokens corresponding to enabled transitions in the patterns, some of which may exercise on the current event (if the guards at these transitions evaluate true). Each pattern keeps track of the tokens that may need to be duplicated or moved, and is given control to exercise the tokens, even if the pattern may return the control immediately because no transition fires on the input event.

A slightly improved approach is to compute, for every input event, the tokens, across all the patterns that *can possibly* be exercised. This approach looks like:

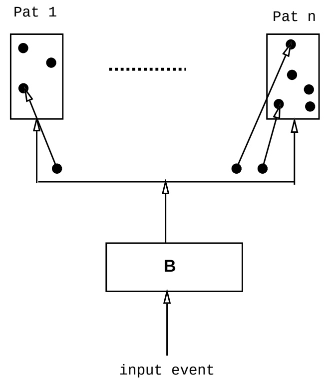


Figure 10: Another approach to handling multiple patterns

The box B takes in an input event and computes the tokens, across all patterns, corresponding to enabled transitions, that can possibly make a transition on that event. This makes it unnecessary to hand over control to patterns that have no enabled transition to be exercised on the current event and also avoids the lookup done by each pattern individually to determine enabled transitions (though that may be very efficient) by coalescing the individual lookups.

However, we can do better than the above approaches. These approaches disregard the evaluation of guards at the transitions while computing the list of active tokens, and evaluate the guard for each enabled transition, even when the guard expressions are similar. Common subexpression elimination techniques from compiler theory can be applied to avoid their re-evaluation across guards. This idea can also be combined with that of representing match patterns as a network [For82]. The amount of common subexpression elimination realized depends on the number and nature of the guards. More expression elimination can be achieved if the guards can be broken into simpler, more primitive expressions, similar to the case of compiling code.

This can be done by defining a virtual machine with simple instructions to evaluate the guards. A standard definition of the machine instruction set will also help in the portability of "compiled" guards and patterns across machines. The virtual machine needs to be extensible so that newer information present in more sophisticated audit trails can be easily incorporated. Some instruction types supported are:

1. Assignment, i.e. unification.
2. Indexing to retrieve fields from the abstract audit record.
3. Useful inbuilt operations to interact with the operating system and general purpose utility functions.

4. Regular expression matching.
5. Testing of conditionals.

For an example of such a machine, see Appendix C.

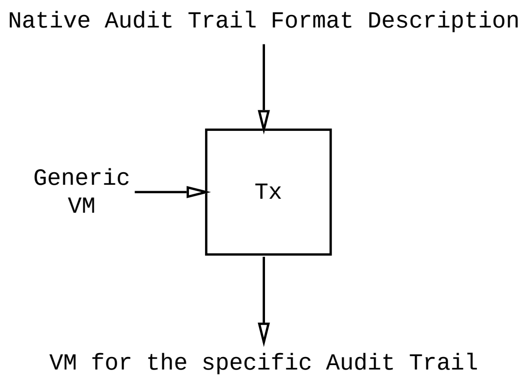
5.3 Portability

By portability we mean:

- Portability of intrusion signatures across different interpretations of the same auditing level, by different manufacturers of the same underlying operating system. The similarity of the underlying operating systems dictates similarity of flaws across the versions. For example the differing interpretations of C2 security by SunOS and AT&T SVR4.

  By representing signatures in a machine independent format; using a standard definition of the virtual machine matching the patterns; and using an abstract, extensible audit trail that is independent of any specific vendor implementation the portability of intrusion signatures can be ensured.

  Two levels of translation of the native audit trail, once to the abstract audit trail, and again to read and process the abstract trail internally for matching, can be avoided by an appropriate description of the native audit trail format in the backend of the virtual machine. The virtual machine for any specific audit trail architecture would then look like:



- The same signatures should be usable when ascending the auditing level for a given OS.

  This can be achieved with an appropriate definition of the abstract audit trail, when signatures written for a lower security auditing level are moved to a higher security auditing level. The converse, however, is not true in general, because there may be information in the higher security auditing lacking in the lower one. In this case an appropriate error is flagged.

  Generically, signatures can be thought of in two equivalent ways, either as program specifications, or as data to a matching program. In the former, the program specification is compiled for each audit trail while in the latter, the matching program is compiled for the specific audit trail. The dependence on specific audit trail architectures can be removed by defining an intermediate, standardized audit trail but results in an extra level of indirection in matching.

6 Extensions/Future work

When the guards corresponding to all transitions with the same label are merged together for CSE, it is not clear what order they must be merged in. There are no semantics to the expressions forming each guard and ideally all the expressions in all the guards must be merged in an order to achieve maximum overlap between the expressions, taking into account the probability of occurrence of each expression. Rete network generation has applicability to this problem and can perhaps be used to compile the guards in an order to achieve much better performance.

It would be interesting to determine if certain audit records can be omitted, i.e. not exercised on any token in any pattern, without affecting the accuracy of the detection, for the class of intrusions of interest. If the accuracy is affected, it would be useful to characterize the relationship between the types and amount of omission and the accuracy of detection for different classes of intrusions.

We mentioned in section 4.4 that states can have arbitrary number of tokens. This may not be possible in practice. It would be of interest to determine how matching is affected if the capacity of states to hold tokens is restricted and some replacement scheme put into effect to determine the token to be discarded when adding newer generated tokens. It might be that the class of intrusions common today follow a locality of attack rule with respect to the tokens.

