Part 3:

Analysis of IDS Detection Methods
In order to do a study on the effectiveness of the detection methods being employed by an Intrusion Detection System in practical implementation, four distinct intrusion detection methods have been chosen for further analysis. These methods are not an exhaustive list of all known detection methods in use [13]. They are selected to distinguish the different ways in how an intruder can be detected. There are literally dozens of methods being used by IDS each having different names; however, many of them are just variations of the four methods chosen for this research report. The following is a summary of the detection methods to be studied:

1) Behavior-based intrusion detection by observing the sequences of system calls – This method monitors the privileged processes of an operating system and checks them for any anomalous behavior.

2) Behavior-based intrusion detection through file integrity checking – This method scrutinizes file properties for changes from a previously known state.

3) Knowledge-based intrusion detection using state transition analysis – This method looks at the state transitions caused as a result of user operations as see if they would lead to an intrusion

4) Knowledge-based intrusion detection using pattern-oriented modeling – This method tracks the data and privilege flows for any indication of users exploiting operational security problems.

The product of this research report would be an analysis on how well the detection method in question deals with the following categories described earlier in this paper:

• Effectiveness of the detection method against intrusion types – The detection method will be assessed on how well it handles attacks from various types of external predators, internal predators, and malicious processes.
• *Capacity of the detection method in overcoming error types* – The detection method will be assessed on how well it handles false positive and false negative classification errors.

• *Performance and impact of the detection method in practical implementation* – The detection method will be assessed on how it might be expected to perform in a real production environment.

All of the analysis and information given in this section of the research report is based on an in-depth literature review of security reports and comparative work carried out by other researchers cited in the bibliography, and also on deductive analysis of how the aforementioned three categories are dealt with by the detection method in question.
3.1 Behavior-based intrusion detection by Observing the Sequences of System Calls

3.1.1 Description of Methodology

Detecting an intrusion based on behavior or deviations from a known behavior does not necessarily refer to the behavior of the user of the system at that time. It could also refer to the behavior of a particular program used by the operating system. One such method of intrusion detection was suggested by Steven A. Hofmeyr, Stephanie Forrest, and Anil Somayaji in their article titled “Intrusion Detection Using Sequences of System Calls” [19]. Their method of intrusion detection is based on the observation that many intrusions have successfully resulted from the exploitation of the statuses of privileged processes.

Many operating systems, such as UNIX, have programs that run in the background which performs services that require access to system objects. In order to protect the integrity of the operating system, these running programs are not accessible directly by users or even the application programs that need to use the functionality of these programs. These programs are called privileged processes. One of the reasons privileged processes are often exploited targeted by potential intruders is because privileged processes need superuser status from the operating system to access system resources. In UNIX, privileged processes run in the background as servers, such as sendmail, which have superuser status. Superuser status may also be obtained by users through the setuid mechanism by invoking the privileged processes. If the privileged process had a problem due to an error in coding or has developed a vulnerability due to a problem within another privileged process, they can be potentially exploited by ordinary users to gain the superuser privileges of the process, thereby allowing the intruder access of the entire system.
Fortunately, privileged processes only perform a specific limited function and in UNIX, privileged processes access system resources through the use of system calls. This has become the basis of the research done by Hofmeyr et. al. [19]. One of the observations made by them is that the behavior of privileged processes is relatively stable over time, with particular functions of the processes always following a standard sequence. Also, over time the actions and functions of privileged processes usually do not go through much variation. As a result, the range of behaviors exhibited by privileged processes are limited and less varied when compared to the range of behaviors exhibited by users. This limitation of behavior observed in privileged processes is made possible in UNIX by using the program specification method which constrains the process in a manner so that it can only perform those operations the process is designed to do by specifying the system calls used by it. The properties of an IDS methodology are determined by the discriminator or observable used, and in most cases, irregularities can be detected when the observable is a dynamic characteristic of the program or process. In this detection methodology, the authors have chosen short sequences of system calls to be the observable.

Having determined the observable, this method of intrusion detection performs anomaly intrusion detection as previously mentioned: it depends on the deviation of a known acceptable behavior to detect and determine what could be intrusive behavior. The method has two stages of operation: -

1. Stage One: Initially build up profiles or databases of acceptable and normal behavior for each privileged process available in the system.

2. Stage Two: Use the newly created databases to monitor current system behavior for significant deviations from normal.

It has be hitherto mentioned that 'normal' in this method is defined in terms of short sequences of system calls used by the privileged processes. As can be seen in
Stage One, each program or privileged process will have its own database. This is necessary to correctly determine not only the acceptable behavior of the program, but also the acceptable behavior of the program within the confines of the host operating system. For example, the sequence of system calls used by lpr in SunOS may be quite different from the sequence of system calls used by the same program in AIX. Having a generic database for the same privileged process to be used by the IDS in two distinctly different versions of the operating system would cause many instances of false positives if the generic database only takes into account the acceptable behavior matching in both operating systems, or false negatives if the generic database takes into account all acceptable behaviors shown in both operating systems. In addition, if there is only a single database of acceptable short sequences of system calls without distinguishing the program making the calls, this would cause many instances of false negatives. The reason for this is because what is an acceptable use of short sequence calls in one program might possibly be an unacceptable use in another program. Granted there would be a lot of advantages in terms of installation, deployment, and resource usage if the databases could be made generic and compatible across many different platforms – reflecting the direction of computer systems development where there is more bias towards increased uniformity and integration to enhance portability and maintainability. Unfortunately, all the advantages of uniformity will become disadvantages and potential weaknesses when errors can be exploited by intruders. Suppose that an attacker discovers a way to compromise the security of a single computer, every computer with the same configuration will become similarly vulnerable. Hence, the exercise of profiling normal behavior used in this IDS method would greatly prevent generic system attacks.
To further understand how this method of intrusion detection works, the stages mentioned in the previous paragraph will be broken down to its component algorithms. In the first stage, the IDS will construct its normal database for each program.

1. Whenever a program or privileged process is executed the IDS will scan the program and follow the traces of system calls generated by the process. The trace does not include the parameters used by the system call. This is done to reduce the complexity of the detection algorithm and also to reduce the use of system resources like disk space. So if a system call generates the following trace:-

```
open myfile.txt
```

The trace will be concatenated and only register the system call itself

```
open
```

2. The IDS will follow the trace until the privileged process has completed its present operation. Let's say that the process being monitored was `sendmail`, the trace that is generated might look like the following:-

```
open, read, mmap, mmap, open, read, mmap
```

3. Once the IDS has this trace, it will slide a window of size $k$ through the trace to get all unique short sequences of system calls seen in the trace. The value of $k$ can be adjusted by the IDS administrator to determine sensitivity of the IDS. So, using the previous example, if the administrator sets the value of $k$ to 3, the previous trace will generate the following unique short sequences:-

```
open, read, mmap
read, mmap, mmap
mmap, mmap, open
mmap, open, read
```
4. In order to save storage space and to simplify anomaly detection these unique sequences will be stored in the program database as trees that are rooted at particular system calls as seen below:-

```
open  read  mmap
     /     /     \
read  mmap mmap open
     /     /     /     \
mmap  mmap open read
```

5. The trace is run repeatedly for that program or privileged process until all possible *acceptable* short sequences that might occur in a normal non-intrusive situation is generated. As the window size \((k)\) increases, the database size will grow as well because more unique sequences are possible in that trace. Similarly the database size is reduced if the window size \((k)\) decreases. The size of the database is recorded in terms of the number of unique sequences \(N\).

Stage One is run for all privileged processes during a determined control period. In this control period, anomaly detection would not yet be active because the generation of the normal databases is yet to be completed.

Once the normal database has been completely generated, the IDS will go into the second stage. Stage Two's algorithm is as follows:-

1. Using the same method for tracing used in Stage One, the tracing will now be used to check new traces of privileged process behavior. The window size \((k)\) will be the same as it was specified in the first stage. This is because the sequences are stored in that length, therefore comparison could only be made if the lengths are equal.
2. When the privilege process executes, the IDS will scan a trace for the entire operation and also trace all other privileged processes forked from the operation. Each of these new forked processes will have their individual traces.

3. For each new trace, all overlapping short sequences of size $k$ will be collected and compared each time to the privileged processes normal database to see if they are represented there. An illustration of the sequence comparison process is shown in Figure 2 below.

Figure 2: The graph above shows how the IDS is used to detect anomalous sequences as a result of a modified sendmail program residing in the system.
4. If a new sequence does not match any of the sequences shown in the database, then it is considered a mismatch.

5. If the number of mismatches in a new trace increases, then this would be indicative of anomalous behavior. The higher the number of mismatches, the stronger the signal of possible anomalous behavior would be.

*Figure 3: Basic structure of an IDS tool that observes the Sequences of System Calls.*

*Figure 3* above shows the basic structure of an IDS based on this detection method. In this method of intrusion detection, it is the responsibility of the IDS system administrator to decide what is the threshold of anomaly signal strength before an intrusion alarm is set off. The administrator must practice caution in determining this threshold. A low threshold could possibly set off alarms repeatedly when they might just be legal process behavior that was not recorded in the database generation stage. A high threshold, on the other hand, might overlook intrusion methods that consistently
generate low or moderate anomaly signals. In general, short sequences of system calls are good discriminators between normal and abnormal operating characteristics of several UNIX programs.
3.1.2 Effectiveness of Detection Method against Intrusion Types

One of the greatest benefits with scrutinizing how privileged processes are being used is that any attempt to replace these privileged processes with modified or hacked versions or exploit a weakness in coding will be detected. Monitoring the sequences of system calls used by these processes guarantees this, assuming the window size \((k)\) used to monitor the traces is large enough to detect an anomaly in the sequence. In this section, the categories of intruders and intrusion types brought forward by Denning [11], Anderson [1], and Brunstein et. al. [5] will be used to see how this detection method holds up to each and every one of these categories.

*External predators*

1) *Attempted break-in* – There are at least two methods favored by would-be intruders from outside an organization in their attempt to gain access to a secure network. The most amateur attempt would be by password guessing. As mentioned earlier in this research report in the Section 2.1 that outlines intruder types and intruder personalities, intruders who use this method first gain a user list and password file and use brute force techniques to guess a weak password [24]. Since a detection method based on monitoring sequences of system calls is only able to detect anomalous behavior of the privileged process, this IDS method is not able to detect intruders using the password guessing technique. Suppose an intruder manages to gain access through a guessed password, he would now be categorized as a *masquerader*. A second method of attempted break-in from an external predator would be to modify privileged processes or use modified versions of privileged processes from a guest or anonymous user account to gain *superuser* privileges. The detection method described in this section would be very effective in detecting these types of break-ins. This is because
modified versions of the privileged processes would display a totally
different sequence of system calls used than the ones stored in the processes’
acceptable normal databases. Hofmeyr et. al [19] has proven this by testing
their detection method against three known vulnerabilities in syslog,
sendmail, and lpr that has been published in CERT advisories. In their tests
all attempts to exploit these vulnerabilities were clearly detected by their
detection method and showed high anomaly signals consistently with each
intrusion attempt.

Internal predators

2) Masquerading or successful break-in (Masqueraders) – This intrusion
detection method would not be very effective in detecting intruders that are
using hijacked passwords or user identities. In fact, Hofmeyr et. al. [19]
have admitted to this shortcoming in the research and development of this
detection method. The reason for this is that it is highly unlikely that a user
would continue to use weaknesses in privileged processes to gain access to
private file areas. In addition, although this method is an anomaly or
behavior-based detection method, it does not take into consideration how a
legitimate user is supposed to behave. The only exception would be if the
intruder only manages to gain access to the account belonging to a lower-
privileged user and needs to obtain access to a higher-privileged account like
root. This act would be similar to those used by clandestine users. Any
activity to exploit privileged processes at this point will be detected,
assuming the masquerader uses these exploits.

3) Penetration by legitimate user (Clandestine users) – The previous paragraph
mentions how a masquerader can potentially become a clandestine user as
well. Regardless of whether or not the user was a masquerader to begin
with, clandestine users generally attempt to exploit the weaknesses of a
privileged process to gain access to a forbidden file area or try to use
processes that are not accessible by them. The latter case would cause
protection faults within the system especially if some forked processes are
not accessible by the user, which can be a potential anomaly detection signal
with this detection method. The former case would give a clearer
confirmation to this IDS method because it once again depends on making a
privileged process behave outside of the norm. Part of the tests Hofmeyr et.
al. [19] used to assess the effectiveness of their detection method were done
using valid user accounts to execute the exploits. The success of the
detection method in those tests reflect how effective this method would be in
detecting real clandestine users in a production environment.

