# Planning, Petri Nets, and Intrusion Detection

Yuan Ho Deborah Frincke Donald Tobin, Jr

Center for Secure and Dependable Software
Department of Computer Science
University of Idaho
Moscow, ID 83844-1010

**Abstract**

Detection of intrusions with multiple sources and intrusions where incomplete behavioral data is available is a difficult task. We propose a new intrusion detection architecture combining partial order planning and executable Petri Nets to detect such attacks. Partial Order State Transition Analysis Technique, or POSTAT, increases the flexibility of the traditional state analysis approach by allowing unordered events in the signature action sequence.

## 1 Introduction

Detection of intrusions with multiple sources and intrusions where only partial behavioral data is available is a difficult task. We propose a new intrusion detection architecture combining partial order planning and executable Petri Nets to detect such attacks. This architecture is embodied in the Partial Order State Transition Analysis Technique, or POSTAT.

Our architecture includes a planning agent, a searching agent, and a site security informing agent (Figure 1). The planning agent constructs intrusion scenarios using a first-order logic description of the known activities and goals of the intruder to spcify an attack sequence. The searching agent takes a Petri Net representation of these intrusion scenarios and uses them to determine whether any of the specified intrusions are in progress. The informing agent processes final information and provides a user interface for the system.

In the remainder of this paper, we describe the POSTAT system and how it differs from traditional state transition analysis in intrusion detection. We provide examples of how our system may be used for both misuse and anomaly detection. Finally, we explain how our partial order planner constructions compact intrusion scenarios for the detection process and explain the role of Petri Nets.

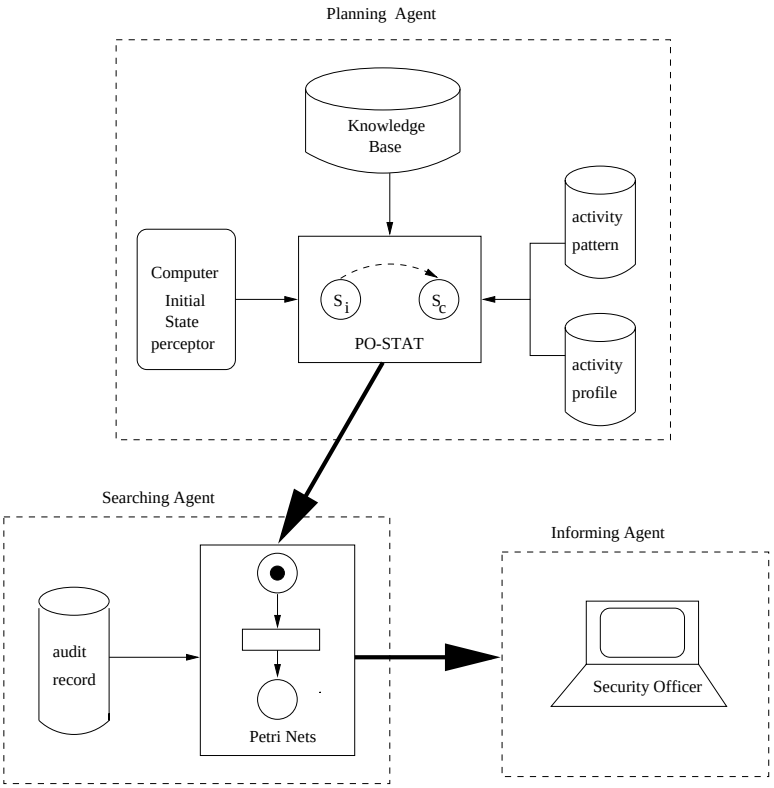


Figure 1: Organization of POSTAT's intrusion detection components

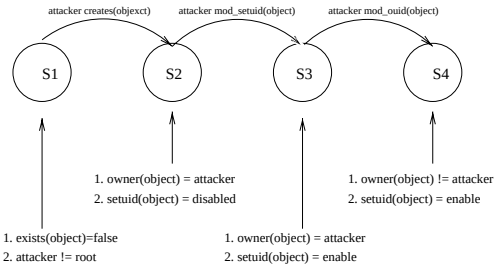


Figure 2: State transition diagram of misuse penetration example

| Step | Command | Comment |
|---|---|---|
| 1. | %cp /bin/csh /usr/spool/mail/root | - assumes no root mail file |
| 2. | %chmod 4755 /usr/spool/mail/root | - make setuid file |
| 3. | %touch x | - create empty file |
| 4. | %mail root < x | - mail root empty file |
| 5. | %/usr/spool/mail/root | - execute setuid-to-root shell |

