State Transition Analysis: A Rule-Based Intrusion Detection Approach

Koral Ilgun, Richard A. Kemmerer, *Fellow, IEEE*, and Phillip A. Porras

Abstract— **This paper presents a new approach to representing and detecting computer penetrations in real-time. The approach, called state transition analysis, models penetrations as a series of state changes that lead from an initial secure state to a target compromised state. State transition diagrams, the graphical representation of penetrations, identify precisely the requirements for and the compromise of a penetration and present only the critical events that must occur for the successful completion of the penetration. State transition diagrams are written to correspond to the states of an actual computer system, and these diagrams form the basis of a rule-based expert system for detecting penetrations, called the State Transition Analysis Tool (STAT). The design and implementation of a UNIX-specific prototype of this expert system, called USTAT, is also presented. This prototype provides a further illustration of the overall design and functionality of this intrusion detection approach. Lastly, STAT is compared to the functionality of comparable intrusion detection tools.**

Keywords— **Security, Intrusion Detection, Expert Systems**

I. Introduction

OVER the past decade, significant progress has been made toward the improvement of computer system security. Unfortunately, the undeniable reality remains that all computers are vulnerable to compromise. These systems are vulnerable to attacks from both non-authorized users (outsider attacks) as well as attacks from authorized users who abuse their privileges (insider attacks). Given this reality, the need for user accountability is very important, both as a deterrent and for terminating abusive computer usage once it is discovered. The need to maintain a record of the transactions that have occurred on a system is also crucial for performing damage assessment.

In recognition of these needs and in recognition that security violations are a fact of life, many computer systems implement a form of transaction record keeping called audit collection. Audit collection is an important management tool, which allows administrators to reconstruct and examine the sequences of actions performed by individual users. Until recently, however, the ability to utilize audit data as a means of detecting system compromises has been limited due to the enormous quantity of data collected. In order to provide enough information to establish accountability and enable damage assessment, the audit collection mechanisms must record the occurrences of all security-relevant events.[1] Because of the large volume of data generated, manual analysis of the audit data is usually impractical.

Within the past few years, there has been a steadily growing interest in the research and development of automated audit data analysis tools, referred to as intrusion detection systems. These tools aid analysts in identifying security violations that might have once gone unnoticed, and provide one of the few, if not only, practical means of analyzing audit data efficiently. In addition, intrusion detection provides an added value to audit data that cannot be realized using manual analysis. When performed in real-time, intrusion detection can use audit data to track user behavior and determine if a user's current actions represent a threat to security.

The Reliable Software Group at UCSB has recently developed a new approach to representing computer penetrations and has applied this approach to the development of a real-time expert system intrusion detection tool. The approach is called *state transition analysis* and is a method for representing the sequence of actions that an attacker performs to achieve a security violation. A *state transition diagram*, which is the graphical representation that the state transition analysis technique uses to represent a penetration, identifies precisely the requirements and the compromise of a penetration and lists only those critical events that must occur for the successful completion of the penetration. By using the information contained in a system's audit trail as input, an analysis tool can be developed to compare the state changes produced by the user to the state transition diagrams of known penetrations.

The next section presents a brief overview of current intrusion detection techniques, identifying how the state transition analysis approach relates to other approaches. Section III introduces the state transition approach as a technique for representing computer penetrations. It also introduces the design of a real-time intrusion detection tool (STAT), which is based on the state transition analysis technique. In section IV a UNIX-specific prototype implementation of STAT, called USTAT, is discussed. Section V discusses both the benefits and costs associated with STAT, comparing its functionality to three comparable intrusion

This research was partially supported by the National Computer Security Center under grant MDA904-88-C-6006.

K. Ilgun is with Advanced Computer Communications, 315 Bollay Drive, Santa Barbara, CA 93117 USA; email: koral@acc.com.

R.A. Kemmerer is with the Reliable Software Group - Department of Computer Science, University of California, Santa Barbara, CA 93106 USA; email: kemm@cs.ucsb.edu.

P.A. Porras is with The Aerospace Corporation, P.O. Box 92957, Mail Stop: M1/055, Los Angeles, CA 90009-2957, USA; email: porras@aero.org

This paper is an extended version of two previous conference papers by the same authors [15,28].

[1] In general, the term security-relevant is left open to interpretation. With respect to the Trusted Computer Systems Evaluation Criteria (TCSEC) [24], a security-relevant event is defined as any event that attempts to change the security state of the system (e.g., changing the security level of a user or changing a user's password).

detection tools that are presented in section II. Section VI discusses the future directions of this project. Lastly, section VII concludes with a summary of the key points presented in this paper.

II. Current Approaches to Intrusion Detection

There has been a steadily growing interest in the research and development of intrusion detection systems. Surveys of implemented intrusion detection systems, many of which are in operation today, can be found in [16], [22], [27], and examples of actual competitive assessments between intrusion detection systems can be found in technical reports by both the Navy and the Air Force [23], [10].

This section surveys various approaches to intrusion detection, and gives examples of currently available tools that employ these approaches. No intrusion detection approach stands alone as a catch-all for computer penetrations; each approach is technically suited to identify a subset of the security violations to which a computer system is subject. The intent of this section is to give a brief overview of current intrusion detection techniques to better identify how the state transition analysis approach fits into the general scheme of things. Understanding the strengths and limitations of these approaches will also provide the reader with a reference point for understanding the benefits, as well as the tradeoffs, to the approach presented in this paper.

A. Threshold Detection

Threshold detection, or summary statistics, is one of the most rudimentary forms of intrusion detection. The goal of threshold detection is to record each occurrence of a specific event and, as the name implies, detect when the number of occurrences of that event surpass a reasonable amount that one might expect to occur within a specified time period. The events that are recorded are such that an unnaturally high number of occurrences within a short period of time may indicate the presence of an intruder. Once the threshold number of occurrences is surpassed, the threshold detector has the option to either preempt the source of the event, if possible, or notify the Site Security Officer (SSO).

When implementing a threshold detector, the most obvious difficulty is identifying the threshold number and the window size for a specific event. Both the threshold number and the time interval of the analysis depend upon the security-relevance of the event being monitored, as well as the historical number of occurrences. Therefore, the choice of these values is often left to the discretion of the SSO.

An example of the type of events that might be monitored using threshold detection is the number of failed logins, the number of I/O errors, or the number of file deletions. Remote connection audit trails provide another profitable area to employ threshold analysis. Systems may install threshold detectors to track unusually large data transfers or other accesses from remote sites.

Threshold analysis alone is a poor detector of intrusions, and is usually implemented as a sub-component of a larger intrusion detection system. Two intrusion detection systems that use threshold detection as a subcomponent are the Multics Intrusion Detection and Alerting System (MIDAS) [30] and the Network Anomaly Detection and Intrusion Reporter (NADIR) [11]. MIDAS is a real-time intrusion detection tool operating on the National Computer Security Center's (NCSC) network mainframe, Dockmaster. NADIR is a real-time audit data analysis tool, which was developed at Los Alamos National Laboratory.

B. Anomaly Detection

Anomaly detection is one of the earliest approaches to intrusion detection. Both statistical and rule-based forms of anomaly detection have been implemented in recent years. The premise behind anomaly detection was introduced in the Anderson report [1], which identified three classes of malicious users:

- the masquerader – an individual who penetrates a computer's access controls to exploit a legitimate user's account
- the misfeasor – a legitimate user who participates in illicit activity on a computer
- the clandestined user – an individual who seizes supervisory control of a computer, and uses this control to operate below the level of audit collection or to suppress audit collection altogether

The Anderson report proposed that masqueraders, and in some cases misfeasors, could be detected by monitoring a system's audit log for user activity that deviates from established patterns of usage.

The objective of anomaly detection is to establish usage patterns within user audit trails over a duration of time,[2] and use these usage patterns as profiles of "normal" system activity. An audit trail that is found to deviate from the user's established usage pattern is flagged and brought to the attention of the SSO. These anomalous sessions may potentially reveal a masquerader or a legitimate user abusing his/her privileges.

The main advantage of anomaly detection is that it provides a means of detecting intrusions without *a priori* knowledge of the security flaws in the target system. The approach requires few, if any, system dependent rules or statistical formulas, making tools that employ anomaly detection highly portable. The two primary types of anomaly detection are statistical profile-based and rule-based. Profile-based anomaly detection uses statistical measures to identify expected behavior. In contrast, rule-based anomaly detection uses sets of rules to represent and store the usage patterns in audit data, rather than statistical formulas.

In 1985, Denning and Neumann presented a detailed discussion of statistical profile-based anomaly detection [5]. Perhaps the best known statistical profile-based anomaly detection system is the Intrusion Detection Expert System (IDES) [17], [19], [20]. The profile-based anomaly component of this system identifies expected behavior at the

[2] The longer the duration, the more accurate the study.

user, group, remote host and target system levels. For an in depth discussion of IDES, the reader is referred to [12].

Two example intrusion detection implementations that employ rule-based anomaly detection are Wisdom and Sense (W&S) [34] and the Time-based Inductive Machine (TIM) approach [3]. Neural network-based anomaly detection has also been proposed in recent work [4], [20].

Anomaly detection is not without limitations. In many environments, it may be difficult to establish behavior patterns for users. For example, in sporadic user environments establishing profiles of normal user behavior would be difficult. This leads to a potentially large number of *false positives*. In recognition of this problem, anomaly detection tools provide parameters to decrease their sensitivity to certain anomalies, while maintaining enough sensitivity to identify penetrations. This fine tuning is a difficult process and perhaps one of the most *ad hoc* aspects of the approach.

There are also penetrations that could compromise a system and yet produce no identifiable anomaly or that produce anomalies that are below the sensitivity levels of the configuration parameters. The rate at which such penetrations are missed, referred to as the *false negative rate*, is just as significant as the false positive rate. To counter these limitations intrusion detection systems that employ anomaly detection often supplement their detection capabilities with expert rules for identifying known penetrations.

C. Rule-based Penetration Identification

A rule-based penetration identification system is an expert system whose rules fire when audit records are parsed that appear to indicate suspicious, if not illegal, user activity. The rules may recognize single auditable events that represent significant danger to the system by themselves, or they may recognize a sequence of events that represent an entire penetration scenario.

Rule-based penetration identifiers have become a common supplemental component of intrusion detection systems that also employ anomaly detection components. The expert rules offer the additional capability of identifying dubious behavior, even when the behavior appears to conform with established patterns of use. Examples of intrusion detection implementations that supplement their anomaly detection components with expert penetration rules include IDES, NADIR and W&S.

IDES has a rule-base component that allows one to represent suspicious behavior based on site-specific security policies, known security flaws, and knowledge of past intrusions [18]. The IDES expert system component evaluates audit records as they are produced. From the perspective of the expert system, the audit records are viewed as facts, which map to rules in the rule-base. A binding analysis is performed to determine if the fact/rule binding is consistent. That is, the expert system verifies that certain fields in the audit record match what is expected by the rule. If they do, the rule's consequent is fired, increasing the suspicion rating of the user responsible for the record. Each user's suspicion rating starts at zero and is increased with each suspicious record. The greater the number of rules fired, the greater the increase in the suspicion rating. Once the suspicion rating surpasses a pre-defined threshold, a report is produced. The IDES anomaly detection component in combination with the IDES penetration identifier rule-base provides a way of learning and representing normal behavior, as well as a way of representing improper behavior. Together, they form a complementary pair of analysis tools that provide coverage for identifying both masqueraders and misfeasors.