4) Inference by legitimate user (Clandestine users) – It is unclear whether this
method of detecting intrusions by monitoring short sequences of system
calls would be effective in pin-pointing a clandestine user that tries to infer a
larger section of a database from a smaller one. ‘Database’ in this context
does not refer to the acceptable normal databases of the privileged processes
used by the IDS, rather they refer to a user or group database accessible by
the user. The act of inference is not indicative of any attempt to exploit a
weakness in a process. Although privileged processes are sometimes used
by user database programs, it is not a necessary tool for inference. Most of
the time, inference is done using complex database queries or stored
procedures; both of which are legitimate user actions. If this is the method
used to infer a secure area of a database, then this IDS method would not be
able to detect this kind of intrusive activity.
5) *Leakage by legitimate user (Misfeasors)* – Because of the fact that this IDS method does not attempt to profile user behavior in any way, it would completely fail to detect any misfeasor activity. As mentioned earlier, misfeasors are people who have legitimate access to a secure data area or programs, but either illegally leak them outside of the system or misuse program (or processes) function from its intended purpose. Misfeasors do not have to turn to exploiting process vulnerabilities because they already have free access and privilege to their target. This would make them completely invisible to this IDS method, unless, of course, they do attempt to exploit any privileged processes to gain more user privileges; which would once again make them *clandestine users*.

**Malicious process**

6) *Trojan horse (Malware)* – This IDS method cannot detect a Trojan horse the moment it is planted in to the system, either by copying the file into the disk area or as an attachment in an e-mail. This is because Trojan horses by design almost always needs to be executed by the user in some way or another. The ability to detect intrusion attempts by Trojan horses is largely dependent on the method in which the Trojan horse releases its destructive payload. If the Trojan horse is script based\(^1\) and the scripts used is within access of the user that activates the Trojan horse, then this detection methodology will not be able to detect the intrusive or destructive activity. This is because the destructive payload used by the Trojan horse will show acceptable normal sequences of system calls – even though the results are undesirable. However script based Trojan horses are easily detectable by all

---

\( ^1\) A script-based Trojan means that it contains a script or collected sequence of legitimate but destructive user commands like 'delete ' or other equally subversive script forms. Execution of a script-based Trojan can also be passive, such as with embedded scripts hosted on a web page visited by the user.
virus scanning software or if the user practices simple cautionary measures, like checking to see what a script actually does before running it. The preferred types of Trojan horses seen recently actually replace privileged processes with modified versions or are modified privileged processes themselves. These types of Trojan horses can be easily detected by this IDS method since the processes will display sequences of system calls that are outside of acceptable normal behavior.

7) Virus (Malware) – Similar to the payloads of Trojan horses, viruses are also equally detectable by this IDS method. Viruses are infected executables or programs within a system that have been modified in some way by another executable or program. This situation might also occur on privileged processes especially in network operating systems. As mentioned before, replacing or modifying the programs or privileged processes would cause it to produce sequences of system calls that are very different from the acceptable normal sequences. For instances, some privileged processes never use the 'write' system call because their normal operations are mostly read-only. Many viruses add the 'write' functionality in modified privileged processes so that it could in turn infect any other executables or programs it forks or works on. When the system call traces are scanned and the 'write' system call is discovered in a read-only process, this IDS method would detect the anomaly and hence intrusion alerts may be sent to the IDS administrator. The administrator will also be able to detect that the processes have been modified upon perusing the trace logs, and in locate which other processes are infected by the virus.

8) Denial-of-service (Malware) – Given the descriptions and methods on how a DoS attack might occur stated earlier in this research report, it is difficult to
see how this IDS method is able to detect such an attack. DoS attacks originating from outside the organization operate by flooding the network with non-returnable ‘ping’ packets that contain spoofed IP addresses. DoS attacks originating from within the organization on the other hand is indicated by resource hogging activities. Unfortunately, both these attack methods usually only use acceptable privileged process operations that do not generate any anomalous sequences of system calls. This situation, again, would be seen by the IDS as acceptable normal behavior and will be overlooked. The exception would be if the DoS attack occurs as a result of a vulnerability in a privileged process or if a privileged process fails due to a buffer overrun or insufficient resource error as a result of the resource monopolization activity of the denial-of service attack. In these last two cases this detection methodology will be able to detect the attack due to abnormal error sequences that occur, however it is not know whether an IDS based on this method can intervene in time to prevent any system damage.

This analysis has shown that the main strength of this method of detecting intrusions by monitoring short sequences of system calls lie mainly in any form of attack that attempts to exploit the weaknesses of a privileged process or modify a privileged process to make it perform an illegal operation. This means that it is most effective against automated forms of attacks and intrusions as is normally done by Trojan horses and viruses. Similarly, any attempt to exploit process weaknesses done by legitimate users of the system is also easily detected. The main benefit of this method is that it complements other detection programs like anti-virus software. While anti-virus software detects known signatures of viruses, it usually falters when faced with new forms of viruses or dynamically changing ones like polymorphic viruses. Since this IDS in effect maps the acceptable parameters of a privileged process, it can
potentially detect all new types or variants of viruses whenever the behavior of the process is anomalous. The main drawback of this system is that it relies exclusively on sequencing information, unlike the specific approach used in other types of intrusion detection methods, which monitors individual operations. This means that any form of attack uses the processes in its accepted, though not necessarily intended, manner will be overlooked. In short, all misfeasor activity, and many forms of masquerader activity will be seen by the IDS as acceptable; a situation which is unacceptable to most administrators.
3.1.3 Capacity of Detection Method in Overcoming Error Types

Because the method brought forward by Hofmeyr et. al. [19] depends on the generation of a normal database from its own system, it is prone to both the false positive and false negative error types that was discussed in more detail in earlier in this research report.

The creation of the normal database takes place in the first stage of the methodology. The success of this method depends entirely on the integrity and completeness of the privileged processes' database. An incomplete database that does not show many of the possible normal sequences of system calls used by a particular process would cause many instances of false positives. For instance, consider a scenario where the sendmail process is undergoing the database generation stage of the IDS. During the database generation period, users utilizing the sendmail program read and send mail normally – allowing the read and send behaviors to be properly recorded and stored in the sendmail process database. Unfortunately, during this generation period, no one performs any forwarding or attachment opening. Once the period is over and the IDS is fully activated, every time a user forwards a mail or opens a mail attachment could potentially generate an intrusion alarm – thereby increasing the possibility of false positive errors. Therefore the IDS administrator must make sure that, at the least, all commonly used functions of a particular program are run through at least a single iteration during the database generation period to prevent false positives from occurring due to an incomplete database.

On the other side of the error spectrum, including abnormal system call sequences in a process database will result in false negatives. It is extremely important that the IDS administrator ensures that no intrusions are attempted during the controlled period when the normal database is generated, otherwise it would contaminate the normal database with unacceptable sequences of system calls. For example, if an
attacker uses a known vulnerability in a privileged process to gain superuser status by using a published sendmail exploit during this period, the IDS would actually record the system calls used by this exploit as acceptable normal behavior in the sendmail sequence database. Once the control period is over and the IDS is fully operational, the same exploit can be used repeatedly by intruders and will continually be overlooked by the IDS — this is because the IDS sees the exploit as acceptable normal behavior. As mentioned in the discussion of false positives and false negatives earlier in this research report, the IDS administrator must decide whether to use a test environment or a carefully controlled production environment to make sure that no database corruption occurs.

Unique to this particular detection method is a situation where including rare normal sequences might reduce detection rates, thereby increasing the possibility of false negatives. This can occur when the sequence of system call overlaps more with abnormal sequences as the size of the normal database increases. Hofmeyr et. al. [19] has brought up the aforementioned concern from the following realization in their testing of this intrusion detection method. Their detection approach is based on the assumption that the acceptable normal behavior of a privileged process is only a small subset of all legal execution paths through a program. ‘Legal’ here must be clearly distinguished from ‘acceptable’. A legal execution is something that a process can do, whereas acceptable execution is something that a process should do. A legal operation might be possible due to a coding error (or not) during development, which would allow a program to do something it shouldn’t be able to in the first place and hence be exploited by an intruder. Therefore the database must not contain every single possible path of legal behavior, but should only reflect behavior shown in normal use. Fortunately, in reality it is likely impossible to collect all normal variations in behavior and normal behavior is likely different in different environments.
Overall, the capacity of this detection method to overcome both the false positive and false negative error types largely depends on whether the process sequence database creation takes place and also on how long the creation process takes. If the IDS administrator decides to create the databases in a test environment, he must make sure that the test system matches the production systems that he intends to implement the IDS on. This means that all aspect of software installed and hardware used must be correctly reflected. The next step would be to provide a means of simulating normal use within the test environment. Quite often this is the larger obstacle, because traffic generation in test environments rarely reflect actual use. Alternatively, the administrator can choose to deploy the IDS into an actual production environment, where system configuration and traffic generation would not be much of a problem. However some control must be maintained to guarantee that only legitimate behaviors are recorded.

Using short sequences of system calls in itself actually reduces the complexity of comparison, especially since no system parameters are involved. The variations of possible short sequences can be greatly controlled by the window size \(k\) used to scan the traces. The variations of possible normal short sequences are further reduced if the database creation is properly controlled and the privileged processes’ operations are exhaustively tested. Those last two measures are the main determinants of how well this detection method handles the false errors.
3.1.4 Performance and Impact of Detection Method in Practical Implementation

In order to assess the effectiveness of the detection method based on monitoring short sequences of system calls, the criteria outlined by Debar et. al. [10] stated earlier in this paper will be used as the system performance measure:-

1) Accuracy – The accuracy of an IDS based on this detection method depends on how complete all the possible normal sequences of system calls generated by the privileged processes are recorded and stored in the database. If all the normal sequences were properly recorded, there would be little possibility that an abnormal sequence will be overlooked. However, this conclusion hinges on the assumption that only abnormal sequences constitute a possible attempt to compromise the system. Attacks that are based on normal operation, for instance attacks done by misfeasors, will be overlooked. Realistically, an absolutely complete database is extremely difficult to achieve and if the database is too large, it could be plagued by the two types of false negative problems exposed in the last section.

2) Performance – This detection method is can be considered to have a high performance because it scans the process’s system call traces as they occur. It is possible to detect an intrusion while in progress especially if the intrusion is based on the exploitation of privileged process vulnerabilities. But all this largely depends on the window size \((k)\) used to scan the traces and the database generated by the window. If the window size \((k)\) is small the database created is also small, allowing the sequence comparisons to be done faster. However, a small sequence might overlook an attack that is based on a longer sequence. This can be prevented if the window size \((k)\) is larger, and comparisons are more thorough. However speed suffers in this case because the database is larger and the comparisons would take longer.
In their own laboratory tests on known system exploits, Hofmeyr et. al. have determined that a window size \((k)\) of 6 would be optimum to document all sequences of normal system calls while not sacrificing comparison speed. This is again based on the knowledge that the acceptable normal behavior of a process is a small subset of all legal operation.

3) *Completeness* – The completeness of this detection method largely depends on the level of accuracy. This method is actually able to detect all deviations from normal behavior without fail, assuming the window size \((k)\) is optimally specified and the normal database has all possible sequences of normal behavior. The other determinant of the completeness is the anomaly signal threshold. As long as the threshold is set to a level where it can confirm an intrusion without generating any false alarms then this detection method can be said to be complete.

4) *Fault Tolerance* – The measure of fault tolerance depends on the IDS software built on this methodology. The constraints would be the host system’s disk capacity and also the buffer length used to store the traces before or while they are being scanned. Unfortunately, this method may crash if the user action forks and re-forks a multitude of privileged processes in a single operation. The increased memory usage in this situation could exhaust the systems resources and bring the entire operating system to a halt.

5) *Timeliness* – An IDS based on this method of monitoring short sequences of system calls can be considered very timely because it processes the traces as they occur. However its true measure of timeliness is again based on the anomaly signal threshold that is specified by the IDS administrator. If the threshold is set too high, intrusions that produce low anomaly signals will be overlooked and the alarm will only be triggered if the anomaly signal
increases. If the signal is too low, the IDS administrator can potentially be
plagued by repeated false alarms, especially if the normal database is
incomplete.

