A PATTERN MATCHING MODEL FOR MISUSE INTRUSION DETECTION*

Sandeep Kumar Eugene H. Spafford

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

Keywords: intrusion detection, misuse, anomaly.

Abstract

This paper describes a generic model of matching that can be usefully applied to misuse intrusion detection. The model is based on Colored Petri Nets. Guards define the context in which signatures are matched. The notion of start and final states, and paths between them define the set of event sequences matched by the net. Partial order matching can also be specified in this model. The main benefits of the model are its generality, portability and flexibility.

1 Introduction

Computer break-ins are becoming increasingly frequent and their detection is increasingly important. Break-ins make the data residing on computer systems vulnerable to theft and corruption. Compromised sites can also be used to launch further attacks, thus achieving another level of indirection for further break-ins. A majority of break-ins, however, are the result of a small number of known attacks, as evidenced by reports from response teams (e.g. CERT). Automating detection of these attacks should therefore result in the detection of a significant number of break-in attempts.

Intrusion Detection is primarily concerned with the detection of illegal activities and acquisitions of privileges that cannot be detected with information flow and access control models. Examples of these include software engineering flaws in programs that allow cross privilege domain executions, insider abuse and failure of authentication procedures. Intrusion Detection models therefore do not directly overlap with traditional security models [11] which are primarily concerned with modeling information flow in a computer system to ensure that subjects are never able to access unauthorized information, or with modeling access control mechansims to prevent unauthorized access to objects.

Current approaches to detecting intrusions can be broadly classified into two categories: `Anomaly Detection` and `Misuse Detection` [20]. Anomaly Detection is based on the premise that intrusive activity often manifests itself as an abnormality. The usual approach here is to devise metrics indicative of intrusive activity, and detect statistically large variances on these metrics. Examples might be an unusually high number of network connections within an interval of time, unusually

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

high CPU activity, or use of peripheral devices not normally used. This approach has been studied extensively and implemented in a large number of systems [19, 18, 12, 14, 5, 8]. It attempts to quantify the acceptable behavior and thus identify abnormal behavior as intrusive.

The other technique of detecting intrusions, misuse detection, attempts to encode knowledge about attacks as well defined patterns and monitors for the occurrence of these patterns. For example, exploitation of the `fingerd` and `sendmail` bugs used in the Internet Worm attack [21] would be in this category. This technique specifically represents knowledge about unacceptable behavior and attempts to detect its occurrence.

This paper proposes a variation of one approach to misuse detection, state transition analysis, by using pattern matching to detect system attacks. Knowledge about attacks is represented as specialized graphs. These graphs are an adaptation of Colored Petri Nets [9] with guards representing signature context and vertices representing system states. The graph represents the transition of system states along paths that lead to intruded states. Patterns may have user-specifiable actions associated with them that are executed when patterns are matched. The model provides the ability to specify partial orders and subsumes matching of sequences and regular expressions. Patterns also have pre- and post-conditions associated with them that must be satisfied before and after the match. Patterns also may include invariants to specify that a condition is or is not satisfied while the pattern is being matched.

There are several benefits to our approach of using a generic model of matching. A significant benefit is the clean separation of the various components comprising a generic misuse detector. With our approach to designing a generic misuse detector, it can be viewed as three basic abstractions. This enables generic solutions to be substituted for each abstraction without changing the interfaces between the abstractions of the model. These abstractions are:

- The Information Layer. This encapsulates the audit trail and provides a low-level data interface to the monitored computer system.
- The Signature Layer. This provides for a system-independent internal representation of signatures and a system-independent virtual machine to represent the signature context.
- The Matching Engine. This encapsulates the method used to match the patterns. It makes the system independent of any particular choice of matching algorithms. It also allows simple substitution of newer or more powerful mechanisms as they become available.

Furthermore, a standardization of the model of matching signatures permits several external representations of signatures to exist, each facilitating the representation of certain type of signature constructs. Other benefits of the model include its extensibility and portability to different event models; its ability to assign priority to signatures and the ability to dynamically add signatures in the midst of matching [sec. 4].