While not directly related to this work, it would be useful to design a Bayesian belief network which incorporates the interdependence between the various anomalies and intrusions common today, as suggested in section 3.4. Such a model can be useful in combining observed data on a system and assigning belief to the hypothesis that an intrusion is occurring given the observed data.

7 Conclusions

In summary, the paper followed the following broad outline:

- It studied and formalized the problem of applying a pattern matching solution to misuse intrusion detection [sec. 4]. It proposed a set of metrics to evaluate a generic misuse intrusion detector.
- It presented a simple engineering solution to the problem [sec. 4.4].
- It analyzed the solution to derive some of its theoretical properties and outlined some of its limitations. It suggested methods and heuristics to improve the solution in practice [sec. 5, Appendices A, B, C].

The model is interesting and appealing from a theoretical standpoint. However, its true test is an evaluation of its implementation running under "live" conditions. We hope to implement this model and get experimental results in the near future. The results will be reported in a future report.

Once the model is implemented our goal is to make it, as well as several sample intrusion patterns available to anyone who wants it. Users can then add their own libraries of signatures to deal with intrusions specific at their site. Interested parties are invited to contact the authors for current status and availability information.

8 Acknowledgements

We thank the Infosec Division of the Department of Defense for funding this work. Reva Chandrasekaran provided valuable comments on structuring the paper. Discussions with Dr. DeMillo, Dr. Young and Dr. Atallah provided insight into the problem.

9 Appendix A

This appendix presents results in matching various categories of attack patterns in our model. All results, unless otherwise specified, are derived for the case of matching with the follows semantics. Arguments are sometimes made in lieu of formal proofs to substantiate the claims. Formal proofs will appear in a subsequent report. The results are presented as `claims` and `observations`. `Claims` are more significant and weighty while `observations` should be regarded as corollaries.

Matching Fixed Linear Patterns

Observation 1: Online fixed linear pattern matching with the *follows* semantics can be done in linear time [Man89, ex. 6.48, pg. 181].

For example, to determine that the pattern *ba* occurs in the input *abcaa* (in the *follows* sense) requires only a single scan of the input. Compare this result with that of approximately matching a sequence (fixed linear pattern) with arbitrarily specified penalties on the deletion, insertion and substitution of pairwise symbols. That result requires $O(mn)$ time and $O(m)$ space [section 4.1] for the online case. m is the size of the pattern, and n that of the input. The primary reason for the $O(n)$ time for the "follows" semantics is that there are no insertion and substitution edges in the approximate matching graph, only matching and deletion edges. Compare the graphs for matching *ba* against the input *abcaa* in the general case and with the follows semantics. As opposed to the much fuller graph in the general case, the graph for the *follows* semantics is more sparse.

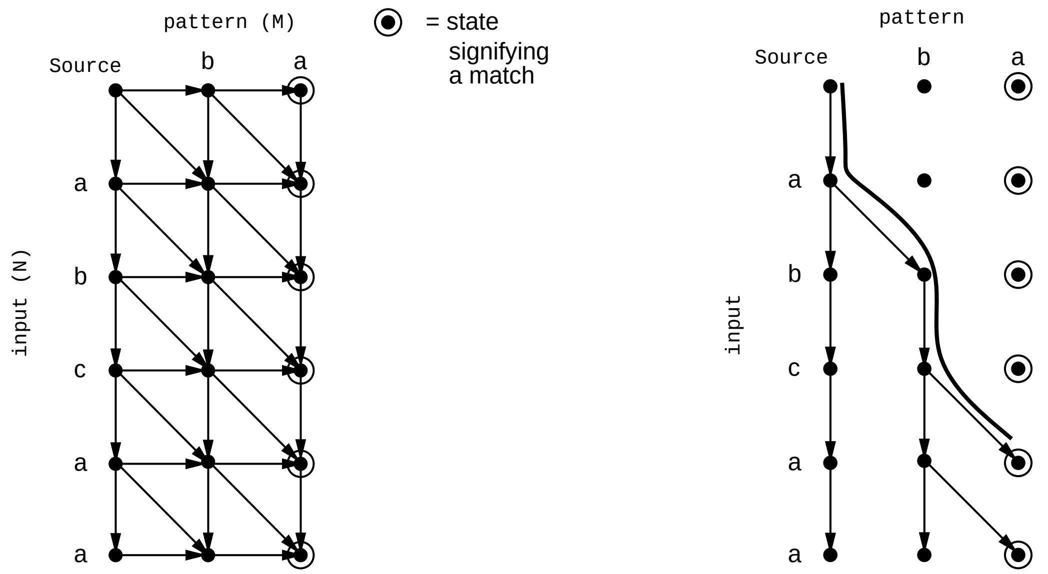
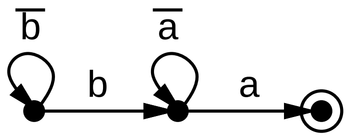


Figure 11: Alignment graphs representing matching with arbitrary insertion/deletion/substitution costs (left) and with the "follows" semantics (right).

Matching *ba* with the follows semantics is the same as matching `(.*)b(.*)a(.*)` with the immediately follows semantics, and when only the determination of presence or absence of the pattern in the input is desired, this pattern is equivalent to matching `([^b]*)b([^a]*)a` in the input. The latter can be directly represented as the deterministic automaton



and does not require a simulation of the non deterministic automaton representing `(.*)b(.*)a(.*)`, or its conversion to an equivalent *DFA*. Converting an *NFA* encoding a fixed linear pattern to be matched with the *follows* semantics, into a *DFA* with the *immediately follows* semantics results in a chain of states with no back-edges to earlier states. This obviates the need for backtracking and failure functions.