Wisdom and Sense (W&S) and NADIR are further examples of intrusion detection systems that combine anomaly detection with penetration identification. In the case of W&S, the anomaly detection component is also implemented as a rule-base. In this system, the penetration detection component is actually combined into the same rule-base to represent site-specific policies, expert penetration rules, and other administrative data. NADIR's implementation is similar to IDES in that it employs a statistical anomaly component in conjunction with a small rule-base of expert penetration rules. NADIR's expert rule-base consists of penetration rules that are developed by interviewing and working with security personnel.

D. Model-Based Intrusion Detection

Model-based intrusion detection attempts to model intrusions at a higher level of abstraction than audit records. The objective is to build *scenario models* that represent the characteristic behavior of intrusions. This allows administrators to generate their representation of the penetration abstractly, which shifts the burden of determining what audit records are part of a suspect sequence to the expert system. Model-based techniques differ from current rule-based techniques, which simply attempt to pattern match audit records to expert rules.

The Model-based technique proposed by Garvey and Lunt [8] supports the abstraction of penetrations via an evidentiary reasoning tool, called Gister.[3] The goal of the tool is to evaluate pieces of evidence against a hypothesis in an effort to build some confidence measure as to the veracity of the hypothesis. As evidence is discerned through the audit trail to indicate that activity represented in a scenario model is being performed, the model is added to a set of *active models*. Active models are used by Gister to increase/decrease a belief measure that a penetration is occurring (e.g., that Gister is witnessing an intruder session).

E. Intrusion Prevention

The search for analysis tools to aid in the protection of not-so-secure computing systems is not limited to the development of intrusion detection systems. Practical tools for preventing intrusions have also been developed. One notable prevention tool is the Computer Oracle Password and Security System (COPS), developed by Farmer and Spaf-

[3] Gister is a trademark of SRI International.

ford [7]. COPS is designed to aid UNIX system administrators in testing their configurations for common weaknesses often exploited by intruders. COPS is a collection of shell scripts and programs that search out and identify a myriad of security holes commonly present on UNIX systems.

COPS calls to the attention of the administrator critical areas that are often neglected or misunderstood. Optimally, COPS helps to educate novice administrators about the problem areas they should be looking for on their systems, and it may also catch mistakes made by experienced administrators.

There are other intrusion prevention tools developed to both detect and prevent the execution of computer viruses and Trojan horses.[4] Just as viruses have been created for various computer systems, code analysis tools have been created to identify and, in some cases, remove the viruses. In their paper [13] Kerchen, et al. provide an example of two virus detection tools for UNIX. Shieh and Gligor have also proposed an intrusion detection system that provides a means to track virus propagation [31].

F. Limitations in Existing Approaches

The state transition analysis approach presented in this paper is a rule-based penetration identification approach. It seeks to improve on some of the limitations that the authors observed in current rule-based penetration identification tools. This section briefly identifies some of the inherent characteristics that limit the effectiveness of current rule-based penetration identification tools, which the state transition analysis technique has been designed to address.

A major weakness in current rule-based penetration identification tools is their direct dependence on audit record fields. In the current systems, rule-bases represent the expected audit trails of penetrations, and these tools essentially pattern match and bind rules in their knowledge-base to audit records. Unfortunately, there is very little flexibility in this one-to-one (rule-to-audit record) representation. For instance, for a given penetration scenario there may be slight variations of the same penetration that will produce different audit record sequences. Thus, even if a scenario is represented in the rule-base, a minor variation of the scenario can go unnoticed. One solution to improving the flexibility of the expert system's ability to identify penetration scenarios is to use higher-level representations of the scenarios in the rule-base (i.e., scenario representations that are audit record independent).

Another limitation to current penetration identification expert systems is their inability to foresee an impending compromise and preempt or limit the damage before it occurs. At best, current penetration identification systems report compromises after they are reached or take measures to terminate an intrusive process once the damage has begun. Current approaches are designed with little, if any, reasoning capabilities that allow them to take preemptive action before a compromised state is reached. Intrusion detection systems should be able to anticipate an impending compromise with some measure of confidence and either forewarn the system administrator or take steps to preclude a penetration before it achieves its compromise.

Lastly, current penetration rule-bases are neither easily created nor easily updated. In general, expert rule-bases tend to be nonintuitive, requiring the skills of experienced rule-base programmers to update them; penetration rule-bases are no exception. Penetration rule-bases are created by interviewing system administrators and security analysts to collect a suite of known penetration scenarios and key events that threaten the security of the target system. The rule-base programmer then identifies the audit records that correspond to the scenario or key event and constructs rules to represent the penetration based on the expected audit records. The development of penetration rules are *ad hoc* and provide little chance for the rule-base to be updated on-site. Procedures that allow system administrators and security analysts to develop and incorporate penetration rules into the rule-base locally should be provided. Doing so will result in more effective rule-base management, allowing site-specific policy information and newly discovered penetrations to be incorporated into the rule-base in a timely manner.

III. State Transition Analysis

This section presents a high level discussion of the state transition analysis approach to representing and detecting computer penetrations. The following section explains the premise of the technique and defines key terms that will be used throughout the remainder of the paper. Next, section III-B continues this discussion by explaining how state transition analysis is used to represent penetrations with a state transition diagram. Section III-C explains how these representations are applied to the states of an actual computer, and section III-D concludes the discussion by presenting a functional description of STAT, which is a real-time intrusion detection tool.

A. Representing Penetrations Using State Transition Analysis

State transition analysis is based on the premise that all computer penetrations share two common features. First, penetrations require the attacker to possess some minimum prerequisite access to the target system. This prerequisite access may range from access to certain files, devices, telephone lines, etc. to possession of information regarding a particular security-relevant function. Second, all penetrations lead to the acquisition of some previously unheld ability. That is, a subject performing a penetration is doing so to gain something. Whether the ability gained is unauthorized access to data, access to another user's privileges, or just a perverse satisfaction in harming others, something is gained. Identifying what is acquired by the penetrator is analogous to identifying what aspect of a computer's security has been compromised by the penetration.

In state transition analysis, a penetration is viewed as

[4] A Trojan horse is a computer program with an apparent or actual useful function that contains additional (hidden) functions that surreptitiously exploit the privileges of the invoking process. [26]

a sequence of actions performed by an attacker that leads from some initial state on a system to a target compromised state, where a *state* is a snapshot of the system representing the values of all volatile, semi-permanent and permanent memory locations on the system. 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 are one or more intermediate *state transitions* that an attacker performs to achieve the compromise.

After the initial and compromised states of a penetration scenario have been identified, the key actions, called *signature actions* are identified. Signature actions refer to those actions that, if omitted from the execution of an attack scenario, would prevent the attack from completing successfully. The information produced by the above steps are represented graphically, as a state transition diagram.

What makes this approach useful for penetration analysis is that it requires the analyst to identify the minimum number of key, or signature, actions for a penetration and to organize them visually, using a representation similar to state machine diagrams. This approach is useful for describing penetration scenarios in that it provides an interesting level of abstraction to the analyst: just above the system call and user command representation, and just below English descriptions. Representing penetrations as a sequence of user commands or system calls as in Table I (This will be discussed in the next section.) does not identify the key action sequences responsible for the compromised state, nor does it identify the minimum requirements of the penetration. English, on the other hand, is too ambiguous and often not very concise. State transition diagrams identify precisely the requirements and compromise of the penetration, and list only the key actions that must occur for the successful completion of the penetration.

B. An Example State Transition Analysis of a Penetration

State transition analysis is best illustrated by example. This section presents an example penetration scenario, and explains how one converts the scenario into a state transition diagram. For brevity, this example will assume that the reader has some familiarity with the UNIX operating system.

Table I presents an example penetration scenario for 4.2 BSD UNIX that can be used to illegally acquire root privilege [2]. In this scenario, the attacker exploits a flaw in the *mail*(1) utility, in which *mail* fails to reset the setuid bit of the file to which it appends a message and changes the owner.[5] As a result, the attacker is able to trick *mail* into creating a setuid shell program that is owned by root and publicly executable.

In step 1, the attacker creates a copy of *csh*(1) and names it after root's mail file. For this step to be successful, the attacker must wait until root has no unread mail, otherwise the attacker will not be able to create the counterfeit mail file. In step 2, the attacker enables the setuid bit of the counterfeit mail file. In steps 3 and 4, the attacker creates and sends an empty message to root via the *mail* utility. The security flaw arises when, in step 4, *mail* fails to reset the setuid bit of `/usr/spool/mail/root` before it sets the file's owner attribute to root. As a result, in step 5 the attacker need only execute root's mail file to gain access to a shell with root privilege.[6]

To model this penetration as a sequence of state transitions, one must first identify the initial requirement state and the target compromised state. To successfully execute the above scenario, the following assertions must hold:

1. the attacker must have write access to the mail directory;
2. the attacker must have execute access to *cp(1), mail(1), touch(1)* and *chmod(1)*;
3. root's mail file must not exist, or must be writable; and
4. the attacker cannot be root.

Of course, variants of this penetration could be used to achieve the compromise without satisfying all four assertions. However, to achieve the above scenario as presented, each assertion must hold. The first assertion is necessary because the entire penetration is based on a flaw within the *mail* utility. Since *mail* specifically searches for mail files in `/usr/spool/mail/`, the attacker must have write permission to this directory to create the counterfeit mail file. The need for the second assertion is obvious; the omission of execute access to any of the listed programs would prevent the completion of a step in the attack, thus causing this specific scenario to fail. The third assertion is needed because while root's legitimate mail file exists, the attacker will not be able to create the counterfeit mail file. Lastly, the fourth assertion denotes the fact that a process cannot forge a counterfeit mail file that it owns. The need for this assertion will become clearer once the compromised state is examined below.

Note that within the first two assertions there is an implied requirement that the attacker has access to some system process, either through a user account or through a remote process. Without access to a process on the system, the attacker would neither have write access to `/usr/spool/mail/root` nor execute access to any of the four programs listed in the second assertion. For the purpose of this example, it is assumed that the penetration occurs on a system employing the standard 4.2 BSD UNIX configuration. Hence, assertions 1 and 2 will automatically be satisfied by the standard UNIX configuration, and are therefore removed.

The compromise achieved through this penetration is somewhat less obvious than it first appears. At first glance, the compromise may appear to be that the attacker gains illegal access to root privilege. However, upon closer inspection it becomes apparent that the compromise occurs before the attacker gains root privilege. Suppose that step

[5] All references to manual pages are to [33].

[6] The appended contents of message x will be taken as part of the symbol table of *csh*.