Our model is generic and does not assume any characteristics of the underlying events against which matching is done or the domain of solution. It provides a mechanism on which matching solutions can be built. For example, the same model applies for the case of monitoring network packet flow, or for monitoring specific patterns in logs generated by general purpose logging utilities. The input events in any of these problem domains can be canonicalized and used as input to our model of matching. Specific instantiations can be made of the model as appropriate to the problem. For example, a specialized version could be created for matching intrusion signatures in the context of UNIX audit trails.

2 Primary Approaches To Misuse Detection

Misuse detection might be implemented by one the following techniques:

1. `Expert Systems`, which code knowledge about attacks as `if-then` implication rules.
2. `Model Based Reasoning Systems`, which combine models of misuse with evidential reasoning to support conclusions about the occurrence of a misuse.
3. `State Transition Analysis`, which represents attacks as a sequence of state transitions of the monitored system [16, 6].
4. `Key Stroke Monitoring`, which uses user key strokes to determine the occurrence of an attack.

These methods are summarized in the following sections.

2.1 Expert Systems

An expert system is defined in [7] as a computing system capable of representing and reasoning about some knowledge-rich domain with a view to solving problems and giving advice. Expert system detectors code knowledge about attacks as `if-then` implication rules. Rules 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 of the rule are performed which may trigger the firing of more rules or conclude the occurrence of an intrusion. The main advantage in formulating `if-then` implication rules is the separation of control reasoning from the formulation of the problem solution. Its chief use in misuse detection is to symbolically deduce the occurrence of an intrusion based on the available data.

The primary disadvantage of using expert systems is that working memory elements (the fact base) that match the left sides of productions to determine eligible rules for firing are essentially sequence-less. It is difficult to efficiently specify an order in which to match facts within the natural framework of expert system shells.[1] Other problems include software engineering concerns with the maintenance of the knowledge base [13] and the quality of the rules, which can be only as good as the human devising them [13].

2.2 Model Based Systems

This approach was proposed in [4] and is a variation on misuse intrusion detection. It combines models of misuse with evidential reasoning to support conclusions about its occurrence. There is a database of attack scenarios, where each scenario comprises a sequence of behaviors making up the attack. At any moment the system is considering a subset of these attack scenarios as likely ones being experienced by the system. 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 easily recognized in the audit trail, and must have a high likelihood of appearing in the behavior.

As evidence for some scenarios accumulates, and decreases for others, the active models list is up-

[1] 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 [3] procedure very inefficient.

dated. The evidential reasoning calculus built into the system permits the update of the likelihood of occurrence of the attack scenarios in the active models list.

The advantage of model based intrusion detection is its basis in a mathematically sound theory of reasoning in the presence of uncertainty. The structuring of the *planner* provides independence of representation of the underlying audit trail syntax. Furthermore, this approach has the potential of reducing substantial amounts of processing per audit record. It would do this by monitoring for coarser-grained events in the passive mode and then actively monitoring finer-grained events when those events are detected.

The disadvantage of model based intrusion detection is that it 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 intrusion model. It is also 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, but a consideration for successful implementation.

2.3 State Transition Analysis

In this approach [16, 6] attacks are represented as a sequence of state transitions of the monitored system. States in the attack pattern correspond to system states and have Boolean assertions associated with them that must be satisfied to transit to that state. Successive states are connected by arcs that represent the events/conditions required for changing state. These conditions, or signature actions, are not limited to a single audit trail event, but may be a complex specification of conditions.

2.4 Keystroke Monitoring

This technique uses user keystrokes to determine the occurrence of an attack. The primary means is to pattern match for specific keystroke sequences indicative of an attack. The disadvantages of this approach are the general unavailability of user typed keystrokes and the myriad ways of expressing the same attack at the keystroke level. Furthermore, without a semantic analysis of the contents, aliases can easily defeat this technique.