Figure 3: Penetration scenario to gain root privilege

# 2 Partial Order State Transition Analysis

POSTAT uses partial order state transition analysis rather than traditional state transition analysis. In traditional state transition analysis, 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 or conditions required for changing state. State transition diagrams correspond to the states of an actual computer system, and these diagrams form the basis of a rule-based expert system for detecting penetrations. Ilgun presents an example of this approach in [IK95]. State transition considers a penetration to be a sequence of signature actions (SAs) performed by an attacker that lead from some initial state to a target compromised state [IK95]. The initial state corresponds to the state of the system just prior to the execution of the penetration, and the compromised state corresponds to the state of the system resulting from the completion of the penetration. Between the initial and compromised states lie intermediate state transitions that occur during an attacker's attempt to achieve the compromise. Signature actions are those that, if omitted from the execution of an attack scenario, would prevent the attack from completing successfully [IK95]. Figure 2 shows an example of an attack using three signature actions which can move the system from an initial safe state into a compromised state. Figure 3 shows the actual attack sequence that once could be used to acquire root privilege [IK95].

Traditional state transition analysis attack patterns specify a strictly ordered and non-overlapping sequence of events. To increase the flexibility of the state transition analysis approach, we use a partial ordering of events. A partial order of events specifies that some events are ordered with respect to each other while others are unordered. A partial order state transition analysis allows more than one sequence of events in the state transition diagram. By using partial ordering rather than total ordering in our state transitions, we are able to use a single diagram to represent the set of all intrusion attempts which involve the same signature actions.

Our partial ordered state transition diagrams are generated using partial ordered planning techniques (Section 4). A partial order plan representation is more powerful than a

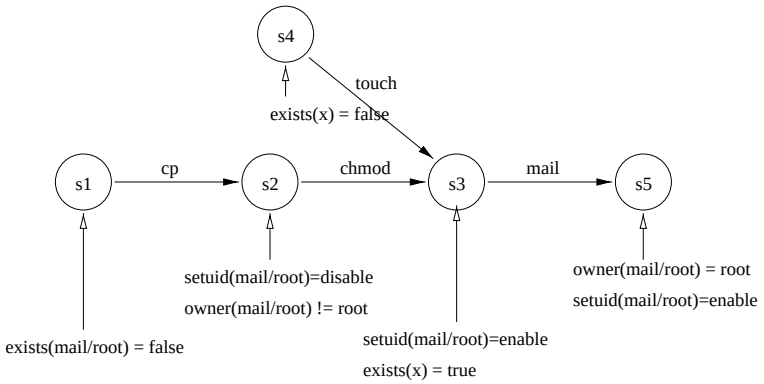


Figure 4: Partial order state transition diagram of the misuse penetration example

total order representation because it allows a planner to postpone or ignore unnecessary ordering choices. In state transition analysis, the number of possible ordering choices grows exponentially as the number of states grows. We restrict this growth using partial order planning. The construction of partially ordered plans was pioneered by the NOAH planner, and investigated in Tate's NONLIN system [RN95]. By using partial order notation, one can take advantage of problem decomposition to deal with complex domains without necessarily suffering exponential complexity. Further, one can represent many possible plans with one canonical form.

A partial order planner searches through the space of plans rather than the space of situations. It starts with a simple, incomplete plan, which we call a partial plan, which it expands until it obtains a complete plan that solves the problem. The operators in this process are operators on plans: adding a step, imposing an ordering that puts one step before another, instantiating a previously unbound variable, and so on. The solution is the final plan; the steps taken to reach it are irrelevant. The principle of least commitment says that a planner should avoid making ordering decisions unless they are necessary components of the plan. Partial ordering constraints and uninstantiated variables allow us to follow a least-commitment approach. Most partial-order planning algorithms search through the space of all possible plans to find one solution that is guaranteed to succeed [RN95].

