USTAT: A Real-time Intrusion Detection System for UNIX

Koral Ilgun *
Reliable Software Group
Dept. of Computer Science
University of California
Santa Barbara, CA 93106

Abstract

This paper presents the design and implementation of a real-time intrusion detection tool, called USTAT, *a State Transition Analysis Tool for UNIX. This is a UNIX-specific implementation of a generic design developed by Phillip A. Porras and presented in [Porr92B] as STAT, State Transition Analysis Tool. State Transition Analysis is a new approach to representing computer penetrations. In STAT, a penetration is identified as a sequence of state changes that take the computer system from some initial state to a target compromised state.*

In this paper, the development of the first USTAT *prototype, which is for SunOS 4.1.1, is discussed.* USTAT *makes use of the audit trails that are collected by the C2 Basic Security Module of SunOS, and it keeps track of only those critical actions that must occur for the successful completion of the penetration. This approach differs from other rule-based penetration identification tools that pattern match sequences of audit records.*

1 Introduction

Most computer systems are vulnerable to two different groups of attacks: Insider attacks and outsider attacks. A system that is known to be secure to an outsider attack by preventing access from outside can still be vulnerable to the insider attacks accomplished by abusive usage of authorized users. Detecting such abusive usage as well as attacks by outsiders not only provides information on damage assessment, but also helps to prevent future attacks. These attacks are usually detected by tools referred to as *Intrusion Detection Systems.*

The most popular and well-known data for an Intrusion Detection System is the audit data. An *audit trail* refers to the (audit) records of all activities on a system kept in chronological order. Since there exists a record for each activity on the system, theoretically it is possible to manually analyze the audit data and detect any abnormal activity on the system. However, the vastness of the audit data provided by an audit collection system often makes the manual analysis impractical. Therefore, an automated audit data analysis tool is the only solution. Intrusion detection systems are sometimes enhanced versions of these analysis tools. The following are the most popular approaches used for intrusion detection.

- Statistical Anomaly Detection
  - Threshold Detection
  - Profile-Based Anomaly Detection
- Rule-based Anomaly Detection
- Rule-based Penetration Identification

USTAT falls in the last category of the intrusion detection tools: Rule-based Penetration Identification Tools. These tools are characterized by their expert system properties that fire rules when the audit records indicate illegal activities. Most of the current intrusion detection tools supplement their anomaly detection components with rule-based expert system components (e.g., IDES [Lunt92], NADIR [Hubb90], and W&S [Vacc89]). The different approaches and their corresponding tools are discussed in detail in [Porr92A].

Current rule-based penetration identification tools have several weaknesses that USTAT aims to improve on. One major weakness is that they use audit records to represent a penetration scenario and try to pattern match their rules to the audit records. The representation of scenarios using audit records is very non-intuitive. This process requires a person who is experienced in the particular intrusion detection system and who has in-depth knowledge of the underlying audit collection mechanism. Also, pattern matching rules to the audit records gives no flexibility to the representation of penetrations. That is, for the same scenario several different audit record sequences might exist and those minor variations might slip unnoticed. USTAT overcomes this problem by using a higher-level audit record independent representation of penetration scenarios. This feature also makes the creation and update process of the rule-base easier.

STAT (State Transition Analysis Tool) presented in [Porr92B] introduces a novel idea to represent computer penetrations and provides an expert system

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

model to detect compromises. STAT makes use of the audit trails that are provided by the audit collection mechanisms of the target operating systems.

Garvey and Lunt [Garv91] also proposed a new intrusion detection approach called Model-Based Intrusion Detection. With this approach they address the above problems and provide an audit record independent technique to represent intrusion scenarios.

USTAT (State Transition Analysis Tool for UNIX), which is introduced in this paper, is the implementation of a prototype of STAT for UNIX. USTAT uses the audit collection mechanism that exists as an add-on package to SunOS 4.1.1, called C2-BSM (Basic Security Module). In the remainder of this paper we refer to the SunOS 4.1.1 C2 Basic Security Module as the BSM.

USTAT's design gives the site security officer (SSO) the opportunity to monitor, detect and possibly preempt certain activities that would be considered illicit or that would cause a security risk for the system.

The following section gives an introduction to the State Transition Analysis approach. Section 3 presents a discussion of the components of USTAT. Section 4 shows the results of various tests performed on different aspects of USTAT. The final section gives conclusion about the research reported in this paper and provides pointers for future research efforts.

2 Introduction to the state transition analysis approach

2.1 State and state transitions

USTAT analyzes the audit data by keeping track of the state changes on the system, where state is defined as follows [Porr92A].