2.5 Summary Characterizing These Four Approaches

All four approaches to misuse detection encode and look for specific attacks and use matching in some form to detect them. If an attack is regarded as a set of steps, expert system rules permit the encoding of sequentiality (and other dependencies) between the steps. However, because of the generality of the match procedure of ascertaining firable rules, such dependencies are inefficient to match directly. Model based systems consider 'models' of intrusion and seek to verify them by looking for evidence to corroborate the model. This is done by using matching techniques on the underlying event trail. State transition approaches can be construed as trying to match the sequence of steps that lead a system to a compromised state. Each step in this sequence may, however, require complex computation for determining its occurrence (typically using expert system rules). Key stroke monitoring is the direct application of pattern matching to key stroke logs to match for suspicious or undesirable patterns.

2.6 Benefits And Limitations Of Misuse Detection

A primary disadvantage of anomaly detection, the other major technique for intrusion detection, is that statistical measures of user behavior can be gradually trained. Miscreants who know that they are being monitored can train such systems over a length of time to the point where intrusive behavior is considered normal. Misuse detection is immune to such training: if the signature for an attack is carefully written, even major variations of the same basic attack scenario can be detected. Moreover, the technique is simpler than anomaly detection. Within the framework of misuse signatures, monitoring of system activity can be automated as well.

The primary disadvantage of this approach is that it looks only for *known* vulnerabilities, and is of little use in detecting unknown future intrusions. However, we can look for known patterns of abuse that might occur after a vulnerability is exploited; although the intrusion itself may not be noted, the subsequent actions could be flagged.

3 Intrusion Detection Using Pattern Matching

Our pattern matching is based on the notion of an *event*. Events are auditable changes in the state of the system, or changes in the state of some part of the system. An event can represent a single action by a user or system, or it can represent a series of actions resulting in a single, observable record.

We further specify events as having tags. Generally, monitored events are tagged with data. In particular, the time 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.

A fundamental requirement of applying pattern matching to intrusion detection is that matching be done with *follows* semantics rather than *immediately follows* semantics. For example, with *follows* semantics the pattern ab specifies the occurrence of the event a `followed` by the occurrence of event b. It does not represent a immediately followed by b with no intervening event. This means that any two adjacent sub patterns within a pattern are implicitly separated by an arbitrary number (possibly zero) of events of any type. This assumption is appropriate in current systems: audit trail generation and modern user interfaces allow users to login simultaneously through several windows thereby generating overlapped entries in the audit trail.

Using *follows* semantics makes the field of discrete approximate pattern matching relevant to intrusion detection. Three characteristics determine the kinds of theoretical bounds that can be placed on the matching solution: `1)` whether matching is off-line or online `2)` whether signatures can be dynamically added or deleted as matching proceeds and `3)` whether all matches of the pattern in the event stream are desired or whether finding a single match is sufficient.

Results in approximately matching various classes of patterns are summarized in fig. 1. These time bounds hold for arbitrary values of deletion, insertion and mismatch costs, and are not optimized for the requirements of misuse intrusion detection. The results are restricted to online matching because we are primarily concerned with real time intrusion detection. *RE* stands for regular expressions and *sequence* refers to a chain of events. The column `match` denotes the type of match determined by the corresponding algorithm. An entry of “all endpts” denotes that the algorithm

Pattern	Time	Space	Preproc	Match	Reference	Comment
Sequence	$O(mn)$	$O(m)$	$O(1)$	all endpts	[22]	Using dynamic programming.
Sequence	$O(mn)$	$O(mn)$	$O(1)$	all	[22]	Using dynamic programming[a] .
Sequence	$O(n)$	$O(m)$	$O(1)$	all endpts	[1, 23]	Pattern fits within a word of the computer. Small integer values of costs.
RE	$O(mn)$	$O(m)$	$O(m)$	all endpts	[15]	Using dynamic programming.
RE	$O(mn)$	$O(mn)$	$O(m)$	all	[15]	Using dynamic programming[a].

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

Figure 1: Some Results from Pattern Matching Applicable to Misuse Detection

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. Finding all matches of a pattern in the input is an all-paths source-to-sink problem and is computationally hard.