The flexibility gained from the partially ordered plan representation allows a planning agent to handle quite complicated domains, which is useful in intrusion detection. A partial order representation provides a more accurate representation of signature actions than a total order, since only required dependencies are indicated. For example, in Figure 4, the only dependency of *touch* is that it occurs before *mail*. There is no connection between *touch* and *cp*, or between *touch* and *chmod*. The total order of Figure 2 is ambiguous: it is unclear which state orderings are required by the problem and which are side effects of the representation. Further, Figure 2 is less concise because each possible ordering must be stated explicitly in a separate diagram. In contrast, Figure 4 clearly indicates which action sequences are responsible for the compromised state.

The compromised state of Figure 4 can be represented by a first order logic statement including the system changes:

$$
\begin{aligned}
&\exists\ \textit{/usr/spool/mail/root}\ \ x \\
&\qquad \textit{/usr/spool/mail/root} \in x \wedge \\
&\qquad \textit{owner(/usr/spool/mail/root)} = \textit{root} \wedge \\
&\qquad \textit{setuid(/usr/spool/mail/root)} = \textit{enable} \\
&\Longrightarrow \textit{compromised(x)} = \textit{true}
\end{aligned}
$$

or, alternately, as a sequence of commands:

$$
\begin{aligned}
&\exists\ \textit{file1, file2, file3, x} \\
&\qquad \textit{owner(file1)} = x \wedge \\
&\qquad \textit{cp(file1, file2)} \wedge \\
&\qquad \textit{chmod(file2, 4755)} \wedge \\
&\qquad \textit{touch(file3)} \wedge \\
&\qquad \textit{mail(root, file3)} \wedge \\
&\qquad \textit{cp(file1, file2)} \prec \textit{chmod(file2)} \wedge \\
&\qquad \textit{chmod(file2)} \prec \textit{mail(root, file3)} \wedge \\
&\qquad \textit{touch(file3)} \prec \textit{mail(root, file3)} \wedge \\
&\Longrightarrow \textit{attacker(x)}
\end{aligned}
$$

This approach is useful for penetration analysis since for each system compromise the analyst need only identify a minimal set of signature actions and any required orderings.

Suppose an attacker performs the following steps:

1% *ls*
2% *ln target -x*
3% *ls*
4% *-x*

The first and third steps do not contribute, but were added by the attacker only to make detection more difficult. However, in the first order statement, it is clear which steps contribute to the penetration attempt:

$$
\begin{aligned}
&\exists\ \textit{file1, file2, x} \\
&\qquad \textit{owner(file1)} \neq x \wedge \\
&\qquad \textit{owner(file2)} = x \wedge \\
&\qquad \textit{ln(file2, file1)} \wedge \\
&\qquad \textit{execute(file2)} \wedge \\
&\qquad \textit{ln(file2, file1)} \prec \textit{execute(file2)} \\
&\Longrightarrow \textit{attacker(x)}
\end{aligned}
$$

# 3 Handling Anomaly Detection

In state transition analysis, the fundamental difference between anomaly detection and misuse detection is that we only identify the possibility of intrusion for each anomalous session, since not all anomalous activities are intrusions. Anomaly detection traditionally assumes that one can establish normal behavior patterns over time and use these patterns as profiles of normal system activity [Lie89]. Profile-based anomaly detection uses statistical measures to identify expected behavior, while rule-based anomaly detection uses sets of rules to represent and store the usage patterns in audit data. Since POSTAT does not have a learning mechanism, we cannot fully implement anomaly detection, but can provide similar functionality.