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
6.	`root%`	

TABLE I
Penetration Scenario 1

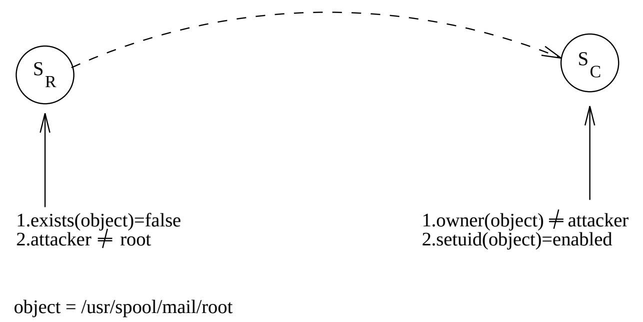

Fig. 1. Requirements and Compromised States of Penetration 1

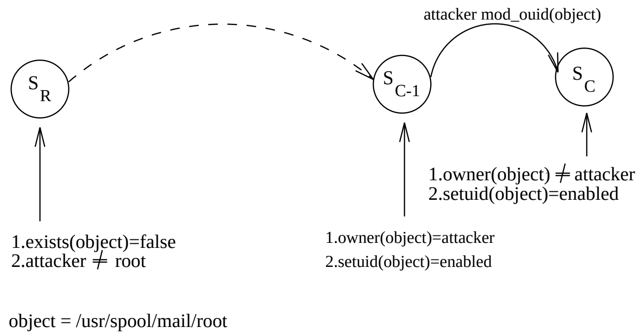

Fig. 2. Intermediate State Transition Diagram of Penetration 1

5 of Table I is not performed. Although the attacker has not used the counterfeit mail file to acquire root privilege, there would still be a security violation in that there now exists a setuid-to-root program that root did not create. The violation actually occurs the moment that *mail* modifies the owner attribute of the mail file while the mail file's setuid bit is enabled.

The initial requirement and compromised states of the penetration are illustrated by Figure 1. Here the penetration is represented as the dashed arc, leading from the initial requirement state S_R to the target compromised state S_C. The dashed arc refers to an unspecified sequence of actions that occur during the execution of the penetration. Listed below S_R and S_C are the assertions that describe the state of the system just prior to the execution of the penetration and just after its completion, respectively.

The next task is to identify the individual actions performed during the penetration that led to the compromised state. Identifying the individual steps of the penetration is best performed by dissecting the steps of the penetration scenario in reverse order. The goal is to identify the minimum number of actions performed during the scenario that can be used to accurately represent the penetration. Actions do not necessarily correspond to command lines entered by the attacker, but instead refer to how a specific state change within the system is achieved. A single command line can produce multiple actions. For example, the execution of the command line in step 4 of Table I causes the following actions to occur:

1. *mail*(1) is executed,
2. process E_UID is changed to root,
3. file /usr/lib/Mail.rc is referenced,
4. file $HOME/.mailrc is referenced,
5. file x is referenced,
6. file /usr/spool/mail/root is referenced,
7. file /usr/spool/mail/root is modified,
8. file /usr/spool/mail/root owner attribute is modified.

Since step 4 had previously been identified as the step in which the compromised state is reached, it is the best place to begin the search for the penetration's signature actions. Recall that the compromised state occurred when the owner attribute of `/usr/spool/mail/root` was changed, while at the same time the file's setuid bit remained enabled. Of the eight actions listed above, action 8 is the only one directly responsible for at least one of the assertions in the compromised state.

Figure 2 illustrates the updated version of the state transition diagram for this penetration, which includes the addition of action 8. Below state S_{C-1} are two assertions indicating that to this point file `/usr/spool/mail/root` is owned by the attacker and the file's setuid bit is enabled. These assertions represent the state of the system just prior to the execution of action 8. The dashed arc leading from the initial requirement state S_R to the intermediate state S_{C-1} indicates that some unspecified action(s) resulted in the creation of `/usr/spool/mail/root` with its setuid bit enabled. The solid arc leading from state S_{C-1} to S_C is labeled to indicate that from the intermediate state S_{C-1}, the attacker modified the owner attribute of `/usr/spool/mail/root`, which led to the compromised state S_C.

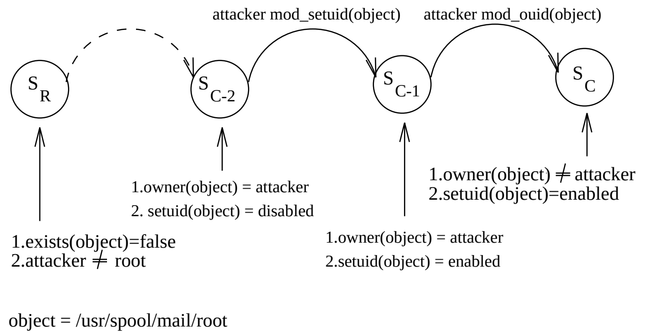

Fig. 3. Intermediate State Transition Diagram of Penetration 1

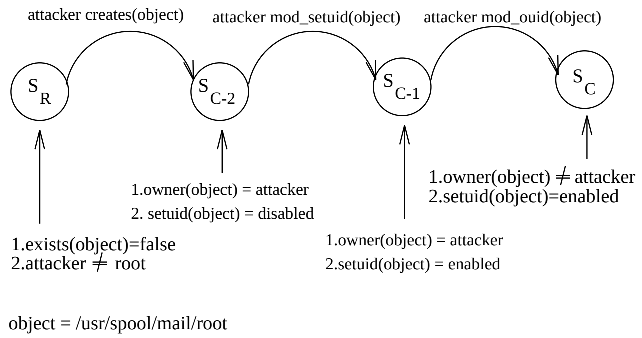

Fig. 4. Final State Transition Diagram of Penetration 1

Next, step 3 of Table I is analyzed to determine what role it had in the overall penetration. In step 3, an empty file x was created and subsequently mailed to root in step 4. However, neither the reference to file x nor the modification of file `/usr/spool/mail/root` were identified as being necessary intermediate actions in step 4 (action 8 is the only necessary action identified in Step 4). The creation of file x contributes nothing to the creation of file `/usr/spool/mail/root` or to the enabling of `/usr/spool/mail/root`'s setuid bit. Thus, step 3 does not contribute any necessary action to the overall penetration.[7]

In step 2, *chmod*(2) is used to enable the setuid bit of file `/usr/spool/mail/root`, and is one of the prerequisite assertions of state S_{C-1}. Figure 3 represents the updated version of the state transition diagram, which includes the enabling of `/usr/spool/mail/root`'s setuid bit performed during step 2. A second intermediate state S_{C-2} is added to represent the state of the system just prior to the execution of step 2's action and following the unspecified action (the dashed arc) that follows state S_R. Below state S_{C-2} are two assertions describing the state of the system just prior to the execution of step 2. To this point, file `/usr/spool/mail/root` exists, it is owned by the attacker, and it has its setuid bit disabled. The solid arc leading from state S_{C-2} to S_{C-1} is labeled to describe the action that enabled `/usr/spool/mail/root`'s setuid bit.

The last piece of information needed to complete the state transition graph for this penetration is to determine the origin of file `/usr/spool/mail/root`. A quick inspection of step 1 reveals that the attacker created /usr/spool/mail/root using *cp*(1), and that `/usr/spool/mail/root` is simply a copy of the root-owned program `/bin/csh`. Step 1 is a key action required by the penetration, because it identifies the origin of root's mail file. Figure 4 represents the final state transition diagram for the penetration.

[7] Adding the creation of file x, or any other non-signature action, to a state transition diagram does not make the diagram an incorrect representation of the penetration. Instead, it merely increases the specificity of the diagram, reducing the number of variations to the penetration that are represented by the diagram.

C. Applying State Transition Analysis to Actual Computer States

As discussed previously, a state transition diagram represents the sequence of signature actions that move the system from the initial requirement state, through zero or more intermediate states, into the final compromised state. Although the term *system state* is very broad, encompassing the values of all volatile, semi-permanent and permanent memory locations on the system, only a fraction of these attributes are needed to represent the signature state transitions of a particular penetration. Accordingly, system state, as it is used here, refers only to those system attributes that are necessary to represent the initial, intermediate and compromised states of a penetration.

To determine which system attributes are needed to perform a state transition analysis, one must first understand the types of security violations that this approach seeks to represent. State transition analysis is intended as a method for representing the sequence of actions that a misfeasor performs to achieve a security violation. It targets all known penetration scenarios that lead to an identifiable compromised state on the system. The term *identifiable compromise* is used to point out that although all penetrations lead to a compromise, not all compromises can be represented purely through the analysis of system attributes alone, and may thus fall outside the scope of state transition analysis.

Examples of penetrations outside the scope of state transition analysis include penetrations that focus on the manipulation of components outside the system's execution domain, such as wire taps. These attacks produce no compromised state perceivable by the examination of system attributes alone. That is, they produce a compromised state that can only be distinguished using knowledge that is outside of the system's execution domain.

If a penetration's compromised state cannot be ascertained without requiring outside knowledge of the attacker's identity or intentions, then the penetration will also be outside the scope of state transition analysis. For example, a masquerader who uses a legitimate user's login ID and password without permission is clearly violating the security of the system. However, there is no way for a state transition diagram to represent any information other than user actions and assertions regarding the state of the system. Thus, the representation of such a penetration using a state transition diagram would be indistinguishable from normal usage.

The key factors that determine whether a penetration is representable via a state transition diagram are

1. the compromise must produce visible change to the system state, and
2. the compromised state must be recognizable without external knowledge, such as the attacker's true identity or intentions.

Up to this point, state transition analysis has been discussed as a method for representing computer penetrations. However, during the development of this approach as a classification scheme for aiding penetration analysis, it was noted that if the state changes of a computer system could be monitored, then the approach could be adapted to aid in detecting penetrations as well. The only requirement needed to transform the state transition approach from a classification scheme to a detection scheme is the development of a mechanism for recording the state changes in system attributes. Fortunately, such a mechanism already exists on many computer systems: the audit facilities.

Audit facilities record the state changes made by users on monitored system attributes. By using the information contained in the system's audit trail as input, an analysis tool can be developed to compare the state changes produced by the users to the state transition diagrams of known penetrations.

Because not all modifications and references to system attributes are recorded within an audit trail, the effectiveness of state transition analysis as a detection technique may be limited. As a result, some penetrations that are representable using the state transition approach may not be detectable using a state transition analysis approach for detection. For example, audit mechanisms do not usually record references or modifications to volatile memory such as registers or user space. Therefore, penetrations involving an illegal modification or reference to volatile memory space may not be detectable using state transition analysis, even though they are representable as state transition diagrams.

The above limitation does not necessarily indicate a weakness in state transition analysis as an intrusion detection approach, for no rule-based penetration detection approach can be expected to detect penetrations that compromise attributes whose accesses are not recorded by the audit mechanism. Such penetrations execute beneath the visibility of audit data analysis tools and correspond to what is referred to as clandestine usage [1]. The interested reader may refer to [9], which examines a number of penetration scenarios and categorizes them into those that are identifiable via audit data analysis and those that are not.

D. The State Transition Analysis Approach to Intrusion Detection

The remainder of this section is dedicated to introducing the design for a real-time intrusion detection analysis tool based on the state transition analysis technique. This tool, referred to as the State Transition Analysis Tool (STAT), is a rule-based expert system designed to seek out known penetrations in the audit trails of multi-user computer systems. STAT extracts and compares the state transition information recorded within the target system's audit trails to a rule-based representation of the known penetrations specific to that system.

The following is a component-by-component overview of STAT, including a discussion of how STAT is integrated into the architecture of a larger intrusion detection system. STAT is designed modularly, consisting of four major components. An overview of each STAT component is provided. Following this discussion, section IV continues the presentation of the STAT design by describing a UNIX prototype implementation, and section V concludes the discussion of STAT by comparing this approach to other rule-based intrusion detection analysis tools.