> *"State is the collection of all volatile, permanent and semi-permanent data stores of the system at a specific time."*

However, for a given instant in time, it is quite infeasible to determine the value of all the data stores on the system. A close observation of the penetration scenarios reveals that we actually need only a fraction of the data stores to represent those scenarios. The attributes we are interested in depend on the particular scenario.

For instance, we can define one compromised state as follows. "A user (non-root) is running an interactive shell with an effective user id equal to root." So, the user can talk to the shell while having root privileges. That clearly defines a compromised state. However, at this time, there is no audit record format that informs the analyzer in such a descriptive manner. Therefore, we try to obtain more information looking at the history of the "compromised" state. We try to see how we have gotten there and we try to answer the following questions.

1. Which actions caused the user to gain an interactive shell with root privileges?
2. How are the system attributes affected by these actions?

The observations made by answering these questions make state transition analysis a very effective tool in describing and detecting penetration scenarios. Since the actions are as significant as the states of the system, the tool is called the state *transition* analysis tool rather than state analysis tool.

2.2 Representing penetrations: state transition diagrams

A state transition diagram is the graphical representation of a penetration scenario. Figure 2.1 shows two major components of a state transition diagram: Nodes that represent the states and arcs that represent the actions.

Figure 2.1 State and Action

The idea of the state transition diagrams stems from a very basic observation of a feature that is common to all penetrations: All intruders start with limited access to a system with limited privileges (= Initial state). After performing some actions they gain some previously unheld ability (= Final state). This is illustrated in Figure 2.2.

Figure 2.2 Initial and Final States

So, we can view a penetration as a sequence of actions that lead from an initial limited access state to a final compromised state. When we construct a state transition diagram we use only key activities

- that make a state change on the system,
- that lead to the final state, and
- that best represent the penetration.

It is easy to observe the first two, but the last one is quite intuitive. There is no clear cut procedure that can define the construction of state transition diagrams, and it is possible that two different persons can come up with different state transition diagrams that represent the same penetration scenario. Which one is the best is difficult to answer.

Each state of a state transition diagram consists of one or more state assertions. In representing penetration scenarios we discovered that we don't need any specific state assertions for the initial state of a penetration scenario. Therefore, all state transition diagrams start with a signature action, which acts like a trigger for the penetration scenario. An example best illustrates the construction of state transition diagrams.

2.3 An example penetration scenario and its state transition diagram

In this section we give an example penetration scenario that is applicable to SunOS 4.1.1. In the following example, the target file, called *target*, is a setuid shell script with the #!/bin/sh mechanism and is owned by root. The file that is linked to *target* starts with a dash '-'. The attacker performs the following steps.

```
% ln target -x
% -x
```

The steps of this process can be explained as follows.

Step-1. The attacker creates a hardlink starting with a dash '-' to root's setuid shell script that contains the #!/bin/sh mechanism.

Step-2. The attacker executes '-x'.

Insight: Whenever a hardlink is created, a new directory entry is created with the target's original privileges and ownership information. The target can be accessed via any link to it. Executing a shell script containing the #!/bin/sh mechanism invokes a subshell. This subshell becomes interactive (meaning that the user invoking the shell can talk to it), because the name of the script starts with a '-'. Since in this case the attacker is executing a setuid file owned by root, he/she receives an interactive shell with root privileges.

There are two steps involved in this penetration and each is necessary for the successful completion of the attack. First we can identify the signature actions: The first one corresponds to the *hardlink* action among USTAT's action types and the second one is *execute*. So far we have the incomplete diagram of Figure 2.3.

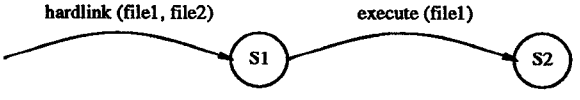

Figure 2.3 An Incomplete State Transition Diagram

Next we identify the state changes accomplished by these calls. In the final state the attacker gains some previously unheld ability: At this point, he/she has an effective user id of root. So, the final state can be identified as:

```
not euid = USER
```

To complete the diagram we should identify the first state (S1). In fact we should include every fact that is used in the first step of the penetration explained above. The new filename should start with a dash, followed by any characters:

```
name (file1) = "-*"
```

The new file is hardlinked to a root owned setuid file. So, the new file's ownership should indicate a user different than the USER who executed the last signature action, as given in the following state assertion.

```
not owner (file1) = USER
```

The file must be a setuid shell script:

```
permitted (SUID,file1)
shell_script (file1)
```

Finally, the file should allow execute access to group or others:

```
permitted (XGRP,file1) or
permitted (XOTH,file1)
```

With these additions the state transition diagram is completed and it is shown in Figure 2.4.