The previous discussion was for matching a single pattern with the *follows* semantics, where any match would suffice. When all matches are desired, the alignment graph of the pattern and the input can be used to yield all matches in exponential time and polynomial space. If the pattern size is small enough to fit the word length of the computer on which matching is done, all matches can be determined in linear time using the algorithm of Baeza-Yates and Gonnet [BYG89] augmented for the wildcard case by Manber and Wu [WM91]. Both algorithms determine whether any character position in the input ends a match with the pattern by conceptually sliding the pattern across the input. This does not result in the correct number of all possible matches because all ways of matching terminating at an input character are coalesced into a single match. This procedure can also not reconstruct the exact match sequence. To determine all matches and the match sequences, the entire alignment graph must be preserved, and all paths from the source to any sink (states marked ◉) must be determined.

Observation 2: There are at most exponential paths from the source to any sink node in the alignment graph for matching a fixed linear pattern with the follows semantics.

The graph allows only two kinds of edges, matching and deletion. If the size of the pattern is m and that of the input is n, exactly m matching and $n - m$ deletion edges are required. This implies that we must pick m events from the input corresponding to matching edges, and the rest are deletion edges. Because the order of the input must be preserved, we have at most ${}^nC_m$ choices. As $\Sigma_i {}^nC_i = 2^n$, there are at most 2^n such choices.

Matching Regular Expressions

Matching `fixed non linear` patterns is more involved than matching fixed linear patterns. Two cases are considered, regular expressions and partial orders.

Observation 3: Regular expressions can be matched with the follows semantics in polynomial time.

Approximate matching with regular expressions has been solved by Wagner and Seiferas [WS78] and Myers and Miller [MM89]. The approach taken in [MM89] is followed here. The solution constructs a non deterministic finite automaton from the regular expression in a special way. States of this automaton label the input symbols and its graph is *reducible* with every state having a maximum in-degree and out-degree of 2. The alignment graph of the pattern against the input consists of $n+1$ copies of the automaton and is similar to that of fig. 11 with the row of states representing the fixed linear pattern being replaced by the automaton graph. The approximate matching solution is the result of applying a dynamic programming recurrence to the states of the alignment graph. This recurrence relation at each state is evaluated in two "topological" sweeps of each copy of the automaton. The time required is $O(mn)$. It permits arbitrary values to be specified for insertion, deletion and substitution costs for each pair of symbols.

When only a single match with the follows semantics is desired, the simulation procedure of [ASU86, fig. 3.31, page 126] can be adapted as follows:

S := ϵ-closure($\{s_0\}$);
a := nextchar;
while $a \neq$ **eof do begin**
    S := $S \cup \epsilon$-closure(move(S, a));
    a := nextchar;
    if $S \cap F \neq \phi$ **then**
        return yes;
    endif
end

Figure 12: A single match for a regular expr with the follows semantics.

The key difference between the two algorithms is shaded. Both follow the subset construction to form the set of all possible states the automaton can be in, after examining each symbol of the input. The difference in the above algorithm is that states can be reached on input α by ignoring some of its symbols. Therefore states that can be reached on input α must be a subset of the states reached from the input αx because x can be ignored by the automaton.

The simulation procedure of fig. 12, combined with the automaton construction procedure in [MM89] can be used to make the following optimizations for the single match case.

1. *Back edges* need not be taken. This optimization follows from the *reducibility* of the graph. Because any match, not necessarily the longest is desired, revisiting states is not advantageous.

2. If all the output states of a state **s** are in the set S, then state **s** can be removed from S. Consider two sets of states $\{s\}$ and $(S - \{s\})$. For any input, performing the **while** in fig. 12 on these sets, denoted WHILE($\{s\}$) and WHILE($S - \{s\}$), results in WHILE($S - \{s\}$) $\supseteq$ WHILE($\{s\}$) because every outgoing edge from **s** leads to states already in S.

The amount of savings resulting from these observations can only be determined through simulation.

If the regular expression fits the word length of the machine on which matching is done, matching may be done in linear time [WM91]. When several regular expressions are to be matched simultaneously, they can be combined to yield a single regular expression.

Matching partial orders, which is more complex than matching regular expressions, is considered next.

Observation 4: If all event symbols occur equally frequently in the pattern, matching fixed linear and regular expression patterns can be done in time $\approx \frac{O(mn)}{\sigma}$, where m is the size of the pattern, n that of the input, and σ is the number of distinct event symbols.

In the construction for matching regular expressions [obs. 3] above (a fixed linear pattern is a trivial regular expression), the maximum size of set S is m. Using the uniformity assumption, the next input event can match at most m/σ elements of S. Thus, exercising n input events takes time at most mn/σ. This gives all positions in the input that end a match with the pattern.

Matching Partial Orders

The following adaptation of [ASU86, fig. 3.31, page 126] can be used to match fixed partial orders. $I_0, \ldots, I_i$ are i initial states of the partial order. For simplicity, all transitions are assumed enabled. Otherwise, the enabling can be incorporated in the function move. The set S is initialized to the cross product of the ϵ-closure sets of each initial state. Thereafter, on each input symbol, S is augmented to include all ways of exercising each thread of the partial order. The pattern is matched if all threads reach the final state F. If S is a cross product, $S[k]$ is used to denote its kth element. The function "move" is as described in [ASU86].