Initially, users will hardly be effected by the operation of an IDS based on this
method because the creation of the normal database is a transparent process even in an
actual production environment. Once again the concern would be on the part of the IDS
administrator, where he must guarantee that no intrusive behavior will be exhibited
during the database creation period. However, problems will occur as the number of
privileged processes increases. To illustrate, consider the initial installation of an IDS
based on this method. In order to reduce resource usage, it is not absolutely necessary
that every single privileged process be monitored. The IDS administrator can decide to
only monitor privileged processes where a large number of vulnerabilities and exploits
are discovered. This also reduces resource use if more privileged processes are forked
from a single process. Should the IDS administrator later decide that more processes
have to be monitored, the entire system will have to undergo the first stage of this
method again: to recreate the normal database. Fortunately, it would be possible to
have the IDS actively monitor while creating the database for a new process at the same
time, but further study would need to be done to determine how to do this effectively.
The usefulness of this detection method also depends on the tracing facilities available
in the host operating systems. This detection methodology was tested and produced on
UNIX where tracing facilities are accessible and the privileged processes are very well
defined and documented. But in operating systems such as the Windows family where
tracing facilities are unavailable and privileged processes are obscure this method might
prove useless. However, if all future operating systems have proper tracing facilities
and well defined processes, this detection method will still be effective because it does
not require specialized knowledge of any particular program.
3.2 Behavior-based intrusion detection through File Integrity Checking

3.2.1 Description of Methodology

One of the earliest forms of intrusion detection ever developed is a method called file integrity checking. This methodology has been used in UNIX servers all over the world as early as 1992 to look for anomalies in file systems; hence making it fall under the category of behavior-based intrusion detection. In a nutshell, it is a method of making sure that a file or set of files being monitored has not been tampered in any way.

At that time, programs based on the file integrity checking method were written in response to a intrusion activities originating from the Internet, and the repeated occurrences of intrusions on the same servers following the original attack. This stream of attacks were the result of “back-door” programs, modified versions of usually harmless programs that have been re-written to provide the intruder with a method of bypassing a server’s authentication and auditing mechanisms. These “back-doors” spread amongst computer systems either through computer viruses, e-mail attachments containing Trojan horses, or through direct file replacement taken place in an earlier undetected intrusion. Manually finding these back doors proved very tedious to system administrators because the changed files might share the same file-sizes and modification dates as the originals – two properties of a program normally scrutinized in manual file tampering checks. This problem precipitated the need for an automated tool to regularly monitor sensitive files and check its contents for any signs of unauthorized modifications.

Initially, integrity checking was done by just having mirrored servers operating in parallel with the main server. It was identical in both hardware configuration and
software content. A simple script was periodically executed that would compare files between the main server and the mirrored server to detect any anomaly in the file's contents. Although this was fairly simple to do in terms of computation, it was very expensive from both the financial and resource usage points of view. In addition it was not scaleable, any hardware expansion would have to be similarly mirrored, and soon the method became infeasible.

To save costs, the idea of using unique file identifiers stored in small databases was explored. The idea was that if the file identifier could be created on an existing file and that identifier is compared against a stored version, any alterations to the file could be determined. This technique could be similarly applied to automatically recognize whether files have been added or deleted from a system. It soon came to be realized that any changes on a file system can be detected by storing a value calculated from the contents of the files being monitored instead of the entire file itself. This calculated value is called a fingerprint or signature of the file, and can be efficiently stored for later comparison against actual file contents. Any minute changes done to a file or its properties would cause the signature to change sufficiently large as to make a chance "collision" or accidental duplication of a signature to be unlikely. Finally, in order to protect a file signature from duplication, the signature functions used for this purpose had to be computationally easy to perform, but infeasible to reverse.

Thus the first software packages based on the method of detecting intruders through file integrity checking were developed and put into service to assist intrusion detection. The objective of this method was to detect and inform system or IDS administrators of notable changes, additions or deletions to the file system. Unlike mirrored systems, it has to be scaleable enough to perform equally well in small networks that contain a few nodes and in enormous enterprise wide systems that consist of hundreds of machines. To guarantee its own fault tolerance, tools based on this
method must not rely on auxiliary programs to function. Reliance on external programs could subject the integrity checking tools to the vulnerabilities of the helper programs, which might themselves be programs that are being monitored by the file integrity checker itself. Another design criteria was added; file integrity checking tools should be able to run without dependence on privilege. This would let normal users of the system assert additional control to the level of intrusion detection on their own private set of files. Furthermore, the tool in its entirety must not allow the host system to be compromised whether it is read or executed by an actual user or an attacker. It does this by only reporting file changes and not effecting any changes itself. A sample structure of an IDS that is based on file integrity can be seen in Figure 4 below.

Figure 4: Basic structure of an IDS tool based on the File Integrity Checking method.
In order to fulfill these design criteria, intrusion detection tools built around file integrity checking methods have three distinct components:

1. **Configuration:** The selection of items to monitor and scalability issues.

2. **File signature:** The selection of signature functions used to generate unique file identifiers used for comparisons.

3. **Database:** The methods of storing file signatures, and issues pertaining to updating and maintaining the file integrity checking database.

To better understand how intrusion detection by file integrity checking works, each of these three parts will be broken down to their component algorithms. The first part is an exploration of the configuration component.

1. In most file integrity checking tools, data pertaining to monitoring items are specified in configuration files that may be exported and shared among other connected systems. This method supports reuse of configuration information based on the common characteristics of the systems being monitored. The configuration file will also allow unique and special-case configurations to be done at hosts that need a higher level of scrutiny.

2. The configuration file may contain lists of single files to monitor. It may also contain more compact and inclusive configurations that encompass entire directories. In addition to that, customizations that are machine specific can also be done, as can be seen in the following sample:-

<table>
<thead>
<tr>
<th># file/dir</th>
<th>selection-mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>/etc</td>
<td>R # all files under /etc</td>
</tr>
<tr>
<td>@ifhost solaris.cs.purdue.edu</td>
<td></td>
</tr>
<tr>
<td>!/etc/lp</td>
<td># except for printer logs</td>
</tr>
<tr>
<td>@endif</td>
<td></td>
</tr>
<tr>
<td>/etc/passwd</td>
<td>R+12</td>
</tr>
<tr>
<td>/etc/matb</td>
<td>L # dynamic files</td>
</tr>
<tr>
<td>/etc/motd</td>
<td>L</td>
</tr>
<tr>
<td>/etc/utmp</td>
<td>L</td>
</tr>
<tr>
<td>=/var/tmp</td>
<td>R # only the directory, not its contents</td>
</tr>
</tbody>
</table>
3. The selection-masks determine the type of function used to generate and check the file signatures.

In order for the configuration file to be properly specified, the administrator must also select the functions to generate the file signature.

1. The objective of the file signature is to simply indicate that a change has occurred in a file, this information would be adequate to infer on the possibility that an intrusion has occurred.

2. Because of the possibility of signature collisions, a situation arising from having two different files producing the same signature, the administrator must choose functions in such a way that an attacker or intruder is highly unlikely to be able to alter the contents of a file in such a way that it would retain its original file signature through coincidence. This could be done by intruders through a method called 'signature spoofing'.

3. The best file signatures could be generated by using message digest algorithms, which are one way hash functions that would produce a relatively short value as a result of the entire file. Message digests are computationally infeasible to reverse because of their size, which are at least 128-bits, and also because of the complexity of the hashing functions in use. Figure 5 on the next page shows the various types of signatures and how susceptible they are to collisions.
<table>
<thead>
<tr>
<th>Signature</th>
<th>Number of collisions</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5 6 7 8 &gt;9</td>
<td></td>
</tr>
<tr>
<td>16-bit checksum (sum)</td>
<td>14177 6647 2437 800 235 82 12 2 1</td>
<td>24375</td>
</tr>
<tr>
<td>16-bit CRC</td>
<td>15022 6769 2387 677 164 33 5 0 0</td>
<td>25059</td>
</tr>
<tr>
<td>32-bit CRC</td>
<td>3 1 1 0 0 0 0 0 5</td>
<td></td>
</tr>
<tr>
<td>64-bit DES-CBC</td>
<td>1 1 0 0 0 0 0 0 2</td>
<td></td>
</tr>
<tr>
<td>128-bit MD4</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>128-bit MD5</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>128-bit Snethu</td>
<td>0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5:** Collision frequencies of various signatures used in a test of the file integrity checking intrusion detection method. The table above was taken from the research done by Kim and Spafford [22].

Once the file signature functions have been chosen, the construction of the comparisons database takes place based on the configurations specified in the configuration file.

1. The database used by the file integrity checker can be encoded in standard ASCII in order to be human readable and also in order to take advantage of the standard text tools available on the host system for database analysis.

2. Since the database contains the signatures of the files being monitored, it should by itself be protected from unauthorized modifications. This can be done either by storing the database in read only media or keeping it in secure root directories. Also, the database should contain no information that might allow an intruder to compromise the integrity checking scheme.

3. The database will be updated regularly to reflect authorized changes to the file systems, therefore to ensure efficiency, updating must not require regenerating the entire database.

After having determined all three main components of the file integrity checker as shown above, intrusion monitoring can now be done through the use of scripting languages like PERL. The following is the algorithm used:-
1. Based on the selection masks specified in the configuration file, a script will initially generate a comparison database listing the message digest signatures of the selected files.

2. The script will then decide the frequency of each integrity checking period.

3. During a checking period, the script will run the same message digest function on the file currently in the system to get its present file signature, and the result is stored in a new database.

4. Once all the new file signatures are generated, the new database is compared against the original one to check for discrepancies.

5. If a discrepancy exists, it means that the contents of the file or properties of the file, such as last access time and modified date, have been changed in some way and might be indicative of an intrusion. Figure 6 below shows a sample report from a file integrity checker that is reporting a discrepancy.

6. The modified file can then be extracted from the system for further analysis.

<p>| changed: -rw-r--r-- root 20 Sep 17 12:49:43 1993 /rhosts |
|---|---|</p>
<table>
<thead>
<tr>
<th>--- Attr</th>
<th>Observed (what it is)</th>
<th>Expected (what it should be)</th>
</tr>
</thead>
<tbody>
<tr>
<td>st_mtime:</td>
<td>Fri Sep 17 12:49:43 1993</td>
<td>Tue Sep 17 20:05:10 1993</td>
</tr>
<tr>
<td>st_ctime:</td>
<td>Fri Sep 17 12:49:43 1993</td>
<td>Tue Sep 17 20:05:10 1993</td>
</tr>
</tbody>
</table>

_Figure 6_: Sample report from the file integrity checker showing a discrepancy in access time detected in the .rhosts file. The output above was taken from the research done by Kim and Spafford [22].

There have been some arguments and debates on whether file system integrity checking tools fall within the domain of intrusion detection systems. The arguments against this method of intrusion detection was that it did not fulfill one of the crucial criteria of IDS: which was detecting an intrusion in real-time. File integrity checkers
depend on periodic scanning of the file system to detect any changes or corruption. It is very possible that an intrusion could occur between scanning periods, by which time the intruder would have already exited the system. On the other side of the coin, the arguments supporting this method of intrusion detection holds that periodic scanning is similar to other intrusion detection methods that depend on the analysis of audit logs. Given a sufficiently large audit log size, the processing abilities of other 'genuine' intrusion detection methodologies would be similarly periodic. Furthermore, the end-results of file integrity checking are identical to that of IDS: the detection of an intrusion or attempts to intrude a computer system.
3.2.2 Effectiveness of Detection Method against Intrusion Types

The greatest benefit with this method of intrusion detection through file integrity checking is that it is able to pinpoint any occurrences of changes on its monitored files such as additions, deletions, and even very minute modifications. As mentioned in the previous section, this is made possible through the use of one-way hash functions that could confirm the integrity of a file. Of course, file integrity checking is only as good as the algorithm used to determine the frequency of checking and also the span of observation specified in the configuration. The algorithm given before is an example of a generic scanning routine that is commonly used. This algorithm could be modified for more customized reporting of file conditions, automated alerts, and any other personalizations that the IDS administrator will deem necessary. This section examines how intrusion detection through file integrity checking fares against the various classes of intruders.

External predators

1) Attempted break-in – Apart from outright password guessing, where an intruder would try to guess weak passwords in default system accounts, an attacker might initiate a break-in on a system by first logging into an anonymous or guest account and try to steal the password file, such as ‘passwd’ in UNIX. An attempt to copy the file or open it in a text editor would cause the ‘last accessed’ timestamp property to change, as can be seen in many operating systems. A file integrity checker can be configured to check for any changes in file access times by including its ‘last accessed’ timestamp as a value to generate the file signature through the one-way hashing function. A file with this special configuration is called a “trap file” and it acts as a tripwire to be placed against snooping intruders. “Trap files” can be monitored with a higher frequency compared to other files,
perhaps once per minute, so that the IDS administrator can be alerted each
time the file’s timestamp is unavoidably updated by an access attempt. By
setting tripwires such as these on other similarly sensitive files, any attempt
to break-in from outside the organization can be easily detected by this
intrusion detection method.