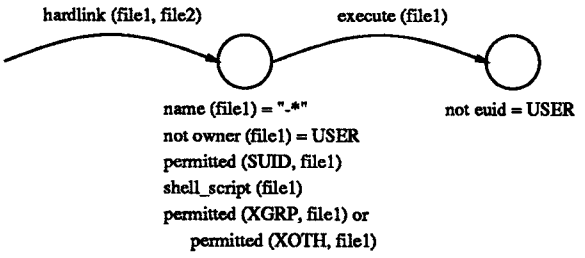

Figure 2.4 Final State Transition Diagram

For a complete description of the state assertions used in USTAT, see Section 3.2.2.

3 Ustat

In Section 2.2 we described how a penetration scenario can be represented using state transition diagrams. In this section, we explain how USTAT stores these scenarios and uses them to detect penetrations. USTAT can be characterized by three basic properties:

- USTAT is a real-time expert system intrusion detection tool,

- It employs rule-based analysis on the audit trails of multi-user computer systems, and
- It searches for known penetrations.

USTAT is designed to be a real-time system. One of its main features is to attempt to preempt an attack before any damage is done to the system. This preemption is possible only with real-time analysis. The major issue in real-time analysis, however, is whether USTAT 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 4.

USTAT's ability to detect cooperative attacks, to detect penetrations, the steps of which may span more than one user session, and its ability to foresee an impending compromise distinguish it from other rule-based penetration identification systems. For instance, the example scenario presented in the previous section can be performed by two different cooperating attackers, or by one attacker in two different login sessions. USTAT is able to detect such variations in attack scenarios. For an in-depth discussion of USTAT's features and its implementation details refer to [Ilgu92].

USTAT consists of the following components.

- The preprocessor
- The knowledge-base
  - The fact-base
    - The fact-base initializer
    - The fact-base updater
  - The rule-base
    - The state description table
    - The signature action table
- The inference engine
- The decision engine

Except for the preprocessor, these components characterize a typical expert system. USTAT provides modularity by designing each of these components separately and interfacing them together. Figure 2.5 illustrates the connectivity of the components of USTAT.

3.1 The 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.

3.1.1 Audit collection

Operating systems with NCSC [1] evaluation of C2 or higher are required to provide audit collection mechanisms. The BSM is designed to be compliant with the NCSC requirements for a system at the C2 classification. The BSM provides improved security features over standard UNIX operating systems. It has the following add-on features to SunOS 4.1.1.

- Shadow password files
- Object reuse
- Device allocation, deallocation
- Audit collection

Among these features, our primary interest is the audit collection. For more information about the BSM refer to [Bsm91].

3.1.2 Format of Ustat audit records

The USTAT audit record structure is defined by the triple:

<SUBJECT, ACTION, OBJECT>

meaning "SUBJECT performs the ACTION on the OBJECT." Each of these attributes contains further fields that are used to reveal as much information as possible about the particular attribute. The *SUBJECT* is identified by the triple:

<Real User ID, Effective User ID, Group ID>

The *ACTION* is identified by the triple:

<Action, Time, Process ID>

Finally, the *OBJECT* is identified by the eight tuple:

<Object Name, Permissions, Owner, Group Owner, Inode #, Device #, File System ID, Target>

The *Object Name* is the name of the file identified with its full path. The *Target* field is effective only if the action is *Hardlink* or *Rename*. 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 as regards to USTAT, refer to [Ilgu92].

3.1.3 Filtering process

There are 239 different events that are audited by the BSM. Out of these, only 28 events are used by the preprocessor and mapped onto 10 different USTAT actions. The inference engine operates using these 10 action types. Table 3.1 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. This value means that the call made by the user was not finished successfully. It did not make any change to the system attributes and hence it cannot cause a state transition.

[1] National Computer Security Center

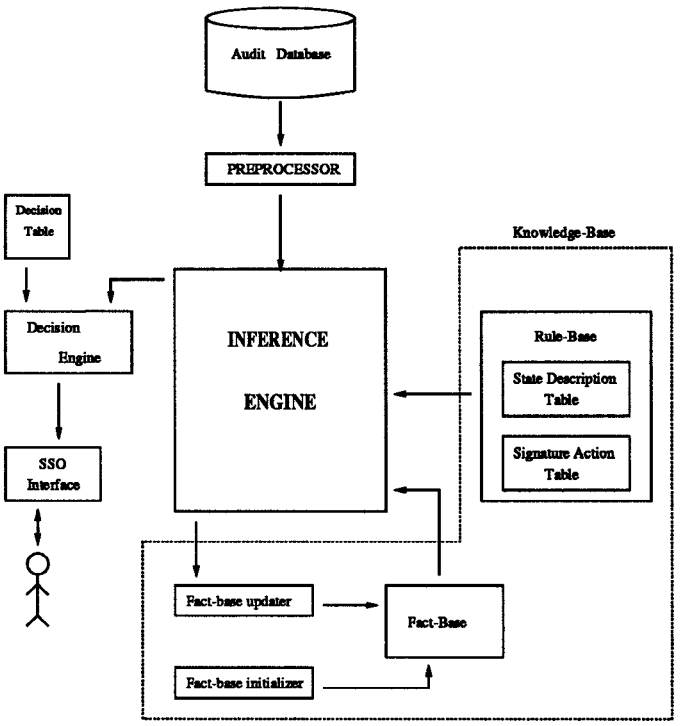

Figure 2.5 USTAT's Components

Note: 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.