While approximate pattern matching is useful in misuse detection, the general problem 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.

After studying common numerous UNIX vulnerability descriptions from such sources as the CERT security advisories, and those detected by the COPS [2] and TIGER [17] tools, we noted a temporally-related partitioning. We were able to classify intrusion attacks on UNIX as follows:

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. Examples include searching for altered permissions or certain special files.
2. **Sequence.** The fact that several things happened in strict sequence is sufficient to specify the intrusion.
3. **Partial order.** Several events are defined in a partial order, for example as in fig. 2.
4. **Duration.** This requires that something(s) existed or happened for not more than nor 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.

We believe that the vast majority of known intrusion patterns fall into categories 1 and 2. This classification is not strictly a hierarchy as characteristics simple to match at lower levels of the classification become intractable at upper levels. These classes can also be further subdivided into finer categories; details can be found in [10].

3.1 An Overview of Our Model of Matching

We examined various regular methods of representing and matching our attack signatures. Regular expressions can represent only the simplest types of attacks. Context-free and attribute grammars are not easy to extend to approximate matching and do not lend themselves well to a graphical

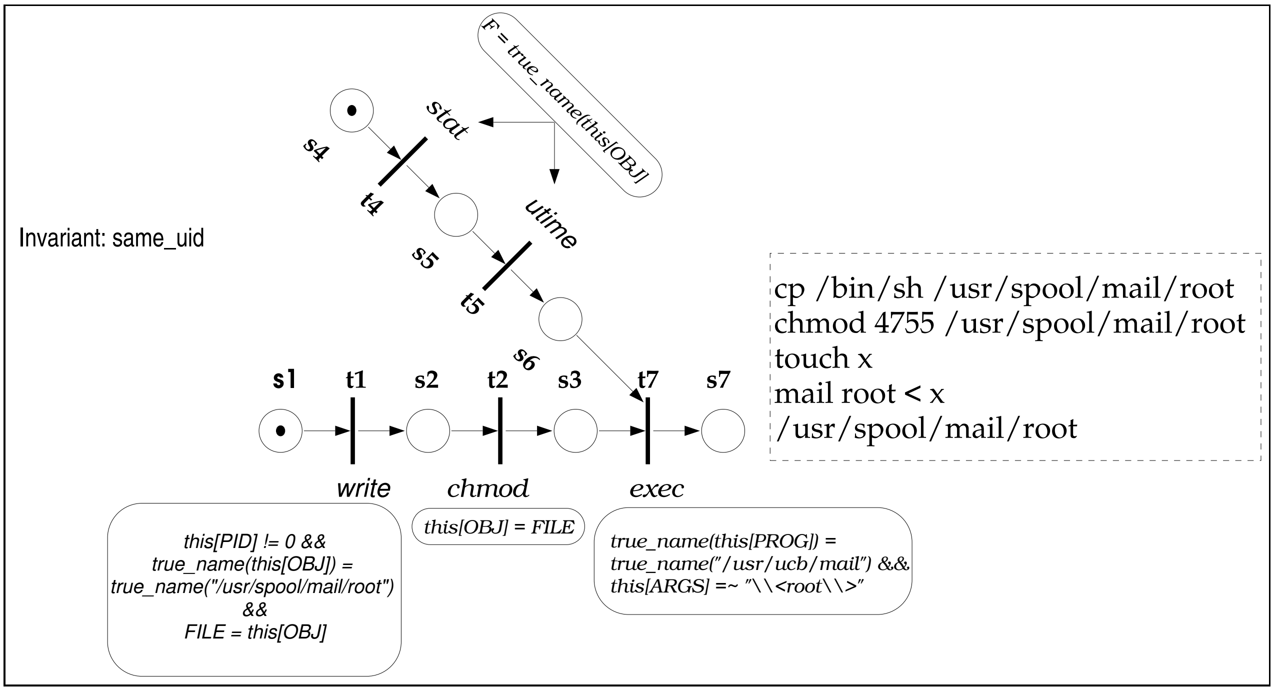

Figure 2: Representing a Partial Order of Events

representation. Regular expressions and context-free grammars do not permit matching to be conditional on the value of specified expressions. Attribute grammars allow conditional matching only in an indirect way. We settled on basing our model of matching on an extension of Colored Petri Nets [9] as they suffer none of these problems.