Internal predators

2) *Masquerading or successful break-in (Masqueraders)* – If an intruder
manages to successfully break into a user account through some other
method aside from the one mentioned, it usefulness of this intrusion
detection method would largely depend on the subsequent activity of the
intruder. If the intruder has successfully achieved his goal of breaking into
the correct user account, and proceeds to steal the information contained in
the account, he would fall into the category of *misfeasors*. If the intruder
needs access to a higher level account of access to unauthorized file areas,
he would be exhibit *clandestine user* behaviors. However, if the intruder’s
main goal was just the break-in itself, then this detection method would not
see anything as amiss and would not cause any alerts. There is an exception
with this situation however. The configuration files used in file integrity
checking could include entries covering user directories and files. This may
be necessary of user accounts that contain sensitive company information,
like payroll or employee records. The file integrity checker can be used to
log any instances of changes in these data areas. Although this would not
produce an alert of any sort, the audit logs produced would inform the
legitimate owner of the files that those files have been accessed or modified
while he was away.
3) *Penetration by legitimate user (Clandestine users)* – An intruder may find it necessary to gain access to higher-privileged user accounts if the account he has broken is not privy to his target data areas. He could accomplish this by adding modified versions of programs into the systems that would create back-doors for the intruder. This is where this method of intrusion detection shines. File integrity checking can and will effectively detect any modified files introduced by the intruder no matter how trivial those modifications might be. The one-way hashing algorithms used to produce the file signatures guarantees this. Therefore, an IDS administrator can easily locate and remove any files that are suspected to contain back-door code. This is only possible, of course, if the files have a stored file signature in the comparisons database to begin with. Otherwise, the modified file will be over looked by the IDS.

4) *Inference by legitimate user (Clandestine users)* – The method of intrusion detection through file integrity checking is ineffective in detecting a clandestine user with lower-privileges inferring the contents of data areas in a database\(^2\) that require higher privileges. This is because inference is usually done by using query languages like SQL – a sufficiently complex query might retrieve the data that the intruder wants. The file integrity checker can only see that the database file has been accessed, which may not be a problem from the IDS administrator’s point of view since the user has legitimate access to database to begin with. An exception exists if the database program used stores data tables in actual individual files (e.g. Microsoft FoxPro). If secure data areas are carefully segregated into its individual tables, hence individual files, these files can be similarly

\(^2\) The term 'database' in this case refers to a user database and not the comparisons database used by the file integrity checker.
configured to be monitored for any form of unauthorized viewing or modification. How effectively this could be done depends on the database software in use, which falls outside the scope of this research report.

5) **Leakage by legitimate user (Misfeasors)** - To some extent, intrusion detection with file integrity checking might be able to detect misfeasor activity if local security policy limits the access times to sensitive data. This is an uncommon activity, only used to protect extremely sensitive data areas. Such a security policy may, for instance, specify that access to these data areas are only allowed within office hours, when there is enough staff available to act as checks and balances to observe the legitimate user's activity. Any violation to this security policy can be easily detected by the file integrity checking method especially if the file is accessed, modified, or copied outside the specified working times. However, this is a tedious and limiting process, especially if the company views all data as sensitive but does not want to impede on the flexibility and freedoms of its workers. As such, detection of misfeasor activity through this method is only feasible if there is enough urgency to protect the data. It would be infeasible to monitor all data or files in such a way, as this may produce many false positive occurrences that are the result of erratic, but legal, user behavior.

**Malicious process**

6) **Trojan horse (Malware)** – The existence of a Trojan horse or its activity can be easily detected by this method of intrusion detection, although usually only *after* the Trojan horse’s destructive payload has been released. It might not be effective in finding the actual Trojan horse file itself. This is because the Trojan horse may be introduced into the system as a legitimate file such as a new e-mail attachment, which does not have a file signature in
the comparisons database to compare to. However, a file integrity checker is able to report, though not prevent, the extent of the damage or modifications done by the Trojan horse. If the Trojan horse replaces a legitimate file with a modified file that provides back-door access to an intruder, this back-door program can be found with ease.

7) *Virus (Malware)* – Viruses are very easy to detect with the file integrity checker. This is because viruses are designed to add or append destructive code in addition to infection routines to existing files. The file integrity checker can also determine the extent of the spread of the computer virus by reporting all modifications that have been detected since the first infection. Again, this is made possible due to the use of the one-way hashing algorithms. A file infected with the virus would inadvertently alter its file signature and signal a possible modification to the IDS administrator. This is also useful to IDS administrators if and when they decide that they have to take measures to contain the epidemic. An IDS based on file integrity checking serves as a dual function in this case as it can also discover the existence of new viruses that have not yet been discovered by anti-virus vendors. It is also just as effective at detecting polymorphic computer viruses that mutate to show a different virus signature to avoid detection by anti-virus software. As a result, an IDS based on this method would greatly compliment anti-virus software and virus detection efforts.

8) *Denial-of-service (Malware)* – File integrity checking, unfortunately, is not able to detect any attempts at denial-of-service. This is because DoS attacks exhaust the resources used by the system, whether it be disk space, free memory, network bandwidth, or even computation time. This failure to detect is the result of its design itself. A detection method based on file
integrity checking depends on modifications done on a single file to work. If no modifications are done, then the IDS would not see anything amiss. A DoS does not tamper with existing files, even though it uses up file resources. Furthermore, file integrity checking depends on the static nature of files as its basis to detect anomalies – denial-of-service attacks do not effect the nature of the files, rather it effects its environment.

The analysis above shows that the strength of an intrusion detection method based on file integrity checking lie with monitoring the stability of the file. Any forms of intrusion that depend on the replacement or modification of files to provide unauthorized access can be easily discovered. The use of file signatures is also integral in detecting non-modification activities such as unauthorized viewing or copying. Not mentioned previously, one of the activities commonly practiced by intruders is log file reduction. This is done to erase any evidence of intrusive activity on a computer system. File integrity checking is unique in that it can also detect any reduction in these log file sizes. However for this to work, some modification would need to be done with respect to the signatures used for these log files. It would be unsuitable to use one-way hashing algorithms as a file signature as the value produced by such functions might not be indicative of a reduction, even though it is indicative of a change. The greatest weakness of this method of intrusion detection is that it will not detect any intrusions that do not effect the file system. For some intruders, it would be enough to just break into a user account, have a look at its directory structure and leave without modifying anything. Such information can be sold to other parties who might be interested in knowing the user’s directory contents. This type of intruder or masquerader cannot be detected by this IDS method. The effectiveness of the method also depends on the frequency of detection used to observe the integrity of the files. If the frequency is done after long intervals, it would reduce the resource and processing strain on the host.
system. However it would also give an intruder ample time to do more destructive activities. On the other hand, timely intrusion reports can be produces to elicit protective responses by increasing the frequency of the detection periods. Unfortunately this would produce a strain on the system in terms of resources. The IDS administrator must determine the balance to effectively detect intruders and protect a computer’s file system.
3.2.3 Capacity of Detection Method in Overcoming Error Types

As with any intrusion detection method that depends on a database of known acceptable behaviors or attributes, an intrusion detection method based on file integrity checking is prone to the possibilities of false positive and false negative errors.

Usually the comparisons database is generated immediately after the configuration file is completely specified by the IDS administrator. Because the database is stored in standard ASCII format to speed up signature comparisons, it has to be protected from unauthorized changes by intruders. As was suggested before, this can be done by storing the database in read-only media, such as a CD-ROM, allowing easy machine access to the original signatures but preventing changes. However, these databases need to be constantly updated to reflect legal changes done to the files, such as when new user information is added into the 'passwd' file. This will create a constant procedural difficulty, especially if the database is stored in read-only media. As a result, the more realistic option is to store the database in only superuser accessible accounts, to deter modification attempts by intruders and also to allow easy updates.

The database generation method described above could be a cause of false positive errors. The initial generation of the comparisons database can generally be considered robust. Files are checked against their original state without problem. However once the first 'legal' modification is done to a file, when would this modification be reflected in the comparisons database? If the update is not done before the next checking period, these legally modified files would show up as a possible intrusion alert; a false positive. This problem can be prevented by having two databases running in tandem, the original comparisons database and a new 'updates' database. A 'legal' modification can be assumed to have occurred if the user modifying the file actually owns the file to begin with. If a file has been legally modified, the file integrity checker would add its new signature into the new database right after the changes have
been made. When the integrity checker scans for correct signatures, it first looks at the new database to see if the filename exists. If it does, the existing file signatures are would pass the comparisons test as the new database reflects the update. If it doesn’t, comparisons would be done against the original database – a signature mismatch here would positively indicate that the file has undergone unauthorized modification, and an intrusion alert can be sent. Similarly, legitimate additions to the users file collection will be added to the new database and its signature will be recorded. The IDS administrators task would now be to determine when the original comparisons database and the new ‘updates’ database will be merged.

But this is not a foolproof solution. What if the files were modified as a result of masquerader activity? An intruder that has gained access to a user account would most likely have ownership of all the files in the directories of the user account. He would inadvertently have the privilege to modify the data would also add an updated signature into the new ‘updates’ database. This would now cause false negative errors. Files changes within the permitted capacity of the user account, even though the account is being exploited by the intruder, would now generate a consistently matching file signature. This is because the file integrity checker sees this as a legal update and adds the entry to the new database. This is a far more critical problem than the previously mentioned false-positives problem, as it allows a modified file, possibly a back-door file, to perpetuate in the file system without detection. This is the most obvious weakness of this method of intrusion detection, it assumes that every ‘legal’ change to a file was done at the behest of the file owner. There are no mechanisms in place to see whether the user being logged on at the time is not an intruder, unless as previously discussed, the user attempts to go beyond the boundaries of his permission level by modifying files which he does not own. One way to solve this false negatives problem would be to treat is as a misfeasor situation. A previously suggested, sensitive
data might only be allowed to be modified within a certain time period where the
identity of the user can be verified, like normal working hours for instance.
Modifications to the monitored file done outside of this time period could be seen as an
unauthorized modification and would not be updated in the new database. Hence, it is
seen as an intrusion attempt. Although, it is a workable solution to combat false
negatives, it might not be a viable option in actual implementation. For effective
intrusion detection, this solution must be applied to all files and user directories. But
once again, this would impede the flexibility of the user's working habits by limiting
the hours when they are allowed to work.

A bigger problem with false negatives could occur if the intruder has access to
the root account that is able to directly modify the contents of the comparisons
database. Although unlikely to happen in normal masquerader or clandestine user
intrusion attempts, due to the possible inclusion of the file access time limitation
mentioned in the last paragraph, it could happen if the owner of the root account is the
misfeasor himself. He could modify a file and run the same one-way hashing algorithm
to produce a signature. This new signature could then be written over the original
signature directly – effectively making the IDS see the file as legitimate and unchanged.
The only way to solve this form of false negative error would be to register the file
integrity checking program as the actual owner and mirror the original comparisons
database to prevent any modifications on either copy.
3.2.4 Performance and Impact of Detection Method in Practical Implementation

To measure the extent of performance in practical implementation, the criteria outlined by Debar et. al. [10] as described in Section 2.4 of this research report will be employed. Here, an assessment of the effectiveness of file system integrity testing and how it holds up as an intrusion detection method will be made.

1) **Accuracy** – The previous discussion on how file integrity checking handles the two error types, namely false positives and false negatives, has shown that the accuracy of this intrusion detection method depends on the configurations made by the IDS administrator with regards to the generation of a new ‘updates’ database and its subsequent merging with the original comparisons database. If rules regarding file ownership such as a criteria to update a stored file signature is not implemented, the IDS will treat every modification as an intrusion. In some rare circumstances, this might actually be the goal of the IDS administrator, such as in archival systems, where updates to a file are not expected to occur at all. However, on normal systems, where files are updated regularly, this would cause many instances of false positives and thus degrading the accuracy of the detection method. Therefore, it is imperative that the IDS be supplied some form of global filter, like the file ownership exception, to control the output of the intrusion alerts and generate only reports of genuine interest.

2) **Performance** – Because the databases storing the file signatures are in plain ASCII format, this causes the comparison against current file signatures to be rather fast. The measure of performance would now lie on the frequency that the IDS scans the file system and checks for mismatches. This becomes a balancing act for the IDS administrator. Bear in mind that each file being compared has to undergo the same algorithm used by the one-way hashing
function, and that this process has to be repeated at every iteration of the file integrity checking process.