$S := \{(\epsilon\text{-closure}(\{I_0\}) \times \ldots \times \epsilon\text{-closure}(\{I_i\}));$
$a :=$ nextchar;
while $a \neq$ **eof do begin**
    $\forall e \in S$, add **Move**(e, a) to S
    $a :=$ nextchar;
    if $\exists e \in S \mid \forall k, 1 \leq k \leq i, e[k] \cap F \neq \phi$ **then**
        return yes;
    endif
end

Move(State s, input a)
{
  return
    $(\epsilon\text{-closure}(\text{move}(s[1])) \times \cdots \times s[i]) \ldots \cup \ldots$
    $\vdots$
    $(s[1] \times \cdots \times \epsilon\text{-closure}(\text{move}(s[i])))$
}

Figure 13: A single match for a partial order with the follows semantics.

Similar to the optimizations mentioned for matching regular expressions, revisiting states may be avoided. However, this does not translate easily into skipping a set of edges in the automaton.

Each cross product represents a possible combination of the i threads of the partial order. All of them together represent all such possibilities. Since each input symbol may exercise any thread, the deterministic matcher must exercise each in turn. Thus each cross product results in i cross products being exercised. Starting with 1, the number of cross products added at each step are $i, i^2, i^3, \ldots$ Since each thread may take up to the size of the pattern to be exercised, the total time required is $mi + mi^2 + \cdots \approx mi^n$.

Observation 5: Fixed Partial Order Pattern matching with the *immediately follows* semantics can be done in time

$$O(N \times |Q|^{|I|})$$

where Q is the set of states, I is the set of initial states of the partial order and N is the size of the input. For example $|Q| = 11, |I| = 2$ in figure 8. In the discussion below, we use the term *NPFA* to denote the non deterministic finite state automaton that recognizes a fixed partial order pattern specification (like fig. 8) and *DPFA*, its corresponding deterministic finite state automaton.

This follows from a straightforward subset construction of the pattern as outlined in fig. 13. The difference in this case from the subset construction for *NFA*s is that there are several "threads" along which an input event can be matched, corresponding to each thread begun at its corresponding initial state of the partial order. Furthermore, non determinism is possible along any thread, just as in the case of an *NFA*. An external event can be discarded or match the next transition of *only one* thread, but nondeterminism prevents us from knowing which one. We must therefore exercise all in turn, and represent the composite transition information by a set of each individual thread transition. So each state of the *NPFA* matching the partial order looks like

$$\{\text{state of thread } 1, \ldots, \text{state of thread } i\}$$

Consider that, on exercising the *NPFA* with a particular string α, the *NPFA* reaches a state s. This state embodies the information of *one* possible way of matching α in the *NPFA*, and the set of states that can be reached on input α represents all possible ways of exercising the *NPFA* on input α. From any state s in the *NPFA*, a transition on the symbol a leads to at most $|Q|^{|I|}$ states, each such state corresponding to exercising the ith thread of the *NPFA*.

This *NPFA* can be converted into a *DPFA* by the straightforward subset construction of automata theory. The number of states possible for the *NPFA* is $|Q|^{|I|}$ and the number of states in the corresponding *DPFA*, $2^{|Q|^{|I|}}$. ∎

Matching With Unification

Claim 1: Pattern Matching with Unification is NP Hard.

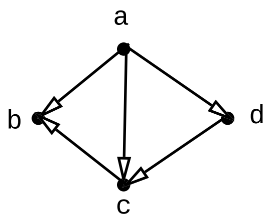
We show that a much weaker type of pattern, *linear pattern* with unification is NP complete. The reduction is from an arbitrary Hamiltonian circuit problem. Let the arbitrary graph whose Hamiltonian cycle must be determined have n vertices and e edges. Generate, from the graph description, another description of all its edges (the input events) such that the description of all the directed edges emanating from a vertex are written together. Let a be the node whose edge description is written first. Let each edge be written as `T(tail, head)`. `T` is a fictitious tuple type with two elements, and is simply used to cast the Hamiltonian cycle problem as a matching problem. This can be done in time $O(e)$. Generate the pattern to be matched against the edge description as input, in time $O(n)$ as follows:

$$\begin{array}{ll} T(a, X) & (1) \\ T(X, Y) & (2) \\ \vdots & \vdots \\ T(P, Q) & (n-1) \\ T(Q, a) & (n) \end{array}$$

Each line of the pattern represents an edge of the graph. The pattern represents a Hamiltonian cycle in the graph. When the pattern matches successfully, X is necessarily different from Y, Y from Z etc. because when successive elements in the pattern $T(X, Y)$ and $T(Y, Z)$ are matched, $T(Y, Z)$ must be matched from the group of edges emanating from Y. If Z is matched with any node previously matched then matching cannot proceed since the group of directed edges emanating from that node have been skipped in the input which cannot be rolled back.

This shows that Pattern Matching with Unification is NP hard. The problem is in NP because a polynomial time bounded oracle can "guess" the values of $X, Y, Z, \ldots$ to be matched.