D.1 Overview of the State Transition Analysis Tool

As noted previously, STAT is designed to detect the same computer penetrations targeted by currently existing rule-based penetration identification tools. Like these tools, STAT is effective in detecting abuse from misfeasors as well as external attackers. Unfortunately, STAT is also equally ineffective in detecting masqueraders.[8] Thus, when incorporated into an intrusion detection system, STAT is expected to work in combination with another intrusion detection tool that specializes in detecting masqueraders (e.g., a profile-based anomaly detector). Collectively, the two tools will complement each other's coverage, providing the ability to detect both masqueraders and misfeasors.

Figure 5 is a flow diagram illustrating the intended use of STAT as a component within a larger intrusion detection system that also includes a Profile-Based Anomaly Detector. The flow of information begins at the top of the figure, where audit records enter the intrusion detection system, and concludes at the *SSO interface*, where the data is organized and presented to the site security officer. The first step in all intrusion detection systems is the collection of audit data. The *Audit Collection Mechanism* is usually provided as a component of the computer system being analyzed. The audit collection mechanism passes the audit records to both the *audit data archiver/retriever*, for permanent storage, and to the *audit record preprocessor*. The audit data archiver/retriever can be as simple as a buffering mechanism that writes the raw audit data into audit files or as sophisticated as a custom database management system used to store and retrieve audit data.

The audit record preprocessor refers to one or more individual preprocessors used by each intrusion detection com-

[8] However, if the masquerader proceeds to perform a penetration known to STAT, then STAT would detect the penetration, and the actual identity of the penetrator could then be established later.

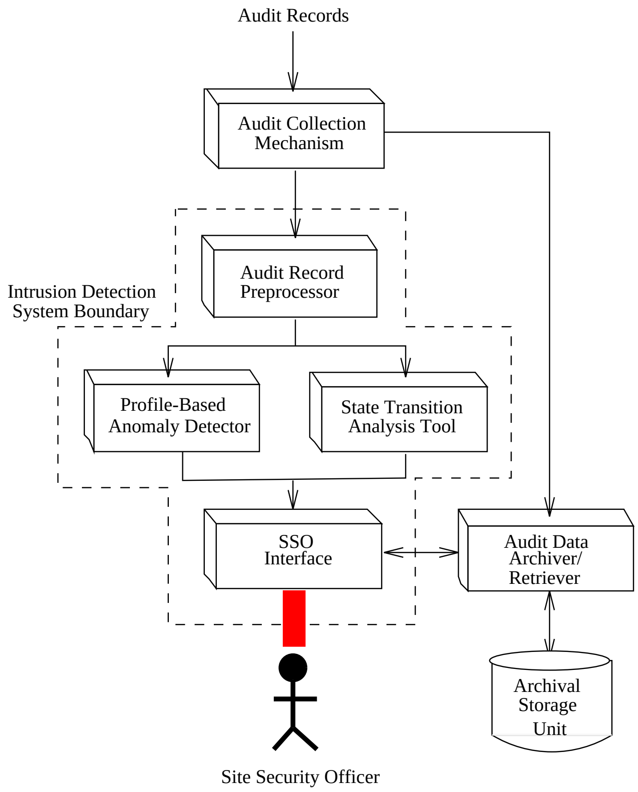

Fig. 5. Organization of Intrusion Detection Components

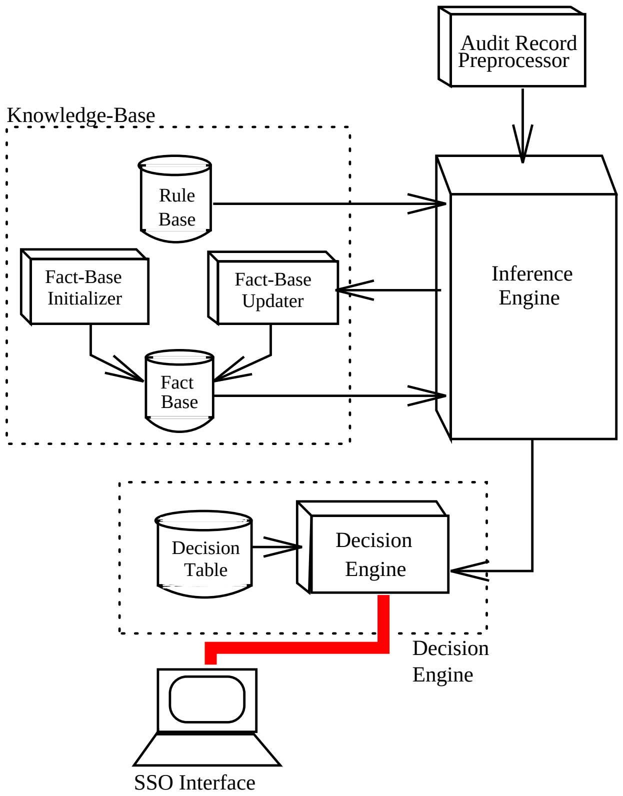

Fig. 6. STAT Component-by-Component Breakdown

ponent to isolate and format certain audit record information prior to its input into the component. In Figure 5, the audit record preprocessor box represents two individual preprocessors: one for the Profile-Based Anomaly Detector and one for STAT. Note that the figure illustrates the architecture of a real-time intrusion detection system; that is, audit records are input to the preprocessor directly from the audit collection mechanism. For batch mode analysis, the audit records would instead be input from the audit data archive.

From the audit record preprocessor, the formatted records are passed to the individual intrusion detection components. Each component independently analyzes the audit records in search of the compromises that it specializes in detecting. The findings of both components are presented to the Site Security Officer via the SSO interface. This interface is important to the overall effectiveness of the intrusion detection system in that it is the sole platform for communication between the intrusion detection system and the security officer. The information must be presented clearly and concisely, for it must be relied upon during security-critical moments. The interface is used by the security officer to interpret the findings of the intrusion detection components, to submit queries to the components, and to set SSO configurable variables within the intrusion detection components.[9] The SSO interface may also provide the security officer with direct access to raw audit records via the audit data archiver/retriever.

The STAT components, and their inter-relationships, are illustrated in Figure 6. STAT is composed of a knowledge-base, an inference engine and a decision engine. It also interfaces with two external components: the STAT Audit Record Preprocessor and the SSO interface.

Analysis begins with the *Audit Record Preprocessor*, which reformats the raw audit records and inputs them into the *inference engine*, the center piece of the entire system. The inference engine monitors the state transitions extracted from the reformatted audit records, and compares these state transitions to the state transition scenarios represented within the *knowledge-base*. The *decision engine* monitors the progress of the inference engine as state transitions represented within the audit trail are found to match the state transition diagrams represented in the knowledge-base. The decision engine specifies the action(s) to be taken based on the inference engine's findings. The details of each of these components are presented in the following subsections.

D.2 The STAT Audit Record Preprocessor

STAT avoids using audit records to represent penetrations within its knowledge-base. Instead, the STAT inference engine uses the audit records as a means of monitoring penetration-relevant state changes that occur within the system. By focusing its analysis on state changes, rather than pattern matching raw audit records to penetration scenarios, STAT is able to detect greater variations within

[9] The STAT design does not currently support query capability or SSO configurable variables (See section VI).

penetrations than can be detected with conventional rule-based penetration identification tools.

Raw audit records are input into the audit record preprocessor where they are filtered and reformatted. Only the audited information relevant to state transition analysis is passed to the inference engine for analysis. The reformatted records correspond directly to the key, or *signature actions*, that are used to represent state transition diagrams. Therefore, different audit records could potentially represent a single signature action.

Signature actions are stated in the form "a subject performs an action [on an object]," where "[on an object]" is optional. The audit record format is:

<*Subject ID, Subject Permissions, Action, Object ID, Object Owner, Object Permissions*>

Table II provides a brief description of each of the fields. Of the six fields comprising the STAT audit record format, three are specified within the guidelines on audit facilities for C2 and above rated systems [25]. These are Subject ID, Action, and Object ID.[10] The remaining three fields are derivable from the information contained within the other three. For example, given the Object ID, the preprocessor can search the object's inode or access control list to determine the Object Owner and Object Permissions fields.

D.3 The STAT Knowledge-Base

The STAT knowledge-base refers to that portion of STAT that collects and integrates all facts regarding STAT's execution environment and its inference rules for detecting penetrations. The modularity added by the knowledge-base is particularly beneficial to STAT in that penetrations tend to be system-specific, and are often version-specific as well. The STAT knowledge-base is comprised of two major subcomponents that contain facts and inference rules. Facts are stored in a *fact-base*, and rules are stored in a *rule-base*. Facts refer to statements that the expert system holds as true regarding its current execution domain. Rules refer to those steps used by the system to infer new facts from the analysis of the current set of facts combined with the input data.

The STAT rule-base consists of rule versions of the state transition scenarios for all known penetrations on the target system. STAT rules are used to establish a relationship between the user actions that are monitored by the inference engine and the state changes that signify an attack scenario in progress. All of the rules associated with a particular penetration are referred to as a *rule chain*. Each state transition in a state transition diagram has a corresponding rule in the STAT rule-base.

STAT rules represent both the signature actions and state descriptions corresponding to each penetration's state transition diagram. Rules are composed of three fields: a state description field, a signature action field, and a rule dependence field. The state description field contains the state assertions of a state within the state transition diagram. The signature action field contains the signature action that leads from the state represented in the state description field to the next state in the state transition diagram. The rule dependence field is used to define an ordering among the states and signature actions.

The rule dependence field lists all other rules in the chain that must fire before the rule's own state transition can occur. For example, the first rule of Penetration 1 (see Figures 1 through 4) is not dependent on any previous state transition. Thus, the rule dependence field for this rule will be empty denoting that the first state transition is independent of all other state transitions. In fact, the first rule of each penetration will have an empty rule dependence field, since there is no state transition required to precede the first state transition. The rule dependence field defines an ordering to a rule chain. The rule dependence field allows a single rule chain to represent multiple variations of a penetration scenario. This is discussed further in section VI-D.

The STAT fact-base is developed in recognition that certain objects within a system sometimes share specific characteristics that make them vulnerable to the same attacks.[11] During the investigation of various attack scenarios, it was realized that most scenarios are applicable to more than one particular file. Instead of duplicating the scenario for each possible file, files that share common characteristics are grouped together. These characteristics are usually attributes that are not kept by the file system (and therefore not provided by the audit records), but rather the attributes that are assigned to the files by the users of the system, e.g., system files, which can be identified by looking at the directories where they are located.

The STAT fact-base contains system-specific information that is relevant to the success of the penetrations represented within the rule-base. It consists of groups of objects (usually files or directories), which are called *filesets*. These filesets are used by the rule-base to increase the generality of the penetration rules. By referencing a fileset rather than a specific file, the rules for a particular penetration are applicable to all files in the fileset, rather than to one particular file.

At STAT initialization time, the identity of each object possessing these attributes is recorded in the fact-base. As one might expect, object attributes are not constant and thus facts within the fact-base are subject to as much change as the objects themselves. Accordingly, STAT employs both a *fact-base initializer* and *fact-base updater* to reconstruct the fact-base upon each start-up and to periodically update the fact-base as the states of objects change.

D.4 The STAT Inference Engine

The code that controls the inference and search procedures of an expert system is consolidated into one logical component, referred to as the inference engine. The STAT

[10] The TCSEC audit guideline for C2 and above rated systems is used here as an example standard. The audit record fields mentioned above are provided by many unevaluated products as well.

[11] This notion is not unique to STAT; it has also been used by COPS [7].