A typical profile within a statistical model might include the following components [Amo94]:

$$< \textit{subject, object, action, e\_pattern, r\_pattern, t\_pattern} >$$

Such a profile stipulates that, whenever the subject initiates an action on some object, it is expected that error conditions will be *e_pattern*, resource usage will be *r_pattern*, and time durations will be *t_pattern*. Suppose the following normal profile for user *joe* is constructed:

$$< \textit{joe, myfile, execute, 0, CPU(00:01-00:04), 2:00-22:00} >$$

This profile indicates that whenever joe executes *myfile*, no errors are expected, CPU usage should be within 1 and 4 seconds, and time of execution should be between 2AM and 10PM. We represent this normal user profile thus:

$$\begin{aligned} &\exists\ \textit{myfile} \\ &\qquad \textit{owner(myfile)} = \textit{joe} \wedge \\ &\qquad \textit{execute(myfile)} \wedge \\ &\qquad \textit{e\_pattern} = 0 \wedge \\ &\qquad \textit{r\_pattern} = \textit{CPU(00:01 - 00:04)} \wedge \\ &\qquad \textit{t\_pattern} = \textit{(2:00 - 22:00)} \end{aligned}$$

We use a first order logic statement to represent profiles (above) with three attributes (*e_pattern, r_pattern, t_pattern*) to identify the anomalous activities. There is no additional outside knowledge required to ascertain the flagged state. The only signature action that moves the system from the initial state into the final flagged state is the anomalous execution of *myfile* which deviates from its normal profile (Figure 5). Any execution of *myfile* which causes the deviation from its normal profile would move the system from the initial requirement state to the flagged state.

Similar profiles can be constructed for other security relevant activities:

$$< \textit{joe, login, execute, 3, CPU(00:01-00:04), PERIOD(00:01-00:30)} >$$

This login profile for user *joe* indicates that, whenever *joe* executes the login command, at most 3 errors are expected within the time period 30 seconds, CPU usage should be within 1 and 4 seconds. This profile is represented as:

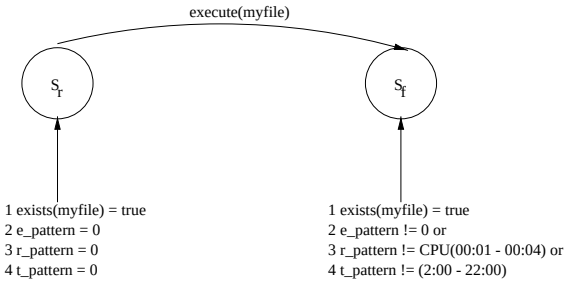


Figure 5: State transition diagram of anomalous transaction example

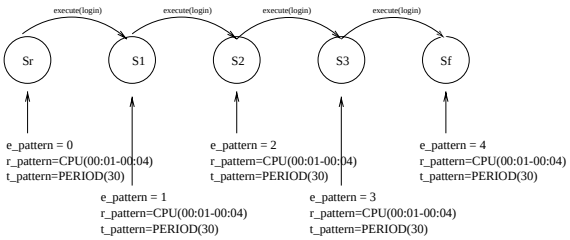


Figure 6: State transition diagram of anomalous login transaction

$\exists$ login

$$
\begin{aligned}
& \textit{execute(login)} \wedge \\
& \textit{e\_pattern} = 3 \wedge \\
& \textit{r\_pattern} = \text{CPU(00:01 - 00:04)} \wedge \\
& \textit{t\_pattern} = \text{PERIOD(00:01 - 00:30)}
\end{aligned}
$$

To use this profile as the basis for intrusion detection, audit records on computer usage taken by this individual must be available. If there is an attempt to break into the *joe* account using password guessing, the *e_pattern* will increase and *r_pattern* and *t_pattern* should remain normal. In the state transition diagram, each failed login which might lead the system from one state (initial or intermediate) into another state is recorded (Figure 6).

## 3.1 Multiple Attackers, Network Attacks, and Incomplete Scenarios

One useful feature of this representation is its suitability for describing many types of attacks. Multiple attackers can be represented by specifying the behavior of each attacker separately, and requiring that they all occur prior to the achievement of the goal. In our initial example, the *touch* of S4 could have been performed by an attacker other than the one issuing the *cp* of S1. Use of partial ordering is a powerful way to handle the asynchronous branches of

coordinated attacks. Network attacks can be handled as well, although it will be necessary to collect the relevant data in one location.