We refer to each signature represented as an instantiation of a Colored Petri Automaton (CPA). The notion of one or more start states and a unique final state defines the set of strings matched by the CPA. Matching begins with one token in each initial state. The pattern is considered matched for each token that reaches the final state. Along the path to the final state tokens can merge or be duplicated. Partial orders can be written with each trunk of the partial order starting at a different start state. Tokens that are merged carry the merge information with them so the entire merge path is stored.

Patterns are internally stored for matching as CPAs. 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. The main differences between our model and CP-Nets are the lack of concurrency in our model, absence of local transition variables, the notion of start and final states, and the notion of pre- and post-conditions and invariants associated with patterns. Moreover, nets in our model are not bipartite, unlike CP-Nets.

Our model is generic and applicable to any well-defined format of input events such as audit trail records, network packets, or other abstractions. Our examples here, however, are taken from the domain of misuse detection in the UNIX environment using audit trails as input.

Consider, as an example, the attack scenario in figure 2 [6]. Its CPA is translated verbatim from the attack scenario for purposes of illustration only. `s1` and `s4` are the initial states of the CPA, and `s7` is its final state. A CPA requires the specification of ≥ 1 initial states (each initial state represents a trunk of the partial order) and exactly one 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.

Each state may contain an arbitrary number of tokens.

A CPA also has associated with it a set of variables. Assignment to these variables is equivalent to unification. Each token maintains its own local copy of these variables because each token can make its own variable "bindings" as it flows to the final state. In CP-Net terminology, each token is colored, and its color can be thought of as an n-tuple of strings, where the pattern has n variables.

The CPA also contains a set of directed arcs that connect states to other states and transitions. The arcs which connect places to other places are ϵ transitions along which tokens flow nondeterministically without being triggered by an event. Each transition is associated with an event type, called its label, which must occur in the input event stream before the transition will fire. In fig. 2 transition `t1` is labeled with the event *write*, `t4` is labeled with the event *stat* and so on. Nondeterminism can be specified by labeling more than one outgoing transition of a state with the same label. There is, however, no concurrency in a CPA: an event can fire at most one transition. A transition is said to be enabled if all its input states contain at least one token.

Optional expressions, or guards, can be placed at transitions. These expressions permit assignment to the CPA variables. Example of these assignments include assignment of values to matched event fields; evaluation of conditions involving equality, $<$, or $>$; and calling built-in and user defined functions. Guards are Boolean expressions which evaluate to *true* or *false*. `this` is a special operator which is instantiated to the most recent event. It may be empty in the case of ϵ transitions. It provides a hook into the event matched at a particular transition. Guards are evaluated in the context of the event which matches the transition label and the set of *consistent* tokens which enable the transition. Tokens are consistent when their variable bindings unify. The set of tokens are unified before being passed to the guards for evaluation.

For example, in order for transition `t7` to fire, there must be at least 1 token in each of states `s3` and `s6`; the enabling pair of tokens (one from `s3`, the other from `s6`) must have consistently bound (unifiable) pattern variables; and the unified token and the event of type `exec` together must satisfy the guard at `t7`. 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 merged token are placed in each output place of the transition.

The process of merging resolves conflicts in bindings (i.e. makes sure that token bindings unify) between 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 it encountered on its path to the current state. The sequence of events matched by a CPA is the sequence of events (or partial order) encountered at each transition by the token that has reached the final state.

A CPA is also associated with a pre-condition, a post-condition, and an invariant expression. These are similar to guards that must evaluate *true* to be successful. Patterns that have no transitions (e.g., verifying that root's `.rhosts` file is not world writable) can be specified using pre-conditions to an empty pattern. Post-conditions are provided for symmetry and to allow the recursive invocation of the same pattern.