Subject ID	The unique identifier of the subject on whose behalf the audit record was generated.
Subject Permissions	The access privileges of the subject responsible for the audit record (e.g. security level, effective UID, group membership, capabilities, etc.).
Action	The action performed by the subject [on the object]. Possible actions are object read, write, create, or delete, program execute, program exit, modify object owner, modify object permission, modify access privilege, and login.
Object ID	The unique identifier or name of the object whose access was recorded. If no object access occurred, then this field is null.
Object Owner	The owner of the object indicated within the previous field. If no object access occurred, then this field is null.
Object Permissions	The access permissions associated with the object. If no object access occurred, then this field is null.

TABLE II
STAT Audit Record Format

inference engine consists of all the algorithms used to monitor and compare the state changes occurring within the target system to the state transition representations contained in the knowledge-base.

The inference engine maintains four interfaces: the audit record preprocessor interface, fact-base interface, rule-base interface, and the decision engine interface. The audit record preprocessor interface allows the preprocessor to pass formatted audit records to the inference engine. Upon receiving each audit record, the inference engine converts the action and object fields into signature action format and determines what state changes are recorded within the audit record. This information is then compared to the state transitions represented within the knowledge-base.

The inference engine maintains two interfaces into the knowledge-base. One interface is between the inference engine and the *fact-base updater*. Facts, as defined in this environment, are dynamic, and require an algorithm capable of updating the fact-base at run time. When audit records are received that record the deletion, modification or creation of an object, the inference engine passes the object's identity to the fact-base updater, which determines if any filesets in the fact-base require modification. The second interface is between the inference engine and the rule-base. Here, the inference engine reads in all rule chains from the rule-base during initialization and generates an internal representation of the rule chains, which identifies all acceptable orders in which the penetration rules may be fired.

Lastly, the inference engine also maintains an interface to the decision engine. This communication link is not direct, but takes place via a fired rule list, which is stored in a shared memory structure that is accessible to both the inference engine and the decision engine. The decision engine monitors the fired rules of the inference engine and responds to each fired rule.

D.5 The STAT Decision Engine

Decision engines add to the modular design of expert systems by providing greater flexibility in customizing and modifying the responses produced by the expert system, without affecting the inference engine or knowledge-base. STAT's decision engine is responsible for deciding what course of action shall be taken in response to the inference engine firing rules. Possible actions include warning the security officer, preempting the execution of the next signature action that will lead to the firing of the next eligible rule or ignoring the newly fired rule.

Decision engines can be interchanged depending on the environment in which STAT runs. For security-critical audit analysis, the decision engine may specify the actions that must be taken to preempt the transition of the next eligible rule. For less security-critical environments that do not require preemption, the decision engine could instead provide warnings to the SSO interface that describe the danger. STAT might also be placed in a surveillance mode where the decision engine becomes more sensitive to certain subjects that fire rules than it is toward other subjects. Surveillance mode could be configurable or could be linked to the operation of a profile-based anomaly detection component, which might increase the decision engine's sensitivity to subjects whose activities generate a high number of anomaly reports.

IV. A Unix Prototype State Transition Analysis Tool

This section describes the design and implementation of the first prototype of STAT. This prototype, which is called USTAT, is for SunOS 4.1.1. This section concentrates on how USTAT stores the intrusion scenarios and uses them to detect penetrations.

USTAT is designed to be a real-time system that is able to preempt an attack before any damage is done. The major issue in real-time analysis, however, is whether intrusion detection systems in general will be fast enough to catch up with the audit records when the user load is high. The results of several tests focusing on this issue are given in section IV-E.

USTAT uses the audit collection mechanism that exists as an add-on package to SunOS 4.1.1, called C2-BSM (Basic Security Module), [12] which provides improved security features over standard UNIX operating systems, such as shadow password files, object reuse, device allocation/deallocation and an audit mechanism. In the remainder of this paper, the SunOS 4.1.1 C2 Basic Security Module is referred to as BSM.

The components of USTAT are exactly the STAT components that were presented in Figure 6. The specifics of these components to USTAT are discussed in sections IV-A through IV-D. Section IV-E presents the results of testing USTAT.

A. The Audit Record Preprocessor

The audit record preprocessor is responsible for reading, filtering, mapping and finally passing the BSM audit records to the inference engine in the format that is required by USTAT. The preprocessor provides the inference engine with a generic audit record format. It also enables the SSO to create the state transition diagrams with the abstraction of USTAT action names instead of BSM-specific event names. Of the 239 different events that are audited by the BSM only 28 are used by the USTAT preprocessor, and they are mapped onto 10 different USTAT actions: Read, Write, Create, Delete, Execute, Exit, Modify_Owner, Modify_Perm, Rename and Hardlink. The inference engine operates using these 10 action types. For instance, the BSM action open_rw (open for read and write) is mapped onto USTAT actions as Read or Write. That is, any signature action in USTAT's rule-base that indicates Read or Write will match to this action. Table III lists the 10 different actions of USTAT along with the BSM event types that are mapped onto them.

The preprocessor also takes the return value of an event into account. It filters out all the BSM records that indicate a return value of -1, which indicates that the call made by the user did not finish successfully. It did not make any change to the system attributes, and therefore, it cannot cause a state transition.[13]

As per the STAT design, the USTAT audit record structure is defined by the triple:

<SUBJECT, ACTION, OBJECT>.

Each of these attributes contains subfields, which are used to reveal as much information as possible about the particular attribute. The subfield structure is shown in Figure 7. Note that this is even more of a refinement than for the initial STAT design of section III-D.2.

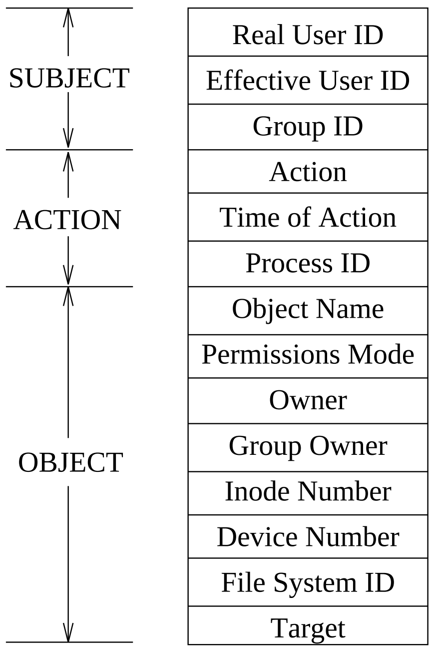

Fig. 7. USTAT Audit Record Structure

Although there are various object types on a target system, (e.g. processes, semaphores, shared memory, files, etc.) the *Object Name* for USTAT is the name of a file identified with its full path. The *Target* field is effective only if the action is hardlink or rename. The *Real User Id* reflects the user id that is associated with the process responsible for the action. The *Effective User Id* is the same as the real user id except when the user executes a setuid program[14] or issues an *su*(1) command. All of the fields in a USTAT audit record can be obtained directly from the BSM audit records. For an in-depth discussion of the BSM features and audit records with respect to USTAT, refer to [14].

B. The Knowledge-Base

As was discussed in section III, the knowledge-base has two components: the fact-base and the rule-base. The fact-base contains information about the objects of the system, and the rule-base is the rule representation of the state transition diagrams.

B.1 The Fact-Base

USTAT's fact-base consists of groups of files or directories (called filesets) that share certain characteristics that make them vulnerable to certain types of attack scenarios. As discussed in the STAT design, these filesets provide a very convenient way of generalizing the penetration scenarios. In this section the characteristics of each of the USTAT filesets are described.

[12] The C2-BSM is designed to be compliant with the TCSEC requirements for a system at the C2 classification [32].

[13] Unlike USTAT, statistical anomaly detection systems use the return field to detect browsers who perform abnormally high numbers of unsuccessful attempts or external attackers who repeatedly fail to pass logon authentication.

[14] These programs (or scripts) temporarily change the user's effective user id to the owner of the program, while the real user id remains unchanged.

USTAT Action	BSM Event Types
Read	open_r, open_rc, open_rtc, open_rwc, open_rwtc, open_rt, open_rw, open_rwt
Write	truncate, ftruncate, creat, open_rwc, open_rwtc, open_rw, open_rwt, open_rt, open_rtc, open_w, open_wt, open_wc, open_wtc
Create	mkdir, creat, open_rc, open_rtc, open_rwc, open_rwtc, open_wc, open_wtc, mknod
Delete	rmdir, unlink
Execute	exec, execve
Exit	exit
Modify_Owner	chown, fchown
Modify_Perm	chmod, fchmod
Rename	rename
Hardlink	link

TABLE III
USTAT Actions vs. BSM Event Types

Initializing the fact-base for USTAT is done by the fact-base initializer module and by some additional manual processing. The fact-base updater consists of routines that keep the fact-base up-to-date for the consistent operation of the inference engine. The current version of the fact-base used by USTAT is given in Table IV.

The first six filesets are used directly in state assertions, whereas the last one is used by the inference engine to identify variations of scenarios through references by hardlinks.

FILESET	**CHARACTERISTICS**
Fileset #1	Restricted read files
Fileset #2	Restricted write files
Fileset #3	Files authorized to read Fileset #1
Fileset #4	Files authorized to write Fileset #2
Fileset #5	Non-writable system executables
NWSD	Non-writable system directories
HARDLINK	System hardlink information

TABLE IV
USTAT Filesets

- The files in Fileset #1, restricted read files, should not be accessed via arbitrary utilities, because they contain sensitive information that if read by an ordinary user could compromise the system security. In some UNIX systems these files are left readable by everyone. For instance, Discolo in [6] illustrates how plaintext passwords can be obtained from `/dev/kmem`. In recognition of this vulnerability and the potential for similar ones, USTAT makes use of Fileset #1. The word "restricted" is used because there are certain system programs that need to read these files for proper execution. These programs are identified in Fileset #3.
- Similar to Fileset #1, files in Fileset #2 should not be modified except by the programs in Fileset #4. A typical example of a member of Fileset #2 is `/etc/passwd`. This file should not be overwritten by any program, except by the passwd(1) command. The password program provides legitimate write access to the `/etc/passwd` file. Any other write to this file (except by the super-user) should be identified as a security violation.
- Fileset #3 consists of the files that are authorized to read files in Fileset #1.
- Fileset #4 consists of those files that are authorized to write on files in Fileset #2. For instance, the *passwd* command has a legitimate need to write to the /etc/passwd file, and it can be used by any user. Therefore, it should be included in Fileset #4.
- Fileset #5 consists of publicly accessible, executable system files that are commonly subjected to Trojan horse attacks. The files in Fileset #5 should not be deleted because of the denial of service problem, nor should they be overwritten (except by the system administrator) because of a possible Trojan horse implantation.
- The idea of the non-writable system directories is similar to the idea of non-writable system executables. These directories usually consist of publicly executable

files and therefore they are also subject to Trojan horse attacks. For example, if somebody creates a fake *ls*(1) program in one of these directories, it is possible that in the victim's path, the name of the target directory comes before the directory where the actual *ls*(1) program is located. Therefore, write access to all of these directories should be denied, unless the access is by root. NWSD contains the names of these directories.