Incomplete scenarios can also be represented. An incomplete scenario is one in which some details of an attack are unknown. For example, when the Internet Worm first was detected, not all details of that attack were identified immediately. Using our technique, this would not be an issue, since it is only necessary to identify the known transitions that occur as part of the attack. However, one danger of using incomplete scenarios is an increase in the number of false positives.

# 4 Intrusion Scenario Construction

An intrusion scenario is defined by Heady [HLMS90] as "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource." We use a simple planning agent to construct plans that achieve goals which describe situations that are desirable for an intruder. The planning agent consists of goals, states and actions. In order to decide what to do in its plan, the agent combines the current state of the environment with information about the results of possible actions.

In state transition analysis, a penetration is viewed as a sequence of signature actions performed by an attacker that leads from some initial states on a system to a target compromised state. Before describing planning techniques in detail, we need to formulate state transition analysis as a planning problem:

- *Initial state:* An arbitrary logical sentence about a situation of the system.
- *Goal state:* A logical statement for certain situations of the system.
- *Signature actions:* Critical actions causing the state transition in the system.

A plan is formally defined as [RN95]:

1. A set of plan steps; each step is one of the operators for the problem;
2. A set of ordering constraints; each ordering constraint is of the form $S_i \prec S_j$, which is read as "$S_i$ before $S_j$" and means that step $S_i$ must occur sometime before step $S_j$;
3. A set of variable binding constraints; each variable constraint is of the form $\nu = x$, where $\nu$ is a variable in some step, and $x$ is either a constant or another variable;
4. A set of causal links; a causal link is written as $S_i \xrightarrow{c} S_j$ and is read as "$S_i$ achieves $c$ for $S_j$", where $c$ is a necessary precondition for $S_j$. Causal links serve to record the purpose(s) of steps in the plan, i.e., a purpose of $S_i$ is to achieve the precondition $c$ of $S_j$.

There are preconditions associated with each signature that indicate what must be true before the signature action can be applied, and postconditions indicating effects of a signature action. The task of our planner is to find a sequence of actions that allows a problem solver to accomplish some specific task, here an intrusion. The POSTAT planner is used to find a sequence of signature actions and their dependence, producing a penetration scenario. This information is used to search the intrusion scenarios.

A partial order planner is one that can represent plans in which some steps are ordered with respect to each other and other steps are unordered. Since some of the signature actions for intrusions are required to be ordered, partial order planning is appropriate. A *general planning agent* consists of a knowledge base and a planning engine. A *knowledge base* contains information (logical assertions) about every signature action of the system, including first order logic representation of the pre- and post-conditions of these actions. A planning engine will generate one set of the signature actions and its dependency for each pair of initial state and compromised state.

The planning agent's knowledge base also contains the dependency of state assertions for each signature action. It is used by the planner to define a partial ordering among signature actions. The advantage of this kind of representation is that it is more flexible and uses less space. For example, the precondition of a signature action consists of $k$ assertions. We use a set of symbols: $PS_1$, $PS_2$, ..., $PS_k$, to represent each state assertion. The data structure in the knowledge base for the logical assertions of a SA is:

$$\{PS_1, PS_2, ..., PS_k\} \ \wedge \ \{PS_j \prec PS_k\} \ \wedge \ ... \ \wedge \ \{PS_l \prec PS_m\}$$

A partial order planning algorithm starts with a minimal partial plan, and on each step extends the plan by achieving a precondition of a step. This is done by selecting a signature action that achieves some unfulfilled precondition of a step in the plan. The *causal link* for the newly achieved precondition of each signature action is also recorded. It is important to track causal links for the purpose of partial ordering. When there is no ordering requirement in the causal links in some steps, we can neglect their dependency in the output. The result is now represented by a set of signature actions and the dependency between these signature actions. Assuming an intrusion scenario is composed of $n$ signature actions, i.e., $SA_1$, $SA_2$, ..., $SA_n$, the data structure of scenario is specified as

$$\{SA_1, SA_2, ..., SA_k\} \ \wedge \ \{SA_i \prec SA_j\} \ \wedge \ ... \ \wedge \ \{SA_l \prec SA_m\}$$