Scanning the file system infrequently would reduce the overhead imposed on the system resource such as computation time and memory usage, but the performance of the IDS would falter because the large time gap could be a window of opportunity for an intrusion attempt. The situation is reversed if the scanning is done far more frequently, there will be little chance that an intrusion attempt could occur between periods, however the system resources being used would be quickly exhausted.

3) Completeness – The level of completeness in this method of intrusion detection through file integrity checking depends on its ability to reduce the chances of a false negative error from ever happening. Based on the discussion in the previous section regarding the necessity to protect against intruders such as masqueraders and misfeasors that may subvert the validity of a modification to a file, if measures are not taken then there is a very high probability that intrusions would go undetected. Granted, there are concerns that these measures would limit worker freedoms, but if this method of intrusion detection were to be used, then it would have to be a compulsory measure. Otherwise completeness would suffer.

4) Fault Tolerance – Decisions regarding the storage of the signature databases is the sole determinant of the fault tolerance level of this intrusion detection method. This is because the databases are kept in ASCII format to improve speed and reduce comparison complexity. If the databases are stored in very secure and restricted locations where all activities pertaining to database updating can be meticulously controlled, then this method can be said to have an extremely high level of fault tolerance. The file signature
generation algorithms themselves are also very fault tolerant, assuming the
IDS administrator chooses to use irreversible one-way hashing functions that
would not overlap or cause signature collisions.

5) *Timeliness* – Like performance, the timeliness of this method depends on the
frequency of the comparisons. Fortunately, depending on the scripting
method used to report comparison results, changes to certain files might
elicit different reactions and alert levels. For example, reporting scripts can
be customized to instantly inform the IDS administrator if system binaries
have been changed, whereas changes to low-level audit logs would be noted
in a digest that is produced upon completion of each checking cycle.

As mentioned in the beginning of the analysis on this method of intrusion
detection, file integrity checking is one of the earliest forms of intrusion detection. The
simplicity and speed offered by this method makes it useful not only in intrusion
detection software, but also in other software such as virus scanners, file transmission
programs, and other applications that need to check on the integrity of a file. One of the
most popular file integrity checkers used right to this day is ‘Tripwire’ a free module
that is used in UNIX systems. Unfortunately, this method is already showing its age.
As was discussed earlier, this method of intrusion detection has a great weakness
against misfeasors and masquerader types of intrusions, making it very prone to false
negative errors and undetected intrusions. The growing need to make businesses run
longer into the night makes access time control more and more infeasible. Therefore, it
would be best if file integrity checking is limited to scrutinizing totally static files and
system binaries. A second intrusion detection method would have to be deployed
alongside a file integrity checker in order to monitor sensitive files that are more
dynamic.
3.3 Knowledge-based intrusion detection using State Transition Analysis

3.3.1 Description of Methodology

While behavior-based intrusion detection methods rely on anomalies from a known characteristics of a user or system behavior as its method of locating an attempt to break-in, knowledge-based intrusion detection takes a more direct approach, by using the characteristics of known intrusions techniques as their detection basis. One such intrusion detection method called State Transition Analysis was put forward by Koral Ilgun, Richard A. Kemmerer, and Phillip A. Porras in 1995 in their paper entitled “State Transition Analysis: A Rule-Based Intrusion Detection Approach”. This intrusion detection method is based on the premise that every successful intrusion technique shares two common characteristics:

1. An attacker or intruder must possess minimum prerequisite access to the system.
2. Their intrusions result in the acquisition of previously unpossessed abilities.

If what is now possessed could be known, then the aspect of the computer's security that has been compromised by the intrusion could effectively also be known. The method is used by a rule-based expert system, called a State Transition Analysis Tool (STAT) which is designed to seek out known intrusion patterns in the audit trails of networked computer systems.

State transition analysis views computer system intrusions as a sequence of actions performed by a intruder that brings the system from some initial requirement state to a target compromised state – the state of the system before the intrusion being the initial requirement state, and state of the system after the intrusion being the compromised state. The actions mentioned here are called signature actions. They are the key actions that are most responsible for a successful intrusion to execute. If any
one of these signature actions were omitted from the attack sequence, it would prevent the intrusion from successfully completing. These actions are itemized in one or more intermediate state transitions taken place between the initial and compromised states. The complete picture of the intrusion and the critical steps that allow the system compromise to be achieved can be represented graphically as a state transition diagram as shown in Figure 7 below.

Figure 7: A sample of a State Transition Diagram. Sr and Sc signify the uncompromised and compromised states. Sc-n show intermediate states that lead to the compromise.

The state transition diagram is the key to understanding how an attack occurs in state transition analysis. It represents the sequence of signature actions that move the system from the initial requirement state, through one or more intermediate states, into the final compromised state. System state are those attributes of the system that are necessary to signify initial, intermediate and compromised states of an intrusion. Most rule-based systems depend on exact sequences of entries in the audit logs to positively identify an intrusion or an intrusion attempt. Any variation to the log file sequences, either as a result of some variation of the attack pattern or network congestion, would cause the IDS to fail to identify an intrusion. To avoid this problem from occurring, state transition analysis requires that the audit trails be preprocessed, extract the state transition information recorded within them, and compare it to a rule-based representation of the known intrusion techniques. These rules are generated through an interpretation of the state transition diagram.
Before a state transition diagram and also the subsequent rule-base can be constructed a command-by-command sample of the exact sequences taken by an intrusion technique must be obtained. From it, the analyst is required to derive the minimum number of signature actions from the intrusion sequence and to organize them visually. The following is the method of creating a state transition diagram as described by Ilgun et. al. [21], it starts with basic diagram construction:-

1. First, identify the initial requirement state (Sr) and the target compromised state (Sc).

2. Below these to states are listed the *assertions* that describe the condition of the system before and after the intrusion is executed.

3. These states are then connected by a dashed arc that represents the yet to be identified sequence of intermediate state transitions and signature actions that occur during the execution of the intrusion.

4. The signature actions that lead to the compromised state are then individually identified. The key is to find only the most minimum number of actions performed that can be used to accurately represent the penetration. These actions refer to how a specific change within the system is achieved. The state transition diagram would not be incorrect if a non-signature action is added, however it would make the diagram more specific. If a diagram is more specific than it should be, it would reduce the number of variations that can be done to duplicate the intrusion, thus making the resulting rule base more rigid.

5. The best way to identify the signature actions of an intrusion sequence is to dissect the intrusion scenario in reverse order.

6. Once a signature action is discovered, a new intermediate state transition (Sc-1) is added between the initial requirements state (Sr) and the compromised state (Sc). The assertions of the system condition at that point is listed below Sc-1.
7. A solid arc is now added from Sc-1 to Sc and is labeled to show the action that is done by the intruder to achieve the compromised state.

8. The signature action that is to precede Sc-1 is then to be located and similarly added between Sr and Sc-1 and is labeled as Sc-2. The same set of steps are repeated until the diagram reaches Sr.

The completed state transition diagram can be see in Figure 8 on the next page. Once the state transition diagram is created, the rule base entry to the IDS can then be constructed.
Figure 8: Here, a state transition diagram is used to dissect a known intrusion into recognizable state assertions. This graph was extracted from the research done by Ilgun et. al. [21]
All intrusion scenarios that can be reproduced through command execution can be efficiently dissected using this method. However, Ilgun et. al. [21] have found that two key factors must be fulfilled for an intrusion sequence to be correctly represented in a state transition diagram. Those conditions 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

The first factor requires that an intrusion scenario cause the system to produce an identifiable compromised state. This means that the system compromise must be represented by singular analysis of system attributes alone. As mentioned before, the method of intrusion detection through state transition analysis requires the use of system auditing facilities, which records system attribute state changes. IDS tools developed around the state transition analysis method compare stored state transition diagrams, or more specifically rule-base entries, of known intrusion techniques to the state changes produced by users by using audit trail information on the system as input. Unfortunately, auditing facilities in most operating systems do not record every single change and reference to system attributes. This fact alone infers the necessity for the second key factor revealed previously. Some forms of system compromise are not identifiable, only by having knowledge that is outside of the system’s execution domain could the compromised state be distinguished. This knowledge may be in the form of the identity or intention of the intruder, pieces of information that cannot be ascertained through system analysis. Intrusions such as these are executed beneath the visibility of audit data analysis tools and hence falls outside the scope of state transition analysis.

Assuming the system has the necessary audit facilities and the IDS administrator or security manager has a list of known intrusion techniques and are able to break them down into state transition diagrams, an IDS tool based on state transition analysis can
now be implemented. The basic structure of a state transition based IDS tool can be seen in Figure 9 shown in the following page. The Audit Record preprocessor reads all relevant audit files and processes them into a state transition analysis format similar to the rule-base entries. The Inference Engine gets these processed records and compares them to the Knowledge Base. The Knowledge Base contains the fact base and the rule base. The fact base contains information regarding similarities in file or system vulnerabilities that could be exploited by one or more unique types of intrusions. The fact base selects the rule-base entries that conform to the pattern in the preprocessed audit trails that is presented by the Inference Engine. If the Inference Engine finds a match or something close to a match, it alerts the Decision Engine to take action. The Decision Engine stores appropriate action information that has been pre-configured by the IDS administrator. Actions taken by the Decision Engine may range from preempting the last state of the intrusion or to alert the IDS administrators. The entire IDS tool is controlled by the IDS administrator through the interface.
The core intrusion detection operations in this IDS tool is performed by the Inference Engine using information garnered from the Knowledge Base. As previously mentioned, the rule-base entries are actually derived from the state transition diagrams created using state transition analysis. The rule-base consists of signature actions and state assertions that are chained together as derived from the state transition diagrams. The Inference Engine uses a table to detect a possible intrusion. Each row on the table
(n) represents a known intrusion scenario, while each column represents the state assertions steps (S) that are fulfilled for an intrusion scenario. A sample table can be seen in Figure 10 below.

<table>
<thead>
<tr>
<th></th>
<th>$S_1$</th>
<th>$S_2$</th>
<th>$S_3$</th>
<th>$S_4$</th>
<th>$S_5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>h</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n+1</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n+2</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10: A sample of an Inference Engine table. The numbers 1, 2, h, n..., n+1 indicate the different intrusion techniques recognized by the IDS. $S_1$ to $S_n$ are placeholders for states that need to be achieved to reach the compromise. X marks the state already achieved by the intruder. The table above was taken from the research done by Ilgun et. al. [21].

The following is the method in which the Inference Engine uses the preprocesses audit trails to detect intrusion activity:

1. Each step of user activity creates a state assertion that is derived from the audit trail.

2. If the state assertion matches the first state seen in a rule-base entry, the first column of the row (n) on the table ($S_1$) represented in the intrusion technique is marked.

3. The row and its marks are duplicated in a new row (n+1). This duplication is necessary because the original row still represents part of another instantiation. This is because the state assertion still holds, therefore even if preceding state assertions do not match a rule-base entry, other trailing state assertions still might; hence the original vulnerability exists.
4. If the next state assertion matches the continuing state in the rule-base entry, the row is similarly marked (S2) and duplicated (n+2). If it does not match, the row is maintained.

5. If a state assertion no longer holds, either by file deletion or any other activity, the row and other duplicate rows that depend on that state assertion is deleted.

6. If the state assertions continue to match as seen in the entire rule-base chain for that intrusion technique, then an intrusion alert is sent to the Decision Engine, along with all pertinent user and system information.

Intrusion detection using the state transition analysis approach actually lets the IDS administrator intervene before an intrusion or system compromise is reached. Given a rule-base entry chain, the Decision Engine can decide to either freeze the user account one or two steps before the last state assertion. This is an effective way to prevent any damage occurring on the system. This also allows the IDS administrator ample time to gather as much information about the intruder and the attack that might be useful in legal prosecution.
3.3.2 Effectiveness of Detection Method against Intrusion Types

The intrusion detection method based on state transition analysis has a unique benefit, it is not only able detect known intrusion techniques, it is also able to detect variations of these intrusions. This is because the method does not look at the actual audit records directly like most other knowledge-based intrusion detection systems do, rather it extracts information pertaining to state assertions from the audit logs and determines whether these state assertions are characteristic of an intrusion attempt. An added benefit of using state transition diagrams to dissect known intrusion techniques is the ability to preempt the final state assertion that would lead to a compromised system state, as mentioned in the previous section.

To assess how well this intrusion detection method fares against actual attacks, the categories of intruders and intrusion types bought forward by Denning [11], Anderson [1], and Brunstein et. al. [5] specified earlier in this research report will once again be used.