For example, for the graph below



one possible edge description and its corresponding pattern are:

T(a, b)		
T(a, d)	→ edges emanating from *a*	
T(a, c)		T(a, X)
	→ no edges emanating from *b*	T(X, Y)
T(c, b)	→ edges emanating from *c*	T(Y, Z)
T(d, c)	→ edges emanating from *d*	T(Z, a)
	`Edge Description`	`Pattern`

Matching With Simplified Occurrence Constraints

Observation 6: Given a sequence of input and a *fixed linear pattern*, there is a polynomial time algorithm to decide whether the entire pattern can be detected in a certain amount of time.

As an example consider the pattern bca to be matched in 8 time units against the fixed sequence of input events:

Time:	1	3	4	7	11	12	19	23	25
Event:	a	b	a	c	b	c	a	b	a

Figure 14: A simple example of a time bounded pattern match.

With the "follows" semantics, to determine the minimum time of occurrence of the pattern bca starting at any b, determine the occurrence of the next c, followed by the occurrence of the next a. Thus, the minimum time of occurrence of bca in fig. 14 at $t = 3$ is 16, and that starting at $t = 11$ is 8.

and that is the minimum time for the occurrence of the pattern bca starting at that b. Therefore all we have to do is to compute this value for every b.

The minimum time of occurrence of the pattern bca, then, is

$$\min_b(\text{time of occurrence of } bca)$$

The worst case time of this algorithm is $O(n^2)$, given n events, because there are at most n potential instances of the pattern that can be matched, and each event might require to be matched against each instance.

Observation 7: Given a fixed sequence of input and a *fixed linear pattern*, there is an exponential time off-line algorithm to decide a pattern match when the maximum time between successive events is given.

Construction: Label all the input events of the same type with subscripts starting at 1 and increasing sequentially (see fig. 15). Construct a tree of the labeled events such that successive levels of the tree correspond to successive events in the pattern. In the pictorial representation of the input in fig. 16, edges are labeled with the corresponding event for illustration.

Pattern:	*abc*								
Time:	1	3	4	7	11	12	19	23	25
Event:	a	b	a	c	b	c	a	b	a
Label:	1	1	2	1	2	2	3	3	4

Figure 15: An example of a pattern with constraints on successive events.

For illustration, the entire tree is depicted, in reality only those edges satisfying the maximum time criterion between successive events are present in the real tree. The construction of the tree requires a worst case time of $n^{(m+1)}$ with n events and pattern size m because the depth of the tree is m and its branching factor is n in the worst case. All nodes at depth m correspond to matched patterns. The space requirement is exponential because the maximum number of possible matches is ${}^nC_m$, which is $< 2^n$.

Note: If, in addition to the maximum time between events, a maximum overall time is also specified, that can also be handled in a straightforward manner. This construction requires that the event sequence be prespecified.

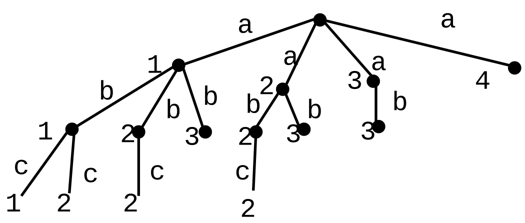


Figure 16: Converting the input to a tree for matching.

More recently [AA93] have shown much better bounds for the generalized version of this problem. Their solution requires $O(r \log n)$ time, where n is the number of input events and $r = \Sigma_a P(a) \times T(a)$, $P(a) = \#$ of occurrences of a in the pattern, and $T(a) = \#$ of occurrences of a in the input, which yields an off-line upper bound of $O(mn \log n)$.

Matching General Patterns

Having considered the complexity of matching fixed linear patterns, regular expressions, partial orders and matching with unification, we investigate some structures in the input that reduce the complexity of matching for the above mentioned problems as well as for the general case. The results of the previous section show that deterministic matching requires considerable expense in space and time for the conversion of non deterministic patterns to a deterministic recognizing automaton. This expense precludes their precompilation even though it is one time. Matching requiring all possible matches is much more expensive, in the general case, than when any match will do. Single match algorithms can be applied if all the matches are non overlapping. In this case the automaton can be restarted in its initial state, once it has reached the final state. Presented below are some structures in the input that might be useful in detecting particular types of intrusions.

Observation 8: An exhaustive search can be avoided when matching, if some or all of the guards in the pattern specification are monotonic.

As an example, consider figure 8 with the following guards:

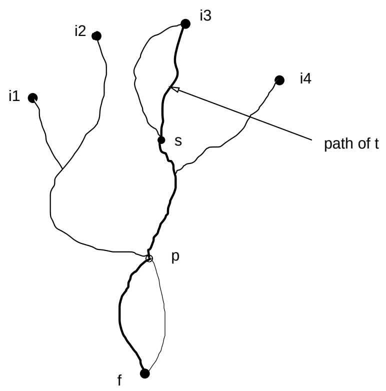
At the transition cutting the edge $\{11 \Rightarrow 10\}$: `T1 =` $\langle$`time`$\rangle$
At the transition cutting the edges $\{1 \Rightarrow 7, 8 \Rightarrow 7\}$: `T2 =` $\langle$`time`$\rangle$ `&&` $T2 - T1 \ \leq \ 5$

During matching, consider that there is a token t_1 at state 10, duplicated to t_2 in state 8 and awaiting merging with t_3 in state 1. Assume that the transition can never fire for the combination of tokens $\{t_2, t_3\}$ because the guard $T2 - T1 \geq 5$ for this pair cannot be satisfied. 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 failure of the guard $T2 - T1 \geq 5$. This conclusion was possible because the time stamps of successive events was non decreasing and the operator $\leq$ is monotonic. That is if $a \leq b$ is false then $a + x \leq b$ is false $\forall x \geq 0$. This observation is not applicable for non monotonic data fields.