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 3.1 USTAT Actions vs. BSM Event Types

One major task of the preprocessor is to provide the inference engine with a generic audit record format to aid in portability. However, some action names specific to the target system need also be included (e.g., hardlink). The preprocessor also enables the SSO to create the state transition diagrams with the abstraction of USTAT action names.

One might argue that it is easier for the SSO to create the state transition diagram by directly using the system's audit records that correspond to the command line sequence of the attack scenario. This may be true. However, with such a state transition diagram small variations to the attack scenario are very likely to slip unnoticed. USTAT overcomes this problem by mapping groups of many similar actions onto single USTAT actions.

3.2 The knowledge-base

Figure 2.5 shows the knowledge-base components within the dotted lines. The fact-base contains information about the objects of the system, and the rule-base is the rule representation of the state transition diagrams.

3.2.1 The fact-base

USTAT's fact-base consists of groups of files or directories that share certain characteristics that are vulnerable to certain types of attack scenarios.

Initializing the fact-base for USTAT is done by the fact-base initializer module of USTAT and by some additional manual processing. The fact-base updater consists of those 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 3.2.

FILESET	CHARACTERISTICS
Fileset #1	Restricted read files
Fileset #2	Restricted write setup 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 3.2 USTAT Filesets

The first six of these 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.

- The files in Fileset #1 should not be accessed via regular utilities, as 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 [Disc85] illustrates how plaintext passwords can be obtained from /dev/kmem. In recognition of this violation and the potential for similar ones, USTAT makes use of Fileset #1. These files should only be read by certain system files that are identified in Fileset #3. USTAT makes use of the state transition diagram given in Figure 3.1 to detect unauthorized references to the files in Fileset #1.

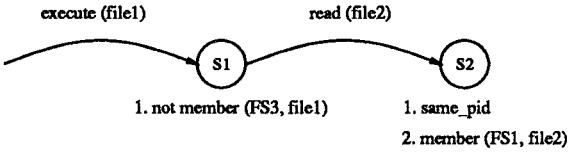

Figure 3.1 A State Transition Diagram

- Similar to Fileset #1, files in Fileset #2 should be denied write access except by the programs in Fileset #4. An example to a member of Fileset #1 is /etc/passwd. This file should not be overwritten by any program, except by the *passwd* 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. USTAT uses a state transition diagram similar to the one given in Figure 3.1 to detect unauthorized writes to the files in Fileset #2.

  Some famous attack scenarios consist of maliciously executing a setuid to root [2] program (e.g. a program that is not a member of Fileset #4) to perform unauthorized activities on the system (e.g. writing to a file that is a member of Fileset #2). Such scenarios can be detected by USTAT since the BSM provides both the effective and real user id of the attacker.

- Fileset #3 consists of the files that are authorized to read files in Fileset #1. The meaning of *authorized* does not include *being able to*. For instance, although cat(1) can be used to display the contents of any file as far as permission bits allow, it should not be defined as authorized to read a file 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 legitimate write access to the /etc/passwd file, can be used by any user, and therefore should be included in Fileset #4.

- Fileset #5 consists of publicly accessible, executable system files, which are common subjects to Trojan Horse attacks. These files should not be deleted because of 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 subject to Trojan Horse attacks. If somebody creates a fake *ls* 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* program is located. Therefore all these directories should be denied write access unless the access is done by root (or the system administrator).

- 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, we 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, since the object in the next step was not identical to the object that it was looking for. To overcome

[2] These programs (or scripts) temporarily change the user's effective user id to root. The real user id remains unchanged and fortunately both id's are recorded by the BSM.

this, USTAT's fact-base keeps information about all hardlinks on the target system.

3.2.2 The rule-base

State transition diagrams provide a graphical representation of penetration scenarios and their effects on the system states. The information contained in these diagrams are stored 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 the information contained in these files to match the actions of incoming USTAT audit records to the actions of state transition diagrams.

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

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 file *nwsd.set*.

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*)

3.3 The inference engine

In this section we describe the operation of the inference engine. The inference engine forms the heart of the control mechanism. The inference engine does not "know" what rules and facts should be or could be in the knowledge-base. For any given inference step, the inference engine uses all the relevant rules and facts that are available to it at that time. USTAT's inference engine uses an event driven, forward chaining inference scheme [Mart88].