External predators

1) Attempted break-in – The two characteristics common with successful intrusions as outlined by Ilgun et. al. [21] earlier in this research report, attackers having rudimentary access to the system and possessing previously unheld properties, correctly corresponds to that of a basic attempted break-in by external predators. Usually, the potential intruder may only have basic access to either guest or anonymous accounts, which only provide the most basic functionality to the system. Using state transition analysis, it is easy to detect changes in the permissions and properties given to those accounts because these changes will show up as state assertions when the audit records are preprocessed. For instance, a guest account usually starts out with the most minimal access to system
areas. If additional permissions to more secure areas or file collections are suddenly obtained, then this activity will show up as a series of state assertions and can positively identify that an intrusion is taking place. IDS administrators can customize their intrusion detection packages to immediately terminate a guest account right after the first state assertion holds regardless of the intrusion technique in use, for instance, when a new permission level is achieved. This lower tolerance to suspicious activity can be safely integrated to the organization’s security policy, because terminating guest accounts would not effect regular legitimate user activity. Stricter measures like these will automatically prevent any further attempts to compromise the system or take advantage of the various vulnerabilities in the software packages because rudimentary access, the first of the common characteristics of a successful intrusion, is being denied to an attacker. An external predator will not be able to misuse guest privileges, and would have to look for other ways to break into the system, thereby making state transition analysis a very effective means of dealing with outside intruders.

**Internal predators**

2) *Masquerading or successful break-in (Masqueraders)* – In their research of state transition analysis, Ilgun et. al. have admitted that this intrusion detection method would be initially unable to detect masquerader activity. What they mean by this is that if the goal of the intruder in to snoop around to contents of a legitimate user’s file system by assuming his identity then this would not cause any changes in system states, and therefore no state assertions would be made because the intruder is within the domain of the infiltrated account’s boundaries. This is because state transition analysis or using state transition diagrams to dissect intrusion techniques makes no
attempt to profile normal user behavior. This falls in the domain of external knowledge of the attacker, knowledge that is not and cannot be recorded in any of the system’s auditing facilities. However, Ilgun et. al. also add that if the masquerader attempts to use known intrusion techniques to gain additional system access, perhaps by exploiting a system vulnerability, then state transition analysis will detect this as intrusive activity. This is similar to clandestine user activity that will be explained in the next section.

3) *Penetration by legitimate user (Clandestine users)* – One of the benefits of using state transition analysis as an intrusion detection method is that it makes no distinction between clandestine user activity and masquerader activity that attempts to gain additional access. A penetration by a legitimate user or by someone pretending to be a legitimate user is usually done by attempting to gain additional permissions to gain access to more secure areas of the system by exploiting the vulnerabilities that exist in the programs accessible by the clandestine user. As mentioned before, any attempt to obtain a previously unpossessed property will generate a state assertion. A series of state assertions that sequentially match that of a known intrusion technique will trigger a positive detection alert. However, all of this hinges on the assumption that the intrusion technique is known to the IDS administrator, and the technique has been properly dissected using state transition diagrams and stored in the IDS Knowledge Base as a rule-base entry. If the intrusion carried out by the clandestine user is a variation of a known intrusion technique, then a positive intrusion identification can be made. Similarly, even if the intrusion technique is not known but the exploitation of an unknown system vulnerability generates the same state assertions as that which is shown in a known intrusion technique, then
positive identification can also be made. However, if the clandestine user employs an entirely new penetration technique that produces an unknown or inconsequential state assertion, then the intrusion will pass unseen by the IDS and no intrusion alert will be sent. This is a false negative error and will be explained in much more detail in the next section.

4) *Inference by legitimate user* (*Clandestine users*) — The state transition analysis described in this research report is tailored for intrusion detection, more specifically detecting intrusion activity that would generate state assertions that give an intruder access to a previously inaccessible part of the system. It is not specifically intended to detect clandestine user activity that attempts to gain access to secure information through inference of a database. This is because normal database command activity, either by using query languages or stored procedures, do not generate audit records that could visibly show a state assertion if there is any. Again, this is one of the key factors to determine whether an intrusion technique can be dissected by using state transition diagrams. Therefore, in this case, state transition analysis as an intrusion detection method will not be helpful in detecting any clandestine user activity that uses inference techniques. However, if the IDS were tailored to specifically monitor database activity, possibly by integrating it with the database program, then it would be effective in detecting inference activities. This would require much customization of the IDS with specific focus on databases and monitoring database logs. Although database IDS are outside the scope of this research report, tailoring state transition analysis to suit specific system areas have already been done. One such customization is done in NetSTAT [49], an IDS
which uses state transition analysis to detect and preempt possible network based attacks.

5) *Leakage by legitimate user (Misfeasors)* – The research on state transition analysis done by Ilgun et. al. claims that this method is able to effectively detect intrusions originating from external predators and misfeasors. The first of these claims have already been proven in this study, however there is some doubt on whether this method is effective against misfeasors. Using the previously stated definition of misfeasors as legitimate users of the system who leak information normally made privy to them to outside parties, it is hard to see how this detection method could possibly make an positive intrusion detection. If the data that is stolen or leaked by the misfeasoar originally belongs to the misfeasor himself in terms of access rights, then any attempt to copy out or modify the information would not produce a signature action or state assertion. Consequently, this would not show up in the state assertion table or match up with any known rule-base entry.

*Malicious process*

6) *Trojan horse (Malware)* – State transition analysis is able to treat any activity performed by an automated process similarly to activity performed by actual human users. If a user executes a Trojan horse program that contains any destructive payload, the automated process taken by the payload will be logged by the auditing facilities under the victim’s identity. Like any actual user activity, every command executed by a user triggered automated processed will be similarly audited, and hence similarly preprocessed for its state assertions. The only difference would be the speed in which the processes take place. If the log files are preprocessed
fast enough, then Trojan horse activity can be thwarted just as well using this method. However, if there is a large amount of audit logs that need to be preprocessed, the IDS might not be able to extract the state assertions fast enough to preempt any action or compromise done by the payload of the Trojan horse.

7) **Virus (Malware)** - As with Trojan horses, viruses can also be detected by state transition analysis if it attempts to exploit a known vulnerability of a system. The added advantage of this method in detecting viruses would be the nature of viruses, it attempts to overwrite files with modified versions of the files. In a networked operating system like UNIX, it is very rare that a normal user would have ownership or write access over system executables. If the virus attempts to infect an executable by overwriting it with a modified copy, the attempt would produce a signature action and consequently a state assertion that could trigger an intrusion alert. State transition analysis will also be able to preempt any destructive activity that might be performed by the viruses. Unfortunately, the capacity of this intrusion detection method against viruses is again determined by how fast the audit logs can be preprocessed.

8) **Denial-of-service (Malware)** – It is not immediately clear whether state transition analysis would be effective in dealing with denial-of-service attacks that originate from within the network, for instance by using up system resources like computation time or system memory. If the operating system audits every resource allocation to a particular user, then it would be conceivable that a state assertion can be produced, thereby making the detection method effective. However, if resource allocations are not audited, then this type of attack would be invisible to the IDS. As was
mentioned before, state transition analysis can be modified specifically to
detect network-based attacks like DoS, which is much more common now.

But, this would be categorized as a special use for state transition analysis,
one that could possibly run alongside host based intrusion detection.

It is clear from this research that intrusion detection by using state transition
analysis is most effective when faced against a known intrusion technique or its
variants. Virus or malicious process activity could be similarly stored as a rule-base
entry to make the IDS more resistant to automated attacks. Unfortunately, the method
displays a glaring vulnerability when faced with leakage problems caused by legitimate
users, like misfeasors.
3.3.3 Capacity of Detection Method in Overcoming Error Types

Knowledge-based intrusion detection methods react quite differently to detection errors when compared with anomaly or behavior-based intrusion detection methods. While they are a very effective means of detecting known intrusion techniques, they tend to be rather ineffective when faced with a new way to break into a system. These advantages and weaknesses also apply to intrusion detection by using state transition analysis.

Because it is a knowledge-based intrusion detection method, state transition analysis sees a very low occurrence of false positive rates. State transition analysis specifies that unless an activity causes a state assertion or series of state assertions that is a possible part of a known method of attack, then that activity would not be mistaken for an intrusion. The use of a table to map out these state assertions also minimize the possibility of false alarms because the method requires that not only must the state assertions occur, but they also have to follow a sequential series as specified by the rule-base entries of known intrusion techniques. As previously mentioned, these rule-base entries are stored in the Knowledge Base along with the fact-base entries as a source of information to locate an intrusion. If a state assertion of a rule-base entry holds but subsequent state assertions do not hold, or causes the original state assertion to no longer hold, then no alarms are triggered to alert IDS administrators. An added level of flexibility is also provided by the practice of using state transition diagrams to dissect a known intrusion technique to find only the key signature actions that causes these state assertions. This prevents an IDS built around this method from seeing a non-signature, which may be more common than real signature actions, as an intrusion indicator. The IDS will also be able to detect variations of an intrusion technique. Often, potential intruders will try to fool an IDS by masking their actions with unrelated commands, either by executing one command after another over long periods of time,
or executing the attack from two different user accounts. State transition analysis will see through these masking attempts because what it looks for is state assertions caused by the actions, not by the exact audit entry generated after a command or by the identity of a user. This is made possible by the Audit Record preprocessor mentioned earlier, which reformats each audit log entry and looks for any state assertions that may be relevant. Ultimately however, the ability of this intrusion detection method in overcoming false positive errors is, of course, dependent on how well each known intrusion technique is dissected into its key signature actions and its resulting state assertions. If the state transition diagram is improperly used to break down an intrusion, either by including a non-signature action or by misinterpreting the resulting state assertion, then there is a possibility that it may signal a false positive alert.

Unfortunately, because intrusion detection by using state transition analysis is a knowledge-based approach, it is very vulnerable to new forms of intrusions. The entire method is dependent on the knowledge of how intrusions occur to work properly. This is because state transition analysis must be able to identify the state assertions that have been applied to detect an intrusion. If an intrusion technique is not known to the IDS administrator, then the only hope of a positive detection would be if the new intrusion is a variation of some previously known intrusion technique. This is one of the main advantages of the method of intrusion detection using state transition analysis – it is able to positively identify variations of an intrusion. The example presented in the research done by Ilgun et. al [21] demonstrates how a vulnerability in the ‘mail’ program allows a particular state assertion to be reached that is critical to the intrusion. Similar vulnerabilities yet unknown in other system programs may also cause the same state assertion, thereby allowing the state assertion to hold. If the unknown intrusion technique produces these assertions, then it will be treated as a variant of a known intrusion technique and can be positively detected, even though the exact method is not
known. However, if a vulnerability in a program causes a different state assertion to hold, then the IDS will not be able to detect the intrusion. The best way to deal with these unknowns would be for the IDS administrator or rule-base analyst to regularly subscribe to security advisories and newly discovered intrusion detection techniques released by security organizations like CERT or their own operating system vendors. This will allow then to produce their own state transition diagrams and the subsequent rule-base entry to detect the new method of intrusion.

The second vulnerability of state transition analysis to false negative errors is echoed in the one of the two key factors previously mentioned in the description of this method by Ilgun et. al. The intrusion or compromise must produce a visible change in the system state for a positive detection to occur. If the intrusion produces a state assertion that is a result of a signature action not logged in any of the auditing facilities of the host system, then the intrusion cannot be mapped using state transition diagrams and hence cannot be detected by the IDS. The audit logs themselves must also be protected from modification and deletion, or at the very leased be preprocessed fast enough for the state assertions to be identified and stored in a different area of the system. If the audit logs are corrupted or deleted by the intruder, the state assertion cannot be detected by the IDS and it will not see the action as intrusive. Apart from securing the audit logs, mirroring the records in a separate area as they are produced might help in guaranteeing the integrity of the records. Having the IDS preprocess both of copies of the logs would also be helpful in detecting an intruder, however this would require added system resources and might reduce the system's processing power.
3.3.4. Performance and Impact of Detection Method in Practical Implementation

The measures of effectiveness outlined by Debar et. al. [10] will once again be used to assess the performance and impact of using state transition analysis as a detection method in practical implementation. The five categories of effectiveness which are accuracy, performance, completeness, fault tolerance, and timeliness will once again be employed to see how this method holds up in actual implementation.

1) **Accuracy** – Due to the low probability of false positive errors as shown in the last section, state transition analysis can be said to have a very high level of accuracy compared to other intrusion detection methods. This is because the method must have a positive identification of a state assertion that would be a possible sequence of a known intrusion technique. To further reduce false alarms, this intrusion detection method requires that the activities taken by the user produce further state assertions that match the entire sequence of a known intrusion technique.