How can this property be apply. Exactly what part of the search space can be excluded? This is formalized as claim 2 below:

Claim 2: During matching, whenever a monotonic data field d gets 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.

Consider the pattern



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) gets 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, will continue to result in its failure.

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

Claim 3: For the single match case, tokens can be *moved* instead of *duplicated* from states that lead to single input transitions which do not involve unification or make additional bindings to variables.

As a consequence of these conditions, no pattern variable associated with the token changes. Therefore, future duplications do not alter the token 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.

Observation 9: For the single match case, during matching, as soon as all the variables associated with the pattern have been instantiated for a token, that token need not be duplicated further.

This result holds if we are not concerned with finding all matches of the pattern in the input, but the first match will suffice. It follows from the fact that variables, once assigned, cannot be modified. Thus, whenever a token has all its variables bound to values, it becomes unique, and duplicating it simply makes identical copies of the token. Thus, if not all matches are desired, a single copy will do. Note, however, that this does not imply preservation of the number of those tokens, if the token is moved across a transition with several output arcs and one input arc, the number of tokens of that exact type will increase.

The current section discussed some theoretical results obtained in our model. These results focused on efficient matching a single pattern. The next section discusses how several patterns may be matched efficiently.

Observation 10: Matching general patterns, with the exclusion of partial orders, can be done in exponential time.

Given a general pattern of size m (including guards) and an input of size n, matching the pattern to the input with the "immediately follows" semantics can be done in time $O(mn)$ by a simulation of the non deterministic matching. There are an exponential number of choices of selecting a subsequence (not necessarily a substring) of n input events, yielding an overall time of $O(mn2^n) = O(c^n)$.

10 Appendix B: Matching Multiple Patterns

Matching Multiple Fixed Linear Patterns

Observation 11: Matching several fixed linear patterns with the *follows* semantics can be done in time $O(n \times \#\text{ of patterns})$

n is the size of the input. This follows from [obs. 1, pg. 35] by running the matching procedure for each pattern simultaneously. Compare it with [AC75] in which multiple fixed linear patterns can be matched with the *immediately follows* semantics in linear time. The primary reason for the difference is that in [AC75], the set of states maintained by the algorithm matches the set of states reachable on the input in the NFA; while matching with the follows semantics requires the maintenance of the set of states reachable on any prefix of the input with any characters deleted. Because of the way the automaton is constructed in the former, the set of states can be represented by a single state.

Several regular expression patterns, say $re_1, \ldots, re_m$ can be written as the regular expression $(re_1 | \ldots | re_m)$ and matched approximately in $O(mn)$ [page 36], where m is the total length of all the patterns. The optimizations mentioned in that approach are also applicable.

11 Appendix C: An Example of Subexpression Elimination in Guards

Consider, as an example, the following attacks (see fig. 5 for a description of their guards):

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

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)
```

There is considerable similarity between the sub-signatures `1a` & `2a`. For any event of type `LINK`, once one of them is evaluated, the other may need not be. The signatures may be compiled as show below. This is representative of the translation, and may be different in the final form, as the specification of the virtual machine instruction set and its semantics is continuing to evolve. The translation process is not discussed in this report and is not within the scope of this work.

<u>Compilation of 1a</u>

```
OBJ ← LINK
TRANSITION ← 4                    ;this transition is numbered 4 among all the transitions.
T1^1 ← OBJ[SRC_FILE]^1            ;indexing is a primitive, polymorphic operation.
FILE1^1 ← T1^1                    ;global variables are assigned only through temporaries. FILE1 and FILE2
T2^2 ← OBJ[DEST_FILE]^2           ;are variables global to the pattern.
FILE2^2 ← T2^2                    ;all temporary variables are named T<number>.
T3^3 ← OBJ[UID]^3
U^3 ← T3^3                        ;U is also global to the pattern.
T4^4 ← OWNER(FILE1^1)^4           ;owner is a built in function that returns the owner of a file.
IFM T4, U, EXIT                   ;if T4 matches with U, jump to EXIT.
T5^5 ← NAME(FILE1^1)^5            ;built in function giving the filename portion of a full path name.
IFM T5, ''-*'', EXIT              ;if T5 matches "-*" then jump to EXIT.
T6^6 ← SHELL_SCRIPT(FILE1^1)^6    ;built in function to test if a file is a shell script.
IFFALSE T6, EXIT                  ;if T6 is 0, jump to EXIT.
T7^7 ← FPERM(FILE1^1)^7           ;built in function giving the permissions of a file.
T8^8 ← AND T6^6, XGRP             ;XGRP is a constant used to determine if a file is group executable
IFFALSE T8^8, L1
RES ← 1                           ;signals a successful evaluation of the guard.
RETURN                            ;return from this guard.
L1:
T9^9 ← AND T6^6, XOTH             ;XOTH is a constant used to determine if a file is executable by others.
IFFALSE T9^9, L2
RES ← 1
```

```
RETURN
EXIT: L2:
RES ← 0      ;signals an unsuccessful evaluation of the guard.
RETURN
```

<u>Compilation of 2a</u>