The first component of this statement, $\{SA_1, SA_2, ..., SA_k\}$, is the set of signature actions. The additional components of this statement define the ordering constraints between signature action pairs. The scenario for the penetration example in Figure 4 is specified as:

$$\{cp, chmod, touch, mail\} \ \wedge \ \{cp \prec chmod\} \ \wedge \ \{chmod \prec mail\} \ \wedge \ \{touch \prec mail\}$$

Each statement represents a variation of signature action sets which would lead to the same compromised state. It is unnecessary to consider the different combinations of signature

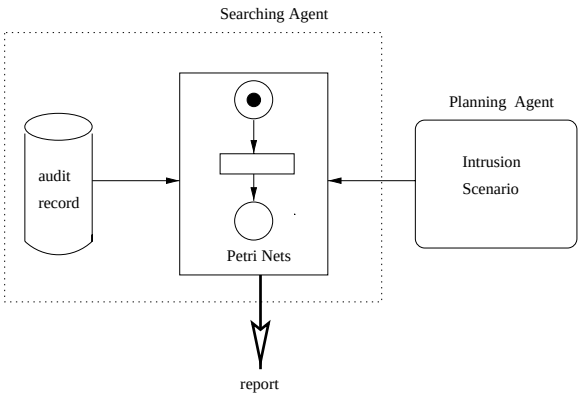


Figure 7: The structure of searching agent

actions, reducing space requirements. Furthermore, when the initial state is changed, there might be more than one set of signature actions, and again, several combinations of them.

If the planner is applied in a real-time system, it needs to observe the initial state of the system, and take input from activity patterns or activity profiles, which represent known compromise scenarios, to generate the planning steps. Another advantage of this partial order planning approach is that the amount of time passed between two signature actions has no effect on the analysis, as long as we continue recording system data and state changes.

# 5 The Role of Petri Nets

A simple Petri Net is based on events and conditions. Events are actions which take place in the system, and their occurrence is controlled by the states of the system. Each system state corresponds to a set of conditions and their values. We represent these as a set of first order logic statements indicating whether a given condition holds or does not hold. Some actions may only occur under certain conditions, and thus we may say that the state descriptions are preconditions for such events. When an event actually does occur, it may cause some preconditions to cease to hold and may cause other conditions to become true.

Each intrusion scenario is represented as a Petri Net. The places of the Petri Net correspond to existing states, or to the preconditions and postconditions of actions. The transitions of the Petri Net correspond to signature actions. We execute our Petri Nets by matching incoming events (our audit trail records) to the intrusion scenario Petri nets. Initial states correspond to initial system states, whereas final states correspond to a state in which an intrusion has apparently occurred.

We have successfully used Petri Nets to model sample scenarios [Ho97] from each of Kumar's five violation categories [KS94]:

- Existence - the fact that something that exists is a violation of security policy,

- Sequence - the fact that several things happen in strict sequence is sufficient to detect the intrusion attempt,
- Partial order - several events defined in a partial order are necessary to conduct the intrusion,
- Duration - this requires that something existed or happened for no more than or not less than a certain interval of time, and
- Interval - things happened an exact interval apart; specified by the conditions that an event occur no earlier and no later than some units of time after another event.

In an **existence** pattern "something" can be a single file or action. Simple existence of a file can often be found by static scanning of the file system. This situation can be trivially modeled by a Petri net with a single marking place, as in Figure 8.

In the second category of Petri Nets, the **sequence net**, we can perform an approximate matching, rather than an exact matching. The firing rule is the same as for the condition-event nets. When a transition is enabled at $M$, it needs to wait for the occurrence of the action in order to fire the transition. Thus, it is not an automatic firing as in other nets. In a sequence net, only one initial place is marked. Figure 9 shows a Sequence Net for an intruder who exploits a flaw in the shell mechanism. The activities represented in the Petri Net are composed of a sequence of the actions and states. $S_1$ is the initial state with marked token, and $S_3$ is the final state. After the action of *ln*, the state of $S_2$ is reached and action *file2* can be fired to reach the final state $S_3$.