2) **Performance** – The performance of this intrusion detection method largely depends on the number of audit record types that the IDS is configured to monitor. For maximum coverage, it has been recommended that the IDS monitor all audit logs that are generated by the host operating system. Unfortunately this comes at the expense of processing power and resource usage. Bear in mind that the Inference Engine used by this detection method does not look at the audit log entries directly, rather they look at the preprocessed versions of the logs that will simplify the task of looking for relevant state assertions. This additional step is an extra overhead faced by the IDS. If the number of unique record types increases then it is conceivable that the IDS may not be able to detect an intrusion and preempt the compromising action in time. This would cause the IDS to no longer
operate under real-time and therefore have a low performance rate. To increase performance, the IDS administrator must choose and balance carefully exactly which audit logs to place under scrutiny. One guideline would be to monitor only the audit records that can positively show a state assertion of known intrusions as seen in the rule-base entries.

3) *Completeness* – Because state transition analysis is a knowledge-based IDS, it’s completeness is directly effected by the size of its rule-base or how many unique types of known intrusion techniques have been dissected using state transition diagrams. In a commercial package, update to a rule-base could be done periodically by having a central source dissect the intrusions into their respective state transitions and distributing them to users of an IDS based on this method. This would provide more coverage against yet to be encountered intrusion attempts. This method has an added advantage over other knowledge-based methods because it can also detect variations of an intrusion technique. As mentioned in the discussions on false negative errors previously, even if the IDS encountered an intrusion technique that exploits a yet uncovered vulnerability of the operating system, as long as the vulnerability produces the same state assertion as a known vulnerability, it too can be detected.

4) *Fault Tolerance* – As long as measures are taken to protect the integrity of the audit records and also the Knowledge Base, then this method can be said to be very fault tolerant. These measures could be either in the form of mirroring the audit records to protect from intruder initiated modification or by storing the records in a more secure area of the system. The audit log

---

1 This is a normal practice with anti-virus software where automated tools are provided to update the virus signature files. The method can be adopted for use with IDS as well. However, since freely distributing an intrusion signature essentially involves broadcasting a system’s vulnerabilities, some measure of distribution control is required on the part of the central source.
preprocessing step guarantees that the IDS never encounters a buffer overrun. This preprocessing step will automatically reject any invalid audit log entries, thereby protecting the IDS from erroneous input.

5) *Timeliness* – State transition analysis as a method of intrusion detection shines most in the area of timeliness. By correctly dissecting known intrusion techniques into its component state transitions, the IDS can be programmed to halt intruder activity one step away from achieving a compromised system state. This ‘lock-down’ technique is able to protect against any data corruption and could also be used by the IDS administrator to gather evidence against the perpetrator. Whatever action desired by the IDS administrator can be programmed using the Decision Engine that was mentioned earlier in the description of this detection methodology.

Actual implementation of this intrusion detection method in the workplace would be fairly transparent. Unlike behavior-based IDS which depend on a period of collecting records of user or program activity to establish normal behavior, this method does not need to undergo such a period. An IDS based on state transition analysis can be deployed immediately with minimum interruption to user activity. However, care must be taken in dissecting the intrusion techniques into the state transition diagrams, especially in the step of identifying signature actions. If a non-signature action is included in an intrusion rule-base entry, then normal user activity could be seen as a trigger for an intrusion alert. Although, this does not constitute a false positive alarm, because a sequence of assertions must be identified, it would cause resources to be unnecessarily devoted to monitoring a non-intrusive user. Nevertheless, as long as the rule-base entries are current, state transition analysis is a very effective method of intrusion detection and can be easily deployed to other systems without requiring much customization.
3.4 Knowledge-based intrusion detection using Pattern-Oriented Modeling

3.4.1 Description of Methodology

While both behavior-based and knowledge-based intrusion detection methods each have their own advantages in overcoming certain intruder types, both of them share a common weakness – they are usually not sensitive to context-dependent intrusion techniques – intrusion methods which involve data and privilege flows that are illegal. The lack of sensitivity to this type of intrusion may cause false negative errors when intrusions are overlooked or cause false positive errors when alarms are mistakenly raised. Shiu-Pyng Sheih and Virgil D. Gligor have developed a unique method of intrusion detection as described in their 1997 paper titled “On a Pattern-Oriented Model for Intrusion Detection”. According to them, pattern-oriented modeling allows the detection of pre-defined intrusion patterns in the system's audit records. This is done by capturing and uniformly defining the dynamic, potential flows of user data and access privileges, and looking for data and privilege flows that are characteristic of operational security problems that are the direct result of a misuse of system's access authorization.

The operational security problems mentioned here are ones that happen independent of the computer systems employed security policy or level of security assurance, and can be caused by any user types, regardless of their level of privilege. These problems occur as a result of complex interactions between users, their data, and also the computer's protection system. Examples and descriptions of these types of problems are given below, as described by Sheih and Gligor [43]:-

1) Unintended use of foreign programs – This situation occurs when the user fails to check his current path or reset environment variables that determine his
command search path. If the user is currently in a shared public directory, he might inadvertently execute a modified version of system commands planted in the public directory by the intruder. This situation might also occur if command aliases are used.

2) **Unintended use of foreign input data** – Similar to the first problem, if the user uses a public directory as his workspace, the contents of his work files can be modified by the intruder. Supposing the file in question was meant to be compiled later on, it could possibly contain malicious code introduced by the intruder, if the original author of the file does not take steps to thoroughly check the file’s contents.

3) **Imprudent choice of default privileges** – Generally, system administrators specify a set of default privileges for each user to accommodate the needs of different file-sharing patterns. Often times these default privileges can cause information leakage – a file writeable by a single user but readable by the group can made writeable by the group if the contents are extracted and copied into a new file.

4) **Use of weak protection mechanisms** – Sometimes, protection mechanisms like ‘setuid’ are visibly retained in the computer system even though they are unsafe. A user might be induced to executing an intruder-modified program that creates a ‘setuid’ file with the user’s identity. As a result, the intruder could potentially run programs from that file under the user’s identity.

Each of these problems constitutes a natural system weakness that would allow the initiation of various intrusion techniques. Fortunately, intrusion detection by using pattern-oriented modeling was specifically designed to overcome these problems.
Like most knowledge-based detection methods, pattern-oriented modeling depends on a repository of intrusion techniques stored in a form of rule-base. Each entry in the rule-base is the result of a representation by a *protection graph* that maps out protection violations seen in most intrusion techniques. As with the state transition analysis method described in the last section, this method also captures system states from the audit records. However, instead of state assertions, pattern-oriented modeling interprets all system events as different types of data and privilege flows. As a result, intrusion events can be defined in similar terms. A protection graph \((G)\) represents all the system states in the structure \(G(V, E, C, F)\). A guide to the elements used in the protection graph is shown in *Figure 11* above.

1) \(V\) stands for the set of *vertices* in the graph. There are two distinct types of vertices. The first vertex represents users or processes that initiate operations and are called *subjects*. The second vertex represents data containers like files and directories and are called *objects*. Subjects are active and can cause data and privileges to flow between subjects and objects; whereas objects are passive and cannot initiate any operations or cause flows.

2) \(E\) stands for the set of *labeled edges* that connect a pair of vertices. The edges are labeled with any of the following: \(r, w, cb, d, Ir, Iw, Icb, Id, r^*, w^*, cb^*, d^*\).
Ir*, Iw*, Icb*, Id*. Each of these labels represents a type of relation between the vertices as a result of an operation. The main direct relations are r, w, cb, and d, which denote read, write, control, and drive relations. Read and write relations show the flow of data between a subject and an object, while control and drive relations show the flow of privileges between a subject and an object. Indirect relations are derived from the original direct relations and have a label prefix I before the relation character (as in Ir, Iw, Icb, and Id). Finally, a suffix * (as in r*, w*, cb*, d*, Ir*, Iw*, Icb*, and Id*) is added to represent a past relation between two vertices and signifies the sequencing of the relations.

3) C stands for the set of protection sets. A protection set is a set of all subjects and objects that might need protection from an intrusion by a subject (user or process) outside the set. In single a protection set, object data and privileges can flow freely between the subjects and objects, however flows between protection sets should be controlled. Guest users, normal users, and superusers are examples of 3 different protection sets.

4) F stands for the legal flow matrix. The legal flow matrix consists of flow policies or restrictions to be enforced in the system. It contains rows and columns corresponding to each protection set (C) and each entry in the legal flow matrix represents the allowable data and privilege flows between two protection sets (c1 and c2). The purpose of the matrix is to distinguish between legal accesses and intrusions. If all incurred potential flows within two or more protection sets are permissible in F, then it is a legal access, otherwise it is an intrusion.

Examples of the most basic and common protection graphs illustrating simple data and privilege flows can be seen in Figure 12 on the following page.
Intrusion detection by using pattern-oriented modeling is done during state transitions by looking at all indirect relations of the system state and determining whether an intrusion pattern exists. Inherently, every subject has a direct read (r) or write (w) relation to every object it reads or writes. A subject has a direct controlled-by (cb) relation to an object it executes and also has a direct drive (d) relation with another subject if it provides an operation to be executed by the other subject. Indirect data and privilege flows, and hence indirect relations, however can occur as a result of the four inference rules below:

1) A subject that directly reads an object (o₁) also indirect reads (Ir) all other subjects or objects that write to the original object (o₁).

2) A subject or object (s₁ or o₁) indirect writes (Iw) all objects (o₂) that is directly written by other subjects (s₂) if the subject (s₂) is executed-by or written-to by the original subject or object (s₁ or o₁).
3) A subject \((s_1)\) is *indirectly controlled-by* \((Icb)\) another subject or object \((s_2\) or \(o_2)\), if the latter \((s_2\) or \(o_2\)) directly or indirectly wrote an object that directly controls the former \((s_1)\).

4) A subject or object \((s_1\) or \(o_1)\) *indirectly drives* \((Id)\) another subject \((s_2)\) if the former directly or indirectly controls or drives yet another subject \((s_3)\) that directly drives the given subject \((s_2)\).

These four inference rules can be seen in *Figure 13* shown on the following page. Indirect relations are also governed by sequencing information. These indirect relations can present protection violations that can be seen in the protection graph \((G)\).

As previously mentioned, the legal flow matrix \((F)\) can contain rules that define legal and illegal flows between subjects and objects of the various protection sets \((C)\). When the system states are extracted from the audit logs, the data and privilege flows derived from that state is mapped out in the protection graph \((G)\) and is analyzed for any violations to the legal-flow matrix rules, which would indicate an intrusion pattern. An indirect data flow from an unprivileged user to a superuser-owned system command, for example, indicates an inter-protection set data flow that is not allowed in the legal flow matrix, and therefore can be seen as an intrusion. Other rules can be implemented in the legal flow-matrix, for instance there may be a rule prohibiting the direct or indirect writing of a user’s sensitive files (like `.login` in UNIX), another rule might prohibit read or write data flows from a high security level to a low-security level or vice versa. Also, to prevent abuse of privileged processes, any superuser-level processes can be prohibited from directly or indirectly executing a file owned by ordinary users. By defining all allowable data and privilege flows as rules in the legal-flow matrix \((F)\), any violation of the rules will automatically trigger an intrusion alert.
RULE 1: A subject that directly reads an object also indirectly reads all other subjects or objects that write to the original object.

RULE 2: A subject or object indirectly writes all objects that is directly written by other subjects if the subject is executed-by or written-to by the original subject or object.

RULE 3: A subject is indirectly controlled-by another subject or object, if the latter directly or indirectly wrote an object that directly controls the former.

RULE 4: A subject or object indirectly drives another subject if the former directly or indirectly controls or drives yet another subject that directly drives the given subject.

Figure 13: The four inference rules used in the pattern-oriented modeling intrusion detection method as seen in a protection graph. For each rule, the first graph shows the existing direct relations; the second graph shows how an inference is made from the rule; and the last graph shows the resulting indirect relation. These graphs above were taken from the research done by Shieh and Gligor [43].
Sheih and Gligor have presented an example of an intrusion technique that stems from the aforementioned operational security problems. A break down of this intrusion technique into a protection graph can be seen in Figure 14 shown on the next page. The intruder preys on unsuspecting victims that venture into a public directory and is a case of unintended use of foreign data. In this intrusion attempt, the goal of the intruder is to modify the victim’s .login file.

1) The intruder creates a modified version of the ls program that would indirectly write to his victim’s .login file. He places this program in a publicly accessible directory like /tmp. This creates a direct write relation between the intruder and ls.