```
OBJ ← LINK
TRANSITION ← 7                        ;this transition is numbered 7 among all the transitions
T10^10 ← OBJ[SRC_FILE]^10             ;Temporary variable numbers are not reset.
FILE1^10 ← T10^10
T11^11 ← OBJ[DEST_FILE]^11
FILE2^11 ← T11^11
T12^12 ← SHELL_SCRIPT(FILE1^10)^12
IFFALSE T12^12, EXIT
T13^13 ← FPERM(FILE1^10)^13
T14^14 ← AND T13^13, XGRP
IFFALSE T14^14, L3
RES ← 1
RETURN
L3:
T15^15 ← AND T13^13, XOTH
IFFALSE T15^15, L4
RES ← 1
RETURN
EXIT: L4:
RES ← 0
RETURN
```

The superscripted numbers in the instructions above correspond to their value numbers as outlined in [CS70]. The expression `OBJ[SRC_FILE]` is given a single value number because indexing is a primitive operation in our virtual machine. Each guard expression begins with an instruction of the form

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

The variable `OBJ` is special and is automatically instantiated to 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 `1a` and `2a` successfully. The other variables of special meaning are `RES`, whose value determines whether the guard has been evaluated successfully, and `TRANSITION`, which refers to the particular guard transition currently being compiled. This number is used to index into a vector of transitions in which each element denotes whether the corresponding transition fires.

Combining the set of compiled instructions from all transitions labeled with the same event type is non trivial. 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 very useful here as they are likely to be small, with little redundancy.

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 organized as a network (similar to Rete networks [For82]) to improve the running time of evaluation by taking into account its dynamic evaluation. When organizing a network, a good configuration needs to be determined as does the duplication and rearrangement of guards (perhaps based on historical statistics of their truthful evaluation).

The approach taken here is to combine the guards in a chain in an arbitrary order and perform elimination *across* basic blocks and guards by introducing the notion of active and inactive regions of code. This notion is similar to that commonly used by data driven SIMD architecture machines to force some of its processors to execute instructions, while the remaining to ignore the same, based on a vector mask to enable/disable each processor. For our virtual machine definition, some instructions are treated differently depending on the type of the region of code. An active region of code is the region executed when the virtual machine is enabled. The rest of the code is part of the inactive region. It cannot be determined statically because the evaluation of conditional expressions influences its boundaries. Expressions evaluated in an inactive region are termed `inactive`. Lack of loops and jumps in the guard expressions enable its translation to have forward jumps only. The virtual machine executing the composite code treats (un)conditional jumps specially. Instead of jumping to the specified label, it stores its address and disables itself by setting its condition code register. When the processor is disabled certain types of instructions are not evaluated by it. Because all jumps are forward, the machine is enabled correctly when the jump address is reached, at which point it resumes its normal operation and evaluates every instruction it encounters.

This artifice ensures that all expressions will *always* be evaluated, and therefore, be available to expressions evaluated later. This regardless of whether the 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 [CS70], the code for *both* the guard expressions looks as shown below.

```
 1.  OBJ ← LINK
 2.  TRANSITION ← 4
 2'. if(!ENABLED_TRANSITIONS[TRANSITION]){
         set processor state disabled
         JUMP 28                          ;has no effect since processor state is disabled
     }
 3.  T1^1 ← OBJ[SRC_FILE]^1
 4.  FILE1^1 ← T1 ^1                      ;assignment to global variables has no effect when
 5.  T2^2 ← OBJ[DEST_FILE]^2              ;the processor is disabled
 6.  FILE2^2 ← T2^2
 7.  T3^3 ← OBJ[UID]^3
 8.  U^3 ← T3^3
 9.  T4^4 ← OWNER(T1^1)^4
10.  IFM T4, U, EXIT                      ;conditional jumps have no effect when the processor
```

```
11.  T5^5 ← NAME(T1^1)^5                       ;state is disabled
12.  IFM T5, "-*", EXIT                        ;if T5 matches "-*" then jump to EXIT
13.  T6^6 ← SHELL_SCRIPT(T1^1)^6
14.  IFFALSE T6, EXIT
15.  T7^7 ← FPERM(T1^1)^7
16.  T8^8 ← AND T7^7, XGRP
17.  IFFALSE T8^8, 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,
20.  L1:                                       ;disable it and continue. A JUMP has no effect otherwise.
21.  T9^9 ← AND T7^7, XOTH
22.  IFFALSE T9^9, 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.  OBJ ← 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 ← OBJ[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 ← OBJ[DEST_FILE]^11                ;compiled away. same value as T2.
33.  FILE2^11 ← T11^11                         ;not compiled away
34.  T12^12 ← SHELL_SCRIPT(T10^10)^12          ;compiled away. same value as T6.
35.  IFFALSE T12^12, EXIT                      ;T6 value propagated to T12.
36.  T13^13 ← FPERM(T10^10)^13                 ;compiled away. same value as T7.
37.  T14^14 ← AND T13^13, XGRP                 ;compiled away. same value as T8.
38.  IFFALSE T14^14, L3                        ;T8 value propagated to T14.
39.  FIRABLE_TRANSITIONS[TRANSITION] ← 1
40.  JUMP next_pattern                         ;instead of RETURN.
41.  L3:
42.  T15^15 ← AND T13^13, XOTH                 ;compiled away. same value as T9.
43.  IFFALSE T15^15, L4                        ;T9 value propagated to T15.
44.  FIRABLE_TRANSITIONS[TRANSITION] ← 1
45.  JUMP next_pattern                         ;instead of RETURN.
46.  EXIT: L4:
```

```
47.  FIRABLE_TRANSITIONS[TRANSITION] ← 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, would asymptotically result in a reduction of 5 statements out of 21, which tends to $\approx 25\%$. 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 run time 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, in order to determine whether the tokens in the initial states of figures 5 and 7 need to be duplicated and moved across to the succeeding state, we need to evaluate the code for every audit record of type `LINK`.