The inference engine uses a structure called the *inference engine table* to handle and detect all penetration scenario instances simultaneously. 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 achieved. Each row also contains information about the history of the instantiation, such as users involved, files involved, related audit records, etc. Table 3.3 illustrates the initial configuration of the inference engine table. Assuming n is the number of different state transition diagrams entered into the state description table, then there is one row for each state transition diagram. None of the rows are marked, which means the initial signature action of each diagram 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 diagram, it duplicates the row and "marks" the corresponding cell on the duplicated row. The reason for duplicating the row is that the original row can still represent part of another instantiation.

To illustrate the manipulation of the table we give an example that uses a hypothetical state transition diagram, called STD_h. This diagram contains three signature actions. Suppose that the first signature action is creating a file that possesses certain characteristics, which are given in the first state of the diagram (State Assertions-1). Also, suppose that the second signature action is the execution of the file created in the first step and is followed by the third signature action, which indicates reading another file on the system. The state that follows the last signature action depicts the final compromised state. This is illustrated in Figure 3.2. From this diagram one can observe that there may be many executions of *file1* by possibly different users once it has been created.

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

Table 3.3 Initial inference engine table

The hypothetical state transition diagram (STD_h) corresponds to Row h of the initial inference engine table.

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

Result: Row h is duplicated and S_1 is marked on the new row. (See Table 3.4).

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

Table 3.4 Inference engine table after action 1

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. These two signature actions correspond to the two rows (h) and ($n + 1$) of Table 3.4, respectively.

Action 2: User B executes file1 and this satisfies S_2 of STD_h.

Result: Row $n + 1$ is duplicated and S_2 is marked on the new row. But, this result should not affect row $n + 1$ since the same file can be executed by another user. The result of the second action is illustrated in Table 3.5.

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

Table 3.5 Inference engine table after action 2

Action 3: User B reads file2 and this satisfies S_3 of STD_h.

Result: Since S_3 is the final state of STD_h a compromise has been achieved. No new row is added to the table. So, as a result the table will still look like Table 3.5. 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 $n + 1$.

3.4 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 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 action 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 all of the above except the last one.

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

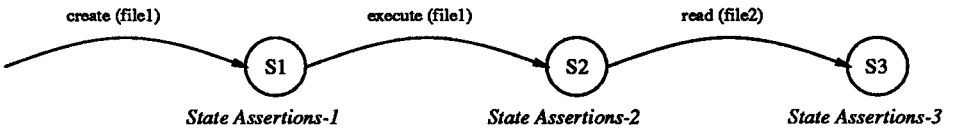

Figure 3.2 STD_h: A Hypothetical State Transition Diagram

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

With this data, the SSO has enough information to either take preemptive action, or to take precautions to prevent further attacks.

3.5 Out of scope

No intrusion detection tool is meant to be a catch-all tool for intrusions. They all have weaknesses and strengths. The following is a list of threats that are out of scope for the analysis of USTAT.

- Manipulation of components outside the system's execution domain, e.g., wiretapping.
- Most denial of service attacks, e.g., excessive CPU utilization.
- Variations from patterns of use, e.g., user A logs on as user B and accesses the files that user B doesn't usually access.
- Failed login attempts, failed file access attempts.

The above list consists of those attacks that are either not recorded by the audit collection mechanism, or that cannot be represented by using a state transition diagram. With this in mind, USTAT will best perform as a component in a larger intrusion detection system by cooperating with an anomaly detector, since the weakness of one is the strength of the other. For instance, an attacker trying to break into a user's account by using the login program repeatedly will generate many audit records with a fail indicator. This high number of unsuccessful attempts is expected to be caught by a statistical anomaly detector. Even if the attacker finally succeeds, the break-in will not be detected by USTAT.

4 Testing Ustat

In this part we give the test results of running USTAT. First we give the results of the functional tests and then the results of the performance tests. All tests were run on a *SPARCstation 1.*

4.1 Functional testing

We performed functional tests on three different features of USTAT.

1. The rules of 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 the 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. Do the rules work if parts of a scenario are performed by different attackers? The example scenario given in section 2.4.3 allows two attackers to cooperate. USTAT was able to detect this attack, since for USTAT it makes no difference whether this attack is performed by only one user or two users.

4.2 Performance testing

The purpose of the performance testing was to have an idea about 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. We tested USTAT with a combination of *audit intensive* [3] and cpu intensive processes. *ps* was run periodically to obtain the cpu and disk utilization of the processes. It is obvious that *ps* 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

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

data was 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.

Although there are many combinations of different types of processes possible, we limited our performance tests to a small number of typical processes. The following were the processes that were involved in the test.