- In UNIX, one physical file may have several pathnames associated with it. Processes can access the file by any of these pathnames. In analyzing penetration scenarios, it has been noticed that variations of the scenarios can easily be accomplished by using different filenames at different steps of the penetration, while still referring to the same physical file. In this case, the inference engine would fail in firing the rule of a penetration scenario, because the object in the next step was not identical to the object in the previous step. To overcome this, USTAT's fact-base keeps information about all hardlinks on the target system in the HARDLINK fileset.

B.2 The Rule-Base

The USTAT rule-base stores state transition information in two text files referred to as the *State Description Table* and the *Signature Action Table*, which store the state assertions and the signature actions, respectively. The inference engine uses this information to match the actions of incoming USTAT audit records to the actions of state transition scenarios.

In the state transition diagrams, each state consists of one or more state assertions, where each state assertion consists of a function name with zero or more arguments. The evaluation of a state assertion results in a *true* or *false* value, and the keyword *not* in front of a state assertion negates the result. The following is a short description of each of the state assertions:

1. **name** (*file_var*) = *file_name*
   Evaluates true if the file_var matches the filename given in the right-hand side.
2. **fullname** (*file_var*) = *full_path*
   Evaluates true if the file_var matches the pathname given in the right-hand side.
3. **owner** (*file_var*) = *user_id*
   Evaluates true if the owner of *file_var* is the *user_id*.
4. **member** (*file_set, file_var*)
   Evaluates true if *file_var* is a member of the *file_set*.
5. **euid** = *user_id*
   Evaluates true if the effective user id of the subject of the audit record being processed equals the *user_id*.
6. **gid** = *group_id*
   Evaluates true if the group id of the subject of the audit record being processed equals the *group_id*.
7. **permitted** (*perm, file_var*)
   Evaluates true if the permission bit given as *perm* is set in *file_var*'s permission bits.
8. **located** (NWSD, *file_var*)
   Evaluates true, if *file_var* is located in any of the directories listed in the NWSD fileset.
9. **same_user**
   Evaluates true if the subjects of the last two signature actions are the same.
10. **same_pid**
    Evaluates true if the process id's of the last two signature actions are the same.
11. **shell_script** (*file_var*)
    Evaluates true if the *file_var* is a *shell_script* with the #!/bin/sh mechanism.

Similarly, the signature actions can be one of the following. These correspond to the action types used by the preprocessor.

1. **read** (*file_var*)
2. **write** (*file_var*)
3. **create** (*file_var*)
4. **execute** (*file_var*)
5. **exit** (*file_var*)
6. **delete** (*file_var*)
7. **modify_owner** (*file_var*)
8. **modify_perm** (*file_var*)
9. **hardlink** (*file_var, file_var*)
10. **rename** (*file_var, file_var*)

C. The Inference Engine

USTAT's inference engine uses an event driven, forward chaining inference scheme [21]. The inference engine uses a structure called the *inference engine table* to keep track of all possible penetration scenarios. At any point in time, this table consists of snapshots of penetration scenario instances (instantiations) that are not yet completed on the target system. Each row represents one instance of a possible penetration scenario. Each column corresponds to one state of a scenario and it depicts how far a compromise is from being completed. Each row also contains information about the history of the instantiation, such as the users involved, files involved, related audit records, etc. Table V illustrates the initial configuration of the inference engine table. Assuming n is the number of different state transition scenarios in the state description table, then there is one row for each state transition scenario. Initially there is no mark on any row, which means the initial signature action of each scenario is being anticipated by the inference engine. Whenever the inference engine detects an audit record that matches the next action and satisfies the next state of a state transition scenario, it duplicates the row and "marks" the corresponding cell on the duplicated row (marking a row is equivalent to firing a rule and also equivalent to a state transition). The reason for duplicating the row is that the original row still represents part of another instantiation. The inference engine anticipates the next unmarked cell (next signature action) of a row.

To illustrate the manipulation of the table, an example that uses a hypothetical state transition diagram, called STD_h (see Figure 8), will be used. Note that the penetration represented by this diagram is lacking the prerequisite

	S_1	S_2	S_3	S_4	S_5
1					
2					
. .					
h					
. .					
n					

TABLE V
Initial inference engine table

	S_1	S_2	S_3	S_4	S_5
1					
2					
. .					
h					
. .					
n+1	X				

TABLE VI
Inference engine table after action 1

initial state as described in section III-A. This is because the only prerequisite to performing this penetration is that the attacker must have standard UNIX access. In USTAT, a Null state is assumed to exist at the beginning of all of the state transition diagrams.

The STD_h diagram contains three signature actions and three state assertions. The contents of the state assertions are left out for simplicity. The state that follows the last signature action depicts the final compromised state. From this diagram one can observe that there may be many executions of *file1* by possibly different users once it has been created.

The hypothetical state transition diagram (STD_h) corresponds to Row h of the initial inference engine table. Now, consider the effects of each action on the inference engine table.

Action 1: User A creates file1 and the state assertions in S_1 of STD_h are satisfied.

Result: Row h is duplicated and S_1 is marked on the new row. (See Table VI).

At this point there are $n+1$ different signature actions that are being anticipated by the inference engine, one for each row. Two of these are related to STD_h. These are the creation of another file and the execution of file1, which correspond to the two rows (h) and ($n+1$) of Table VI, respectively.

Action 2: User B executes file1 and S_2 of STD_h is satisfied.

Result: Row $n+1$ is duplicated and S_2 is marked on the new row. This result should not affect row $n+1$ since the same file can still be executed by another user. The result of the second action is illustrated in Table VII.

	S_1	S_2	S_3	S_4	S_5
1					
2					
. .					
h					
. .					
n					
n+1	X				
n+2	X	X			

TABLE VII
Inference engine table after action 2

Action 3: User B reads file2 and S_3 of STD_h is satisfied.

Result: Since S_3 is the final state of STD_h a compromise has been achieved. No new row is added to the table. As a result the table will still look like Table VII. However, the compromise is reported to the decision engine.

A row is deleted from the table only when the corresponding state assertions no longer hold. For example, row $n+1$ is deleted only when file1 is deleted or when the state assertions in S_1 of STD_h no longer hold for row $n+1$.

Note that two different attackers were involved in the example scenario: users A and B. The inference engine applies the rules of the state transition diagram regardless of the identity of the attacker. That is, the attack of cooperating users and a single user makes no difference to USTAT. The identity of the attacker is important only if the state assertions of a certain state include the assertion "same_user" (see section IV-B.2). In this case the inference engine compares the identity of the user for the last two signature actions of the scenario.

Similarly, the inference engine applies the rules in the rule-base regardless of the login sessions. The amount of time passed between two signature actions has no effect on USTAT's inference mechanism. For instance, the attack scenario of Figure 8 can be performed in two login sessions by performing the first action (create file1) in one login session, and the second and third actions in a later login session. USTAT's inference engine retains the row of the inference engine table that indicated the creation of file1 as long as file1 satisfies the assertions in State Assertions-1. This way, it makes no difference if the second action is performed in another login session.

D. The Decision Engine

The decision engine is responsible for informing the SSO about the results of the inference engine activities. The output of the decision engine could be one or more of the

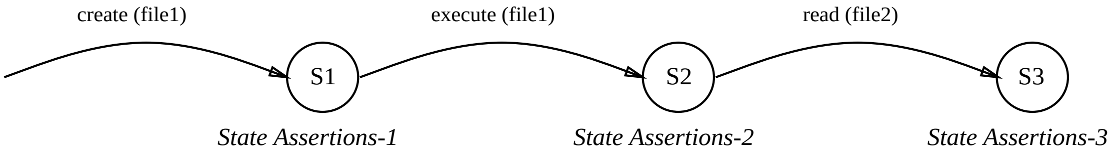

Fig. 8. STD_h: A Hypothetical State Transition Diagram

following actions, which are ranked from the simplest to the most complicated.

- Inform the SSO that a compromise has been achieved.
- Inform the SSO whenever a state of any instance of the scenarios has been satisfied.
- Suggest possible actions to the SSO to preempt a state transition that can lead to a compromised state.
- Play an active role in preempting the attack.

The decision engine implemented for the USTAT prototype performs the first three.

The decision engine employs a structure called the *decision table*, which contains a decision message for each state of a state transition scenario. Whenever the decision engine is notified by the inference engine about a state change it displays the following information to the console of the machine running USTAT.

- the number of the state transition scenario for which this state change has occurred
- the message in the decision table that corresponds to the last satisfied state
- all of the filenames that were involved in this instance of the scenario
- the real user id and the effective user id of the user who performed the last signature action for this instantiation

Figure 9 presents a sample of the output generated by the USTAT decision engine. With this data, the SSO can foresee an impending compromise and has enough information to either take preemptive action, or to take precautions to prevent future attacks of the same nature.

E. Testing USTAT

Because USTAT was the first STAT prototype it was important to run performance tests to assess the real-time capabilities of this approach. The results of both the functional and performance tests are summarized in this section. All tests were run on a SPARCstation 1.

E.1 Functional Testing

Functional tests were performed on three different features of USTAT.

1. The rules representing the state transition diagrams: Do all of the rules work when their penetration scenarios are performed? Preliminary tests showed that all attack scenarios defined in the rule-base are detected by USTAT.
2. The hardlink information: Do the rules work when the attack is performed using hardlinks? For instance, the first part of a scenario can be performed by one file, whereas the second part can be performed by a hardlink to the first file. Test results showed that variations to the scenarios performed through hardlinks are detected by USTAT.
3. Cooperative attacks and multi-session attacks: Do the rules work if parts of a scenario are performed by different attackers or by the same attacker in two different login sessions? One of the attack scenarios defined in USTAT's rule-base allows two attackers to cooperate and allows the user to perform the actions in different login sessions. Tests indicated that USTAT is able to detect such variations of attack scenarios.

E.2 Performance Testing

The purpose of the performance testing was to get some feedback on the processing speed of USTAT and also about the change in the performance of USTAT when there are other processes running on the same system. USTAT was tested with a combination of audit intensive [15] and cpu intensive processes. *ps*(1) was run periodically to obtain the cpu and disk utilization of the processes. It is obvious that *ps*(1) also used some cpu cycles and therefore affected the test results.

In these tests, USTAT was processing audit data that had been collected prior to the test. The audit data was mostly generated by one user (occasionally by a few users) logging on the target machine. It could be considered a batch-mode execution, but it became real-time when USTAT caught up with the current audit data.

The following processes were used in the tests:

- USTAT: Both cpu and I/O (primarily input) intensive. USTAT itself is not audited since otherwise it would result in circular activity.

[15] Audit intensive processes perform a large amount of system calls that keep the audit daemon and hence the disk where the audit records are written, continuously busy.

```
                ********** STD$4 **********
Warning: User executed lpr to print FILE2 owned by somebody else.
If lpr invoked with -r FILE2 will be removed illegally.

FILE1: /usr/ucb/lpr
FILE2: /etc/ttytab
USER: 5502 EUID: 0
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              4643 / 700000 - Tue Jul 28 23:51:11 1992
ACTION:                  EXEC
OBJECT:                  /usr/ucb/lpr
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   1   -rws--s--x
DEVICE - INODE - FSID:   7392  3080  1798
--------------------------------------------------
SUBJECT (RUID-EUID-GID): 5502 - 0 - 24
PID / Time:              4642 / 20000 - Tue Jul 28 23:51:12 1992
ACTION:                  READ
OBJECT:                  /etc/ttytab
TARGET:                  (null)
OWNER - GOWNER - PERM:   0   10   -rw-r--r--
DEVICE - INODE - FSID    208  31  1792
```

Fig. 9. Sample Output from Decision Engine

- Crack: A cpu intensive program that runs encryption routines to guess passwords.
- *find*(1): An audit intensive system program. When run on a large filesystem (e.g., find / -print) it creates a enormous amount of audit data.
- Audit daemon (*auditd*(8)): This process becomes active whenever BSM is installed on the target machine. It controls the generation and location of the audit trail files. It is a highly I/O intensive (primarily output) process. The audit daemon was in the test by default, since USTAT and the audit daemon were running on the same machine.