This leads to the question of the efficiency of this approach. For it is possible that only one of the guards 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 its amount of is dependent on the type of guard expressions and the commonality between them. Because of the nature of the matching process, transitions, enabled once, continue to remain enabled. Thus if a guard is evaluated once, it is likely to be evaluated from then on. The approach presented here provides a mechanism, which can be used profitably given the right set of signatures. An actual system might incorporate both types of approaches, with and without subexpression elimination and, based on heuristics and run time 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.

1. Pattern variable values referenced at a guard but set outside it are not value propagated across patterns, 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*. As a consequence, dead variables can be detected by simply examining the *rest of the code*.

For a good treatment of these and other compiler optimization issues, see [ASU86, FL88].

References

[AA93] Mikhail J. Atallah and Alberto Apostolico. (personal communication), 1993.

[AC75] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18:333–340, June 1975.

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

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

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

[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.

[BM77] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):262–272, 1977.

[Bos88] Computer whiz puts virus in computers. *Boston Herald*, page 1, Nov 5 1988.

[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, MA, June 1989.

[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.* PhD thesis, 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.

[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.

[Den82] Dorothy E. Denning. *Cryptography and Data Security.* Springer-Verlag, London, 1982.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. In *IEEE Trans. on Software Engg.*, number 2, page 222, Feb 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, University of California, Davis, Dept. of Computer Science, 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.

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

[FS91] Daniel Farmer and Eugene Spafford. The COPS Security Checker System. Technical Report CSD-TR-993, Purdue University, Department of Computer Sciences, September 1991.

[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.

[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, University of New Mexico, Department of Computer Science, August 1990.

[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.

[Jac86] Peter Jackson. *Introduction to Expert Systems*. International Computer Science Series. Addison Wesley, 1986.

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

[JLA+93] R. Jagannathan, Teresa Lunt, Debra Anderson, Chris Dodd, Fred Gilham, Caveh Jalali, Hal Javitz, Peter Neumann, Ann Tamaru, and Alfonso Valdes. System Design Document: Next-Generation Intrusion Detection Expert System (NIDES). Technical Report A007/A008/A009/A011/A012/A014, SRI International, March 1993.

[KMP77] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Computing*, 6(2):323–350, 1977.

[Lan92] Linda Lankewicz. *A Non-Parametric Pattern recognition to Anomaly Detection*. PhD thesis, Tulane University, Dept. of Computer Science, 1992.

[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.

[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. Technical report, SRI Computer Science Laboratory, SRI International, Menlo Park, CA, 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.

[LV92] G. E. Liepins and H. S. Vaccaro. Intrusion Detection: Its Role and Validation. *Computers & Security*, pages 345–355, 1992.

[Man89] Udi Manber. *Introduction to Algorithms : A Creative Approach.* Addison-Wesley, Reading, Mass., 1989.

[Mar88] J. Markoff. Author of computer 'virus' is son of U.S. electronic security expert. *New York Times*, page A1, Nov 5 1988.

[Met87] S. J. Metz. Computer break-ins. *Communications of the ACM*, 30(7):584, July 1987.

[MM89] Eugene W. Myers and Webb Miller. Approximate Matching of Regular Expressions. In *Bull. Math. Biol.*, volume 51, pages 5–37, 1989.

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

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

[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.

[Rei87] Brian Reid. Reflections on Some Recent Widespread Computer Break-Ins. *Communications of the ACM*, 30(2):103–105, February 1987.

[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. IEEE, IEEE Service Center, Piscataway, NJ, May 1991.

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

[Sma92] Steve Smaha. Questions about CMAD. Proceedings of the Workshop on Future Directions in Computer Misuse and Anomaly Detection, March 1992.

[Spa88] Eugene Spafford. The Internet Worm Program: An Analysis. Technical Report CSD-TR-823, Department of Computer Sciences, Purdue University, West Lafayette, IN, November 1988.

[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 *Proc. Fourth Workshop on Computer Security Incident Handling*, Denver, CO, 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*. USENIX Association, 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.

[TCL90] Henry S. Teng, Kaihu Chen, and Stephen C Lu. Security Audit Trail Analysis Using Inductively Generated Predictive Rules. In *Proceedings of the 6th Conference on Artificial Intelligence Applications*, pages 24–29. IEEE, IEEE Service Center, Piscataway, NJ, March 1990.

[WB90] Winfried R.E. Weiss and Adalbert Baur. Analysis of Audit and Protocol Data Using Methods from Artificial Intelligence. In *Proceedings of the 13th National Computer Security Conference*, October 1990.

[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, 3rd edition, 1992.

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

[WS78] Robert A. Wagner and Joel L. Seiferas. Correcting Counter-Automaton Recognizable Languages. In *SIAM J. Computing*, volume 7, pages 357–375, August 1978.