- USTAT: Both cpu and I/O (primarily input) intensive. USTAT itself is not audited since otherwise it would result in a circular action.
- Crack: A cpu intensive program that runs encryption routines to guess passwords.
- find: An audit intensive system program, when run on a large filesystem (e.g., find / -print) creates a huge amount of audit data.
- Audit daemon (auditd): 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.

For each test case, we illustrate one table and one or more graphs associated with the test case. The table lists the processes involved in the test run on one column. Next column indicates the amount of cpu time (min:sec) used by each process, and the third column indicates the percentage of cpu time used by each process. The graphs display the cpu usage of selected processes over time.

4.2.1 Test case 1: Ustat only

In this first test we tested USTAT's performance 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. The graph in Figure 4.1 displays USTAT's cpu usage over time. USTAT used about 42 % of cpu during processing and 38 % when idle. The cpu utilization during this idle period seems high. More than 30 minutes (more than 50 %) of cpu time was idle even during USTAT's processing (See Table 4.1). This indicates that for half its runtime USTAT was waiting for the I/O to be completed.

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

Table 4.1 Cpu usages [4] of auditd and USTAT

[4] Idle includes other processes

4.2.2 Test case 2: Ustat and crack

In this test run, the cpu was rarely idle. Because both are user processes, USTAT and crack 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 4.2). The test results also showed that the cpu utilization of USTAT dropped by 5 %. Figures 4.2 and 4.3 display the cpu usages of USTAT and crack over time.

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 4.2 Cpu usages of Auditd, USTAT and Crack

4.2.3 Test case 3: Ustat and find

Being an audit intensive process, find 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 hung after a while (See Graph of Figure 4.6). The only solution to this hang was to terminate the execution of find, 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 cpu time was idle, since USTAT didn't have much opportunity to process audit data (see Table 4.3). Figures 4.4 through 4.6 display the cpu usages of auditd, find and USTAT over time.