Tables VIII through X summarize the results of the test cases. The first column of the table lists the processes involved in the test. The second column indicates the amount of cpu time (hours:minutes:seconds) used, and the third column indicates the percentage of cpu time used by each process.

Test case 1: USTAT only

In this first test USTAT's performance was tested while no other major processes were running. USTAT processed the audit data for about 40 minutes. After this point it became idle and waited for more audit data. More than 30 minutes (> 50 %) of the cpu time was idle even during USTAT's processing (see Table VIII). This indicates that for half its runtime USTAT was waiting for the I/O to be completed.

Test case 2: USTAT and crack

Process	Cpu time	Percentage
Auditd	0:05	0.1
USTAT	32:17	45
Idle[16]	38:56	55
Total	1:11:18	100

TABLE VIII
Cpu usages of auditd and USTAT

Because crack is a cpu intensive process, in this test run the cpu was rarely idle. Both USTAT and crack are user processes, and both competed for cpu time. Compared to the previous test case, in this one crack seemed to fill the unused cpu cycles (see crack's cpu usage in Table IX). The test results also showed that the cpu utilization of USTAT dropped by 5 %.

Test case 3: USTAT and find(1)

Being an audit intensive process, *find*(1) made a tremendous amount of calls that pushed the audit daemon to its limit. Since the audit daemon runs as a root process it caused a considerable slow-down of the user processes. In addition, the processes that required disk I/O from the same disk that was used by the audit daemon, were very likely to hang. For instance, an *ls* command on the audit data directory hung. Since USTAT is also a user process requiring disk I/O from the audit disk, it also hung after a

Process	Cpu time	Percentage
Auditd	0:06	0.3
USTAT	14:51	43
Crack	14:01	40
Idle	5:58	17
Total	34:56	100

TABLE IX
Cpu usages of auditd, USTAT and crack

while. The only solution to this was to terminate the execution of *find*(1), thereby relieving the audit daemon and giving USTAT some opportunity to read the audit data. In this test, USTAT showed more than a 50 % slow-down in processing speed. More than 60 % of the cpu time was idle, since USTAT didn't have much opportunity to process audit data (see Table X).

Process	Cpu time	Percentage
Auditd	6:16	12
USTAT	10:24	19
find(1)	2:36	5
Idle	34:14	64
Total	53:30	100

TABLE X
Cpu usages of auditd, USTAT and find(1)

E.3 Remarks About the Tests

In running the tests both the effectiveness and the efficiency of USTAT were observed. These tests were meant to be a feasibility check rather than to be exhaustive. The actual performance of USTAT will be more apparent after running it on different target systems with different configurations.

So far, the limiting factor appears to be the throughput of the disk that is extensively used by both USTAT and the audit daemon. Cpu intensive processes that run on the same machine as USTAT have little effect on the performance of USTAT. The tests showed that if the cpu intensive process is a user process (not a root process), approximately a 13 % slowdown is experienced in USTAT's processing speed (see test case 2). One possibility for increasing the performance of USTAT is to run USTAT on a dedicated machine (IDES [16] uses this approach.). However, if USTAT and the audit daemon need to be run on the same computer, USTAT can be run as a higher priority process. Although this would speed up USTAT's processing, it would likely cause the audit daemon to fall even further behind than observed for test case 3. Another solution would be to have the audit daemon periodically switch between different disks, so that USTAT will not starve while waiting for a particular disk (see test case 3). Perhaps the best solution would be to modify the audit daemon's source code to redirect its output to USTAT. This would eliminate all of the disk speed limitations and thereby drastically increase USTAT's performance.

V. Comparison of STAT to Current Rule-Based Intrusion Detection Tools

A key objective of this research was to understand the weaknesses of current penetration identification tools and to address these weaknesses through the development and implementation of a new, and hopefully better, analysis method. Section II-F discussed several weaknesses inherent within the design of current rule-based penetration identification tools, which included the following:

- the dependence on audit records for representing penetrations within the rule-base,
- the non-intuitiveness of penetration rules, making rule-base updating difficult for all but the experienced knowledge-base programmer,
- the inability to identify even semi-sophisticated attacks or to detect impending compromises, and
- the inability to detect a penetration until after the compromised state has been reached.

The three example penetration identification tools introduced in section II (IDES, W&S, and NADIR) maintain rule-bases that represent suspicious user activity as well as rules to identify site-specific policy violations. Each of these implementations is audit-record dependent; that is, the tools represent illicit behavior through rules that are based specifically on the fields within the audit records themselves. Unfortunately, this dependence on the audit records results in several weaknesses. First, penetrations that are traceable via an audit trail (i.e., that are subject to intrusion detection) are formulated at the user interface level, where the attacker executes a coordinated set of commands/program instructions — not at the audit level. Current penetration identification tools require the rule-base programmer to first translate a scenario into an audit trail before generating the rule chain. The resulting rule chain will only be fired if there is a direct correspondence, in both the order and content, between the audit record fields and the penetration rules. As a result, even a minor variation within a penetration audit trail may be enough to prevent its detection by IDES, W&S or NADIR.

STAT, on the other hand, focuses on a penetration's signature actions rather than the audit records that record the actions. Furthermore, STAT rules are designed to support the representation of interdependencies among the actions within the penetration scenarios, allowing one rule chain to represent multiple variations in the order in which a penetration's actions may be performed. By providing the ability to support multiple variations within a single penetration rule chain, STAT increases its flexibility in identifying penetrations that would otherwise be missed by other rule-based tools.

Another important advantage of using signature actions rather than audit records to represent penetrations is that they result in more intuitive rule chains. In W&S for example, the rule-base programmer interviews system administrators and security experts to collect a suite of known penetration scenarios and key events that threaten the security of the target system [34]. The programmer must then identify what audit records will be produced by the key event or scenario, and build a rule chain based on the expected audit trail. Thus, to understand the rule chain, one must reconstruct the audit trail from the rules and the scenario from the audit trail. In general, expert rule-bases are non intuitive, requiring the expertise of an experienced rule-base programmer to construct the rules that accurately represent the concepts for which they are intended and, at the same time, do not contradict other rules within the rule-base.

In contrast, the STAT rule-base is specifically intended not to require the use of an expert rule-base programmer, but to be readable and updatable by site security officers and system administrators. State transition diagrams aid in the development of penetration rule chains, in that they provide an analyst with a visual representation of the key actions that must occur in order for an attacker to move the system from the initial prerequisite state to the compromised state. A major difference between STAT and the other tools is that STAT rule chains are constructed from state transition diagrams. In this sense, the state transition analysis approach is similar to the Model-based Intrusion Detection approach proposed by Garvey and Lunt [8] (see section II-D). Both techniques provide a higher level representation of user behavior (i.e., above the audit record level) providing easier readability and rule generation.

Current penetration identification tools are also limited in their ability to identify even semi-sophisticated attacks, such as those performed by cooperating attackers. The IDES rule-base, for example, does not take into consideration two or more users working together to execute a penetration. STAT addresses this limitation by having its inference engine view the audit trail globally, focusing on the events occurring on the system rather than the actions of individuals. When a sequence of actions is found to match the actions represented in a state transition diagram, STAT then constructs a list of the users who contributed steps within the penetration.

Finally, there is an important benefit that results from the combination of the use of state transition diagrams as a method for building penetration rule chains and STAT's ability to represent permutable penetration state transitions within a single rule chain. The benefit is that STAT supports both the generation as well as the representation of longer rule chains than with current tools. These longer rule chains increase STAT's ability to identify impending compromises, by describing the full sequence of events leading up to the compromise (i.e., an entire penetration scenario). Shorter rule chains, in contrast, are limited to representing the compromise itself, providing only a few rules to represent the point at which the compromised state is reached, and ignoring the sequence of events that led up to the compromise. In current penetration identification tools the analyst chooses a few key audit records *ad hoc*, and builds a rule chain from these records. State transition diagrams, on the other hand, provide STAT users with a modest, but intuitive, procedure for developing and organizing a complete list of the key events that lead up to the compromise. STAT also supports the representation of longer rule chains through its ability to represent permutable state transitions within a single rule chain (see section VI-D).

Table XI summarizes the functionality discussed above, comparing STAT to the functionality of three of the other penetration identification tools presented in section II.

VI. Areas of Future Research

This section summarizes a few areas of future research and enhancements that the Reliable Software Group is planning to investigate.

A. A STAT Knowledge Acquisition Subsystem

One obvious question affecting all penetration rule-base implementations is that of completeness. Penetration rule-bases are only capable of representing those known penetrations that are traceable via audit data analysis. Since one cannot verify that all of the security flaws within a particular target system have been identified, one cannot expect a penetration rule-base to include rule chains for every possible system penetration. Accordingly, the ability to facilitate the process by which security administrators are able to update the STAT rule-base, on-site, with new and variant penetrations is highly desirable.

The issue of incomplete rule-bases is addressed in two ways. First, the STAT rule-base is specifically designed to allow non-experienced rule-base programmers, such as system administrators and security officers, to update the rule-base themselves. Thus, penetration rule chains can be added as the security officer learns of new penetrations. Secondly, work toward the development of a heuristic algorithm for learning new and variant penetrations is in progress. Using this algorithm, STAT will be capable of converting state transitions from previously unknown penetration audit trails into signature actions and of automating the integration of these signature actions into the rule-base to defend against future attacks.

B. STAT Decision Engine

Another desirable addition to STAT would be query support for the security officer via the SSO interface. The most direct way to provide this support is via the interface between the decision engine and the SSO interface. Currently data flows only from the decision engine to the SSO interface. However, the decision engine can be redesigned to allow queries from the SSO interface to the decision engine.

A further enhancement to the decision engine would be to add a surveillance mode capability. For example, specific subject ID's could be provided to the STAT decision engine via the SSO interface. The decision engine could then be acutely sensitive to these subjects and generate a warning message each time one of these subjects fires a

	STAT	IDES	W&S	NADIR
Represents site-specific policies	YES	YES	YES	YES
Detects cooperating attackers	YES	NO	NO	NO
Structured rule development (not ad-hoc)	YES	NO	NO	NO
Rules are audit record independent	YES	NO	NO	NO
Supports permutable event sequences	YES	NO	NO	NO
Supports longer rule chains/ detects impending compromise	YES	NO	NO	NO

TABLE XI
Comparison of STAT to Current Penetration Identification Tools

penetration rule. Surveillance mode could also be implemented via an interface between STAT and a concurrently running anomaly detector. As a subject's suspicion rating increases through anomaly detection, so too could the decision engine's sensitivity to that subject.