The reason for having invariants associated with CPAs is more subtle. It seems syntactically inconsistent to us to specify as part of patterns that they *must not* occur while another pattern is being matched. That is, negative pattern specification in a CPA unnecessarily clutters the description of the pattern. The other reason is that the semantics of some invariants cannot be easily absorbed in the framework of transitions and guard expressions. It is more efficient to provide them as primitives in the matching model than to attempt to subsume them within the framework of matching.

As mentioned earlier, our model of matching is generic. It can easily be instantiated for misuse detection for a system running the UNIX environment, for example. This would involve defining the primitives supported in guard expressions. It might also include coding file test operations, set manipulation functions, system interaction hooks, and other operations. The set of invariant primitives supported in the instantiated model must also be defined. The overall structure of matching remains unchanged.

4 Analysis Of Our Matching Model

There are several difficulties in intrusion detection using pattern matching. The dominant one is the sheer rate at which the data generated by modern processors must be matched. We have some confidence that a system as described in this paper can operate at a speed sufficient to operate in near real time. Furthermore, because state is saved in the tokens and their tag fields, there is no need to save (or re-process) extensive logs of the system.

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. This can require substantial overhead in time and space to track 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. In other cases the signature may specify arbitrary permutations of sub-patterns comprising the pattern thus making the recognition problem much more difficult.

The complexity of matching in our model increases rapidly with increasing complexity of signatures. At the simplest end are patterns without guards, for which algorithms from discrete approximate matching are applicable [fig. 1]. The introduction of guards and variables makes the complexity of the matching problem exponential in the size of the CPA if the description of guards is included in its size. Partial order matching takes super-exponential time. Matching can be improved in some cases by exploiting the monotonic nature of event fields, like the time stamp of the event. Evaluating guards can be optimized by defining a virtual machine for their evaluation. By breaking the guard expressions into sequences of simpler instructions, common subexpression elimination can be performed to reduce the size of the sequence. Such elimination can be done across all the patterns. All these results and optimizations are described in [10].

Our model has several important advantages. It is very portable, in the sense that intrusion signatures can be moved across sites without rewriting to accommodate fine differences in each vendor's implementation. Signatures can also be transparently moved to systems with somewhat different policies and ratings. An abstract audit record definition and a standard definition of a virtual machine to represent guards ensures that patterns pre-compiled to an intermediate representation can be moved across systems with minimal overhead.

Signatures can be dynamically added in the matching engine while maintaining the partial matches of signatures already present in it. The only disadvantage of doing this is that some optimizations, like common subexpression elimination of guards, may not be done for subsequently added patterns with respect to patterns already compiled in the engine. Actions can also be associated with patterns by incorporating them as expressions in the post conditions.

Signatures can be prioritized by considering each token as a thread of control. Each thread then fetches events from an event manager and acts on them. By prioritizing certain threads, patterns can be prioritized for matching.

5 Conclusions

The paper outlined a pattern matching approach to misuse intrusion detection. It proposed a generic model of matching based on CP-Nets that can be adapted to different problem domains. We used misuse detection using audit trails under UNIX as an example to illustrate the usefulness and applicability of this approach.

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 will implement this model and derive experimental results in the near future. Comparative performance results with other approaches will be difficult because of the lack of standardized benchmarking vulnerabilities and the unavailabilty of such data for other approaches. We hope that our prototype implementation and benchmarking results will provide the necessary first step in this direction.

References

[1] 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.

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

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

[4] 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.

[5] 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.

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

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

[8] 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.

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

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

[11] Carl E. Landwehr. Formal Models for Computer Security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[12] 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.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Stephen E. Smaha. Tools For Misuse Detection. In *Proceedings of ISSA '93*, Crystal City, VA, April 1993.

[21] Eugene Spafford. The Internet Worm Report. Technical Report 823, Purdue University, February 1990.

[22] 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.

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