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

Table 4.3 Cpu usages of Auditd, USTAT and find

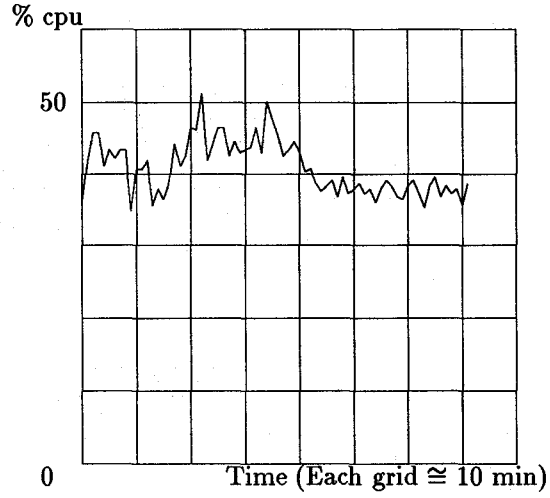

Figure 4.1 USTAT's cpu usage in running USTAT

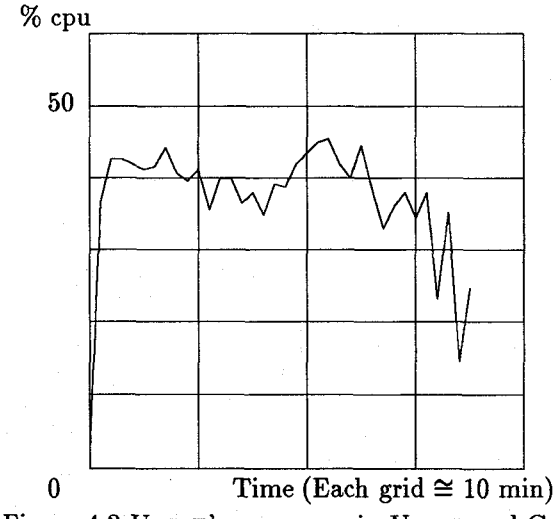

Figure 4.2 USTAT's cpu usage in USTAT and Crack

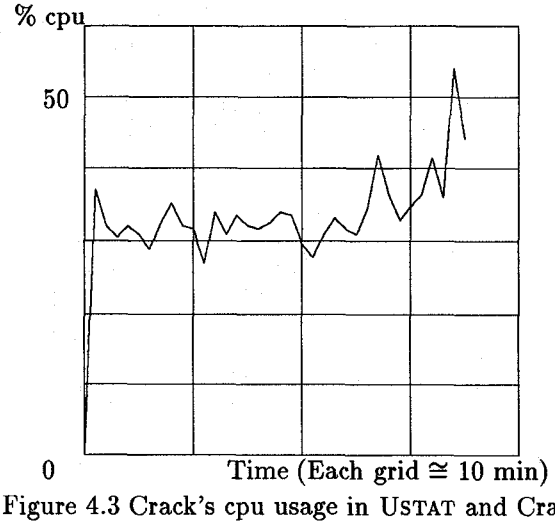

Figure 4.3 Crack's cpu usage in USTAT and Crack

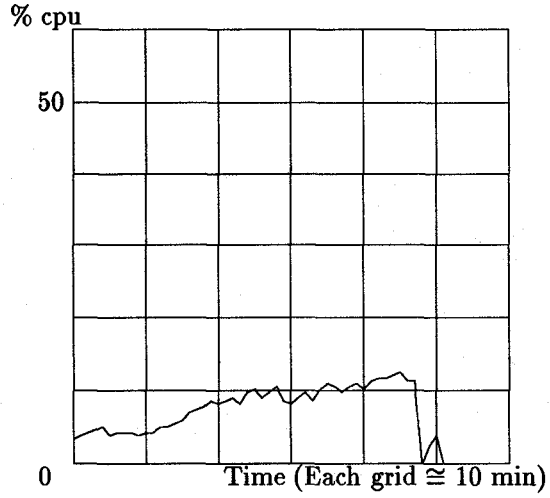

Figure 4.4 Auditd's cpu usage in USTAT and find

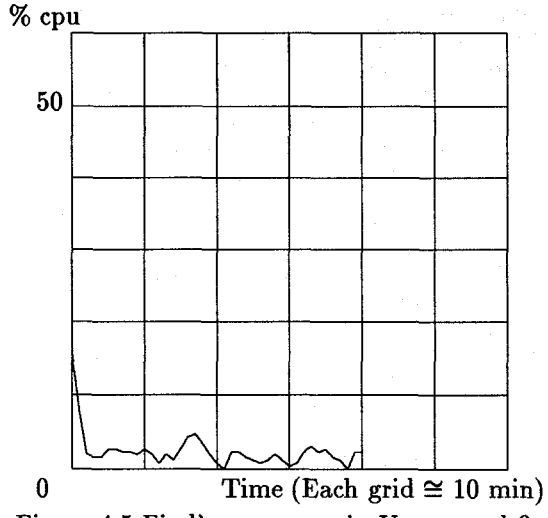

Figure 4.5 Find's cpu usage in USTAT and find

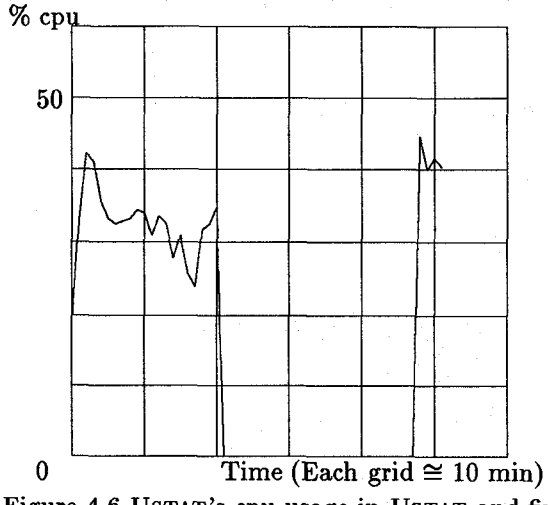

Figure 4.6 USTAT's cpu usage in USTAT and find

The following lists some factors that might affect the speed of USTAT.

1. Size of hardlink information.
2. Number of rules in the rule-base.
3. Cpu load on the machine where USTAT is running: Other users, processes, the audit daemon, etc.
4. Load of the disk on which the audit trails are recorded.
5. The network traffic load if the audit data is sent over the network.

5 Conclusion and future work

In this final section, we first give some comments about the overall results of the test runs. Next, we give some topics to work on in the future to improve the efficiency of USTAT and broaden its scope.

5.1 Remarks about the tests

In running the tests presented in the previous section, we tried to observe both the effectiveness and the efficiency of USTAT. These tests were meant to be a feasibility check rather than to be exhaustive. In fact, 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 affect 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 in Section 4.2). Among the possibilities to increase the performance of USTAT are:

- Running USTAT on a dedicated machine (IDES [Lunt92] uses this approach),
- A periodic audit filesystem switch for the audit daemon, so that USTAT will not starve while waiting for the disk I/O (see Test Case 3, in Section 4.2).
- Running USTAT and the audit daemon on a system with better performance, (faster disk drives, etc.).
- If USTAT and the audit daemon need to be run on the same computer, USTAT can be run as a higher priority process. This would speed up USTAT's processing, but there is still a problem associated with this. As in the test case 3 in Section 4.2, the audit daemon is already falling behind real-time when there are many events to be recorded. Increasing USTAT's priority would likely cause the audit daemon to fall even further behind.