C. Network Audit Support

A natural extension to the STAT effort is to run STAT on audit data collected by several hosts. On a network filesystem where the files are distributed on many hosts and where each host mounts directories from the others, actions on each host computer need to be audited. That means an audit mechanism needs to be run on each host. Running an implementation of STAT on each host might result in inefficient use of computer resources. Also, the possibility of having cooperative attacks on different hosts would make the detection difficult. One approach is to provide a single STAT process with a single, chronological audit trail. For example, the *auditreduce(8)*[17] command installed with C2-BSM takes one or more filenames (audit trails) as input and produces a single time-ordered audit trail. This output can then be used as a single audit trail input to STAT's preprocessor.

D. Permutable State Transitions

The USTAT prototype assumes that state transitions are ordered into a single sequence; however, it is possible that two signature actions within a penetration rule chain will be independent of each other. For instance, consider the creation of a file, followed by the creation of a symbolic link to the file. The actions can be stated as follows: Action1: create (foo), Action2: create (bar), where bar is a symbolic link to foo. UNIX allows one to create a symbolic link regardless of whether the file being linked to exists or not. Thus, a compromise that requires the above two actions is achievable regardless of the order in which they are executed.

STAT was originally designed to support permutable state transitions. However, permutable state transitions were not necessary for the scenarios that were used for the development of USTAT, where each scenario corresponds to a single, non-permutable rule sequence. Therefore, permutable sequences were not implemented in the USTAT prototype. A method for implementing permutable rule sequences in USTAT has been designed.

[17]See [32] or the *auditreduce(8)* man pages for more information about auditreduce.

VII. Conclusion

The state transition approach was introduced in an effort to develop an easily readable representation for computer penetrations. This approach models penetrations as a series of state transitions described in terms of signature actions and state assertions. State transition diagrams are written to correspond to the states of an actual computer system, and these diagrams form the basis of a rule-based expert system for detecting penetrations, called STAT.

The state transition analysis approach targets the same penetrations that are identifiable by current rule-based penetration identification tools. The state transition analysis approach, however, offers several key advantages over existing rule-based implementations. Unlike current rule-based analysis tools that pattern match sequences of audit records to the expected audit trails of known penetrations, state transition analysis focuses on the effects that the individual steps of a penetration have on the state of the computer system. This results in an audit record independent rule-base that is easier to read than current penetration rule-bases. It also provides greater flexibility in identifying variations of known penetrations. State transition analysis also provides a modest, but intuitive procedure for rule generation, rather than the *ad hoc* approaches currently in use. Lastly, the state transition analysis approach has the ability to detect cooperating attackers and attacks across user sessions.

The USTAT prototype of STAT was implemented on SunOS 4.1.1 to validate the functional capabilities and conceptional soundness of the state transition analysis approach. The performance tests that were run on the prototype indicate that the limiting factor is the disk throughput and that cpu intensive processes on the same machine have little effect. Several approaches to increase the performance were outlined.

Future work on STAT will concentrate on adapting it to multiple hosts and to other platforms.

References

[1] J.P. Anderson, "Computer Security Threat Monitoring and Surveillance," James P. Anderson Co., Fort Washington, PA, April 1980.

[2] M. Bishop, "Security Problem with the UNIX Operating System," [Restricted Distribution], Department of Computer Science, Purdue University, West Lafayette, IN, April 1982.

[3] K. Chen, S.C. Lu and H.S. Teng, "Adaptive Real-Time Anomaly Detection Using Inductively Generated Sequential Patterns," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 278-295, May 1990.

[4] H. Debar, M. Becker and D. Siboni, "A Neural Network Component for an Intrusion Detection System," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 240-258, May 1992.

[5] D.E. Denning and P.G. Neumann, "Requirements and Model for IDES - A Real-time Intrusion Detection Expert System," Technical Report, CSL, SRI International, August 1985.

[6] A.V. Discolo, "4.2 BSD UNIX Security," Computer Science Dept., University of California, Santa Barbara, April 1985.

[7] D. Farmer and E.H. Spafford, "The COPS Security Checker System," *Proceedings of the Summer 1990 Usenix Conference,* Anaheim, CA, pp. 305-312, June 1990.

[8] T.D. Garvey and T.F. Lunt, "Model-based Intrusion Detection," *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, pp. 372-385, October 1991.

[9] L.R. Halme, T.F. Lunt, J. Van Horne, "Analysis of Computer System Audit Trails - Intrusion Classification," Sytek Technical Report TR-85012, Mountain View, CA, October 1985.

[10] B. Hubbard, T. Haley, N. McAuliffe, L. Schaefer, N. Kelem, D. Wolcott, R. Feiertag and M. Schaefer, "Computer System Intrusion Detection," Trusted Information Systems, Inc., RADC-TR-90-413 Final Technical Report, December 1990.

[11] K.A. Jackson, D.H. DuBois and C.A. Stalling, "An Expert System Application for Network Intrusion Detection," *Proceeding of the 14th National Computer Security Conference*, Baltimore, MD, pp.215-225, October 1991.

[12] H.S. Javitz and A. Valdes, "The SRI IDES Statistical Anomaly Detector," *Proceedings of the IEEE Research in Security and Privacy,* Oakland, CA, pp. 316-376, May 1991.

[13] P. Kerchen, R. Lo, J. Crossley, G. Elkinbard, K. Levitt and R. Olsson, "Static Analysis Virus Detection Tools for UNIX Systems" *Proceedings of the 13th National Computer Security Conference, Baltimore,* MD, pp. 350-365, October 1990.

[14] K. Ilgun, "USTAT: A Real-Time Intrusion Detection System for UNIX," Masters Thesis, Computer Science Department, University of California, Santa Barbara, July 1992.

[15] K. Ilgun, "USTAT: A Real-Time Intrusion Detection System for UNIX," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 16-28, May 1993.

[16] T.F. Lunt, "Automated Audit Trail Analysis and Intrusion Detection: A Survey," *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD, pp. 65-73, October 1988.

[17] T.F. Lunt, "Real-Time Intrusion Detection," *Proceedings COMPCON*, San Francisco, CA, February 1989.

[18] T.F. Lunt, R. Jagannathan, R. Lee, and A. Whitehurst, "Knowledge-Based Intrusion Detection," *Proceeding of the 1989 AI Systems in Government Conference*, pp. 102-107, March 1989.

[19] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalai, H.S. Javitz, A. Valdes, and P.G. Neumann, "A Real-Time Intrusion Detection Expert System," SRI CSL Technical Report, SRI-CSL-90-05, June 1990.

[20] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, P.G. Neumann, H.S. Javitz, A. Valdes and T.D. Garvey, "A Real-time Intrusion Detection Expert System (IDES), Final Technical Report," Computer Science Laboratory, SRI International, Menlo Park, CA, February 1992.

[21] J. Martin and S. Oxman. *Building Expert Systems: A Tutorial*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

[22] N.J. McAuliffe, L.J. Schaefer, D.M. Wolcott, T.K. Haley, N.L. Kelem and B.S. Hubbard, "Is Your Computer Being Misused? A Survey of Current Intrusion Detection System Technology," *Proceedings of the Sixth Computer Security Applications Conference*, pp. 260-272, December 1990.

[23] B.G. Miller and P.E. Proctor, "A Requirements Oriented Analysis of Computer Misuse Detection Systems," presented at the Seventh Intrusion Detection Workshop, SRI International, Menlo Park, CA, May 1991.

[24] National Computer Security Center, *Trusted Computer System Evaluation Criteria,* DoD, DoD 5200.28-STD, December 1985.

[25] National Computer Security Center, *A Guide to Understanding Audit in Trusted Systems,* NCSC-TG-01, Version 2, June 1988.

[26] National Computer Security Center, *Glossary of Computer Security Terms*, NCSC-TG-004, Version 1, October 1988.

[27] P.G. Neumann, "A Comparative Anatomy of Computer System/Network Anomaly Detection Systems," CSL, SRI BN-168, Menlo Park, CA, May 1990.

[28] P.A. Porras and R.A. Kemmerer, "Penetration State Transition Analysis A Rule-Based Intrusion Detection Approach," *Proceedings of the Eighth Annual Computer Security Applications Conference*, San Antonio, Texas, pp. 220-229, December 1992.

[29] P.A. Porras, "STAT - A State Transition Analysis Tool for Intrusion Detection," Masters Thesis, Computer Science Department, University of California, Santa Barbara, June 1992.

[30] M.M. Sebring, E. Shellhouse, M.E. Hanna and R.A. Whitehurst, "Expert System in Intrusion Detection: A Case Study," *Proceedings of the 11th National Computer Security Conference,* Baltimore, MD, pp. 74-81, October 1988.

[31] S.W. Shieh and V.D. Gligor, "A Pattern-Oriented Intrusion Detection Model and Its Application," *Proceedings of the IEEE Research in Security and Privacy*, Oakland, CA, pp. 327-342, May 1991.

[32] Sun Microsystems, Inc., *SunOS Release 4.1.1. C2-BSM Patch, Revision A*, 2550 Garcia Ave, Mountain View, CA 94043, December 15, 1991.

[33] UNIX Programmer's Manual, 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, Department of Electrical and Computer Sciences, University of California, Berkeley, CA, August 1983.

[34] H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 280-289, May 1989.

Koral Ilgun received the B.S. degree in computer engineering from Bogazici University, Istanbul, Turkey and the M.S. degree in computer science from the University of California, Santa Barbara. He is currently working at Advanced Computer Communications as a software design engineer and developing software for the company's bridge/router products.

Richard A. Kemmerer is a Professor and Chair of the Department of Computer Science at the University of California, Santa Barbara. He also leads the Reliable Software Group at UCSB. He has been a Visiting Scientist at the Massachusetts Institute of Technology, and a Visiting Professor at the Wang Institute and the Politecnico di Milano. From 1966 to 1974 he worked as a programmer and systems consultant for North American Rockwell and the Institute of Transportation and Traffic Engineering at UCLA. His research interests include formal specification and verification of systems, computer system security and reliability, programming and specification language design, and software engineering. He is author of the book *Formal Specification and Verification of an Operating System Security Kernel* and co-author of the book *Computer's at Risk: Safe Computing in the Information Age.* Dr. Kemmerer received the B.S. degree in Mathematics from the Pennsylvania State University in 1966, and the M.S. and Ph.D. degrees in Computer Science from the University of California, Los Angeles, in 1976 and 1979, respectively.

Dr. Kemmerer has served as a member of the National Academy of Science's Committee on Computer Security in the DOE (1987-88), the System Security Study Committee (1989-1991), and the Committee for Review of the Oversight Mechanisms for Space Shuttle Flight Software Processes (1992-1993). He has also served as a member of the National Computer Security Center's Formal Verification Working Group and was a member of the NIST's Computer and Telecommunications Security Council. Dr. Kemmerer is also the past Chairman of the IEEE Technical Committee on Security and Privacy and a past member of the Advisory Board for the ACM's Special Interest Group on Security, Audit, and Control. He is a member of the Association for Computing Machinery, a member of the IEEE Computer Society, and a member of the International Association for Cryptologic Research. Dr. Kemmerer also serves on the editorial boards of the *IEEE Transactions on Software Engineering* and the *ACM Computing Surveys*.

Phillip A. Porras received the B.S. degree in information and computer science from the University of California, Irvine, and the M.S. degree in computer science from the University of California, Santa Barbara. Mr. Porras is currently with the Trusted Computer Systems Department of The Aerospace Corporation. He is an evaluator for the National Security Agency, Trusted Product Evaluation Program, and performs security analyses in support of various government projects. His research interests include intrusion detection, information flow, secure systems design, and hardware assurance.