The third category, **duration net**, requires knowing that something existed or happened for no more than or no less than a certain interval time. Again, "something" can be a single file or action; we are primarily interested in the existence of signature actions. The firing rule is more complicated than with the condition-event nets, since when a transition is enabled, it must wait for the occurrence of the signature action and to checks the satisfaction of the optional expression in order to fire the transition. Suppose that it is a violation of security policy for a user to have three failed login attempts within a one minute time interval. In Figure 10, the existence of three *flogin* is described by three transitions with optional expressions specifying the time restrictions for these three signature actions. When the last transition is fired, the net reaches its final state $S_4$.

The fourth category is the **partial order net**. In a partial order pattern, defining several events in a partially ordered set is necessary to specify the intrusion. This can be considered as more than one sequence. The merger of these sequences is very important because there is only one terminal state in the final result. In a partial order net, there may be more than one initial place being marked. The attack scenario in Figure 3 can be specified by the Petri Net model [KS94] in Figure 11. $S_1$ and $S_4$ are the initial states with marked token, and $S_6$ is the final state. A token is placed in each initial state. After the action of *cp, chmod* and *touch*, the states of $S_3$ and $S_5$ are reached and action *mail* can be fired to reach the final state $S_6$.

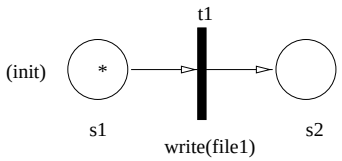


Figure 8: An Existing Petri Net in Intrusion Detection

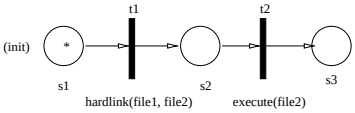


Figure 9: A Sequence Petri Net in Intrusion Detection

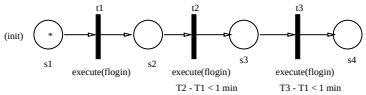


Figure 10: A Duration Petri Net in Intrusion Detection

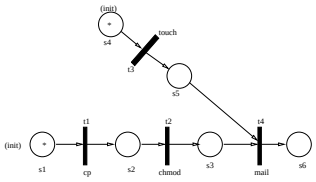


Figure 11: A Partial Order Petri Net in Intrusion Detection

# 6 Conclusion

Preliminary tests of a representative set of penetration scenario (defined in the scenario base) resulted in detection by POSTAT. Some basic features of POSTAT were also tested in these experiments, such as the variations of the penetration scenarios constructed by the planning agent and sensitivity to the order of signature actions. The most challenging task in this state transition approach is how to give a complete definition of the state assertions for all known compromised states and how to define all the signature actions. As with other similar approaches, POSTAT is only capable of detecting attacks if they have a known form, if some aspect of their behavior is known, or if they involve a predetermined undesirable behavior or state change.

# References

[Amo94] E.G. Amoroso. *Fundamentals of Computer Security Technology.* PTR Prentice Hall, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1994.

[HLMS90] R. Heady, G. Lunger, A. Maccabe, and M. Servilla. The architecture of a network level intrusion detection system. Technical report, Department of Computer Science, Univ. of New Mexico, August 1990.

[Ho97] Y. Ho. Partial order state transition analysis for an intrusion detection system. Master's thesis, University of Idaho, 1997.

[IK95] K. Ilgun and R.A. Kemmerer. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3), March 1995.

[KS94] S. Kumar and E.H. Spafford. An application of pattern matching in intrusion detection. Technical report CSD-TR-94-013, Department of Computer Science, Purdue University, 1994. COAST Project.

[Lie89] G.E. Liepins. Anomaly Detection: Purpose and Framework. In *In Proceedings of the 12th National Computer Security Conference*, pages 495–504, October 1989.

[RN95] S. Russell and P. Norvig. *Artificial Intelligence - A Modern Approach.* Prentice Hall, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1995.