As indicated by test case 1 in Section 4.2, the cpu utilization of USTAT when it is idle (waiting for more audit records) is not much different from when it is processing. This is because USTAT is continuously reopening the audit data file and trying to read new data. Instead of making this continuous, we could add a delay between each reopen, so that a high slow-down in the speed of other processes is not experienced.

5.2 Future work

- Performance in a larger intrusion detection system:

  It would be very informative to see the performance of the overall intrusion detection system if we test USTAT as a component of a larger intrusion detection system that consists of USTAT and a statistical anomaly detector (such as IDES).

- Interactive interface:

  Currently, after the USTAT program is started, there is no other input to USTAT except the audit data. Another module can be added to USTAT to provide more interaction between the SSO and USTAT. One desirable feature of this interface would be the capability of adding and removing rules to the rule-base while USTAT is running.

  Also, all state assertions of USTAT are currently built-in. The rule-base can only use those predetermined state assertions. To add modularity, the SSO might be allowed to define new assertions by writing small programs once the interface is defined. This feature is expected to be added to future versions of USTAT.

- Multiple hosts:

  One of our next efforts will be to run USTAT on the audit data collected by several hosts. In this case, the following issues need to be considered. When the BSM is installed on multiple hosts, each host generates its own audit trail. USTAT's preprocessor is designed to operate on a single, chronological audit trail. Therefore, the audit trails need to be merged into one chronological audit trail and then processed by USTAT's preprocessor. The *auditreduce* command installed with the BSM may be used to perform this merge. (See [Bsm91] or the auditreduce(8) man pages for more information about auditreduce). Also, all hosts to be audited must mount their filesystems at the same mountpoints to assure consistency in filenames. There will be some files that will have identical names, but in fact correspond to different physical files, such as the files in /usr/bin. In our analysis, we used the full pathnames to identify files. However, when performing the analysis for multiple hosts, device and inode numbers of files will also need to be used.

- USTAT's security:

  The security of USTAT and the audit collection mechanism is not addressed in this prototype implementation. The following issues come to mind regarding the security of USTAT. Is there any way for an attacker to become a clandestine user? Can the attacker disable the audit mechanism? For example, in the test cases that involved audit intensive processes (such as find), we encountered a considerable amount of delay in the audit daemon's recording time. In this case a malicious user could run an audit intensive process to keep the audit daemon behind real-time, penetrate the system and disable the audit daemon before being detected. It might also be desirable to have dedicated audit disks so that the attacker does not fill the disk space or does not keep the audit disk busy.

Finally, by implementing USTAT the conceptual soundness and functional capabilities of the State Transition Analysis model have been validated. The author hopes that this effort will be followed by more research to serve the area of intrusion detection.

Acknowledgments

First, I wish to thank to my advisor, Prof. Richard A. Kemmerer, for his keen interest and support in this project and for proofreading this paper. I also thank Phillip A. Porras, who developed the State Transition Analysis approach and provided many helpful comments for the implementation of USTAT.

References

[Bsm91] Sun Microsystems, Inc., SunOS Release 4.1.1. C2-BSM Patch, Revision A, 2550 Garcia Ave, Mountain View, CA 94043, Dec. 15, 1991.

[Disc85] A. V. Discolo, "4.2 BSD UNIX Security," Computer Science Dept., University of California, Santa Barbara, Apr. 1985.

[Garv91] T. D. Garvey and T. F. Lunt, "Model-based Intrusion Detection," *Proceedings of the 14th National Computer Security Conference*, Baltimore, MD, Oct. 1991.

[Hubb90] B. Hubbard, T. Haley, N. McAuliffe, L. Schaefer, N. Kelem, D. Wolcott, R. Feiertag and M. Schaeffer, "Computer System Intrusion Detection," Trusted Information Systems, Inc., RADC-TR-90-413, Final Technical Report, Dec. 1990.

[Ilgu92] K. Ilgun, "USTAT: A Real-time Intrusion Detection System for UNIX," Master's Thesis, Computer Science Dept., University of California, Santa Barbara, Nov. 1992.

[Lunt92] T. F. Lunt, R. Jagannathan, R. Lee, S. Listgarten, D. L. Edwards, P. G. Neumann, H. S. Javitz and A. Valdes "A Real-time Intrusion Detection Expert System (IDES)", SRI Technical Report, Feb. 28, 1992.

[Mart88] J. Martin and S. Oxman. *Building Expert Systems: A Tutorial*, Englewood Cliffs, N.J., Prentice-Hall, 1988.

[Porr92A] P. A. Porras, "STAT: A State Transition Analysis Tool for Intrusion Detection," Master's Thesis, Computer Science Dept., University of California, Santa Barbara, July 1992.

[Porr92B] 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, Dec. 1992.

[Vacc89] 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.