2) If the victim enters the /tmp directory and runs ls to get a list of files the execution would cause the victim to be directly controlled by the ls program. The third inference rule mentioned holds that this also causes the victim to be indirectly controlled by the intruder. This is not yet an intrusion because it is a common observation in file-sharing.

3) When the rogue ls program executes, it secretly modifies the victim’s .login file. The system would see this as a legitimate modification execution by a process since the process was initiated by its owner (victim). Therefore, this creates a direct write relation between the victim and his .login file. The second inference rule holds here, which states that the .login file is being indirectly written by the modified ls process. This too is not yet an intrusion pattern because it is a common observation in automated processing.

4) However a second indirect relationship is also created. Since the victim is indirectly controlled by the intruder (see step 2) and the victim directly wrote .login from the system’s point of view, the second inference rule once again holds that the intruder has indirectly written the victim’s .login file.
In this example, the protection sets (C) were not explicitly specified, however this does indicate an abuse of the victim's sensitive files. If the legal flow matrix is specified to only allow direct write relations on sensitive files, this last step would be a clear violation of protection rules. As a result this sequence of data and privilege flows is seen as an intrusion and an alarm can be raised. The basic structure of an IDS using this method can be seen in Figure 15 that follows this page.
Figure 15: Basic structure of an IDS tool based on the Pattern-Oriented Modeling method.
3.4.2 Effectiveness of Detection Method against Intrusion Types

Intrusion detection by using pattern-oriented modeling is just as good at detecting pre-encoded intrusion techniques as any other knowledge-based detection method. However, unlike methods that depend on exact intrusion signatures for positive identification of an attack, pattern-oriented modeling will allow an IDS based on it to detect yet unknown intrusion techniques. The method is able to do this because many intrusion techniques produce the same illegal data and privilege flows. If the prohibition to these flows are explicitly stated as a security flow policy in the legal-flow matrix, then it is conceivable that variations of an intrusion technique that produce identical illegal data and privilege flows could be similarly detected.

Once more, the suitability and effectiveness of this intrusion detection method in overcoming actual intrusion attempts is to be assessed. The categories of intruders and intrusion types specified by Denning [11], Anderson [1], and Brunstein et. al. [5] earlier in this research report will again be employed for this purpose.

External predators

1) Attempted break-in – The intrusion demonstration presented by Sheih and Gligor [43] in the last section is a classic example of what an external predator or intruder from outside the system would do to access a secure system. One characteristic behavior of external predators would be enter the system anonymously, gather as much information as possible in the public areas of the system, and leave behind Trojan horses that will release a corruptive payload. This characteristic will produce the same illegal data and privilege flow as the one demonstrated by Sheih and Gligor. This makes pattern-oriented modeling a very effective weapon in combating attempted break-in by potential intruders.
Internal predators

2) Masquerading or successful break-in (Masqueraders) – Pattern-oriented modeling requires that an intrusion produce indirect relations within their data and privilege flows to assist in intrusion detection. However, if an intruder manages to enter the system via non-technical means, such as by using a stolen or weak login and password set, then the intrusion would not produce any indirect flows at all. Similarly, if the intruder has successfully broken-into his intended target and assume his victim’s guise, any activity that involves extracting the files to an area outside the system, or even modifying or corrupting the data in any way will not be seen as intrusive. In masquerading cases, the intruder (subject) and the victim’s directory contents (object) lie within the same protection set (C). Security flow policies cannot be created to prohibit data and privilege flows from within a protection set because this involves only direct relationships between the user or intruder and his own personal files. Doing so would prohibit all basic user-system interactions and is not feasible. The IDS administrator can choose to introduce a security flow policy to prevent direct writes to the user’s sensitive files (like .login) in hopes to detect masqueraders. This, unfortunately, will limit user freedoms to an extent that might seem inconvenient. Furthermore, if the intruder has managed to break into his intended target’s account, it is rare that he would need to modify any sensitive files, so the chances of the new security flow policy to be effective would be very slim. Therefore, an IDS based on this method cannot see the intrusion taking place if the intruder is a masquerader. Yet, there is still a chance to detect a masquerader if the intruder needs to break into another user account that has better privileges, or if the intruder has not reached his
intended target. This would be similar to clandestine user behavior seen below.

3) Penetration by legitimate user (Clandestine users) – This intrusion detection method will be effective in detecting clandestine user activity. As described earlier on in this research report, clandestine users are legitimate users who attempt to gain greater access to the system by either breaking into another user account or exploiting a weakness in system processes. Just as with external predators, any attempt by a clandestine user to break-into another users account will produce indirect relations between the clandestine user and his victim’s protection set. Break-in attempts that originate from the four operational security problems mentioned earlier are far more common with clandestine users than they are with external predators. This is because there often exists ‘privilege overlaps’ between the normal users of the system, largely due to the setting of default privileges which is the third operational security problem, and the necessity to allow group or shared computing. However, this fact itself escalates the importance for IDS administrators to practice care when specifying the security flow policies to be inserted in the legal-flow matrix. While introducing more flow policies can help thwart clandestine user activity, it could potentially limit legal group or sharing activities. One exception to the effectiveness of pattern-oriented modeling in detecting clandestine user activity is if the intruder exploits a weakness in system processed that do not create indirect relations to the victim’s protection set. Although rare, this case would allow a clandestine user to be invisible to the IDS. If pattern-oriented modeling is the only method used to detect intruders in the system, then it is the
responsibility of the IDS administrator to fix these holes before they can be exploited.

4) *Inference by legitimate user (Clandestine users)* – Pattern-oriented modeling is one of very few intrusion detection methods that can positively detect inference attempts by clandestine users of a secure system. This is because the method itself is based on inference rules to detect indirect relations as described in the Section 3.4.1. The same inference rules used to detect illegal data and privilege data flows between distinct protection sets, can be directly employed in a database setting. Pattern-oriented modeling allows the IDS administrator to treat a database and its data repositories as a large protection set, which can be further divided into much smaller protection sets that segregate sensitive data areas from less sensitive ones. This way an intruder or clandestine user that has rudimentary access to the system is prohibited from inferring the contents of these sensitive data areas regardless of the complexity of query languages or stored procedures used to extract the data. This makes an IDS based on pattern-oriented modeling a perfect complement to systems that need security enforcement and intrusion detection in their secure databases.

5) *Leakage by legitimate user (Misfeasors)* – Perhaps the greatest drawback of pattern-oriented modeling as an intrusion detection method lies in its inability to detect misfeasors. Once again, misfeasors are users that leak out or smuggle the data contained within their own privileged files outside the system; the only difference between misfeasors and masqueraders is that the latter is a genuine intruder that originates from outside the organization while the former is a legitimate user. The weaknesses faced by the method is very similar to that which was exposed in the case of detecting
masqueraders. This intrusion detection method makes no distinction in the activities performed by the users of the system as long as their operations involve vertices (subjects and objects) from the same protection set. All relations derived from the interactions are legitimate and allowable data and privilege flows. There is no way for the IDS administrator to introduce a security flow policy to detect a misfeasor without seeing all legitimate users of a system as an intruder also.

**Malicious process**

6) *Trojan horse (Malware)* – While other intrusion detection methods, either behavior-based or knowledge-based, depend on the nature of a Trojan horses destructive payload to detect a possible intrusion, pattern-oriented modeling takes a different approach. A Trojan horse is activated when a user executes or opens the Trojan horse program, inadvertently releasing a form of destructive payload into the system. Regardless of the nature of the payload, this first creates a direct control-by relation from the victim to the Trojan horse. When the payload is released, an indirect write relation is created from the Trojan horse to whichever file the Trojan horse modifies or to a file that the Trojan horse creates (if any). Notwithstanding the potential compromise done by the destructive payload of the Trojan horse, this activity is actually identical to the intrusion demonstration given by Sheih and Gligor. What is not immediately obvious is the intruder subject that created the Trojan horse. An indirect relation will be created from the intruder to the modified or newly created files, and this can be positively identified as an intrusion when compared against the legal flow matrix. Therefore, regardless of the nature of a Trojan horse’s payload, this method can very effectively detect its existence and prevent its spread.
7) *Virus (Malware)* — Sheih and Gligor did not specifically mention whether pattern-oriented modeling is an effective means of combating computer viruses or intrusions that are the result of computer virus activity. However, a hypothesis can be deduced here in that if the virus activity or its method of spreading is anything like that of Trojan horse activity, then this intrusion detection method can detect a virus incursion, just as well as a Trojan horse-based intrusion. However, there lies a great difficulty in treating viruses like Trojan horses. In Trojan horse activity, it is almost always the case that the identity of the author or owner of the Trojan horse file or program would not be the victim. This means that there would exist interactions and indirect relations between two distinct protection sets. Unfortunately, when viruses infect an executable, and subsequently files used by that executable, the owner identity of the executable or file almost never changes. Once again the IDS will not see any processes spawned from, or files modified by, a virus-infected file as anything but a direct relation between vertices in the same protection set - unless, of course, the owner of the executable was originally from a different protection set. If the former were the case, then pattern-oriented modeling would be useless for the purpose of detecting viral intrusions. Therefore, this detection method can be said to have a limited capacity and mixed effectiveness of dealing with viruses, and is dependent on the properties of the virus-infected file.

8) *Denial-of-service (Malware)* — It is very doubtful that the method of intrusion detection by using pattern-oriented modeling would be effective or useful in any way to prevent a DoS attack. This is because resource usage, like memory, disk space, and computation cycles rarely produce any data or privilege flows as they are defined by Sheih and Gligor. A process initiated
by an intruder that hogs system resources is always the result of direct relations only. It would also be difficult to define DoS attacks in a protection graphs because it is unclear what vertices are involved and if any inter-protection set relations exist.

Apart from its inability to detect masqueraders and misfeasors, pattern-oriented modeling is quite an effective method of detecting intruders that take advantage of operating security problems and exploiting process weaknesses. It is also quite effective in dealing with Trojan horses, and also viruses to a limited degree. However, it would be best if this method were coupled with another intrusion detection method to address its shortcomings.
3.4.3 Capacity of Detection Method in Overcoming Error Types

The method of intrusion detection by using pattern-oriented models is a knowledge-based detection method, and therefore shares the same benefits and drawbacks as other knowledge-based methods when dealing with false positive and false negative error types. This is because the method also relies on the use of a repository to do comparisons when looking for intrusion patterns. However, while other methods rely on the existence of an encoded intrusion signature to match against user operations, pattern-oriented matching in actuality depends on the deviation from allowable operations, or in this case data and privilege flows, for a detection to be made. Although this characteristic is somewhat similar to that of behavior-based detection methods, it is not the same. While behavior-based methods depend on anomalies seen in users or processes, pattern-oriented modeling depends on violations of imposed security policies. These policies are the rule-base entries stored in the legal-flow matrix (F) that limits and restricts indirect relations.

Theoretically, the occurrence of false positive errors in an IDS based on this method can be controlled with very careful specification of the allowable data and privilege flow policies in the legal-flow matrix. Most intrusion techniques usually exploit weaknesses in system processes to cause an illegal privilege flow to gain greater access to the system or access to the intruder’s intended target. The flows generated as a result of these state transitions will exhibit indirect relations that can be pin-pointed by pattern-oriented modeling. If the legal-flow matrix is correctly configured, then the effectiveness of this method in overcoming false positive errors is quite high. Unfortunately, while the detection mechanism itself is quite robust, problems do lie in the security policies contained in the legal-flow matrix. Whenever an intrusion technique is uncovered, other knowledge-based systems can directly encode the intrusion signatures to the IDS for immediate detection. Pattern-oriented modeling
requires that the intrusion technique be analyzed by using protection graphs to pin-point the nature of the exact indirect relation that causes a system to be compromised. In the example presented by Sheih and Gligor [43], the intrusion is detected by the indirect control-by relation by a subject to another’s subjects sensitive files. While this could potentially assist in the detection of unknown intrusion methods that display the indirect relation, it could potentially limit actual legal use if a legitimate operation produces the same indirect relation. Consider a situation where the legal-flow matrix entry in the original example was expanded to include a restriction on normal user files to increase security, but without specifying exemptions to subjects within the same protection set. The added restriction would inadvertently consider all file and process sharing activities performed by user groups, like database updating activities, as a violation of the flow policy, and is therefore an intrusion. This is clearly a false positive error, so added care is needed in specifying the security flow policies in the legal flow matrix.

Pattern-oriented modeling is far more vulnerable to false negative error types, a problem common in other knowledge-based systems. As mentioned in the previous paragraph, the approach created by Sheih and Gligor can only detect intrusions that display an anticipated indirect relation prior to the occurrence of the intrusion. If the intrusion technique produces an unknown or overlooked indirect relation that is not restricted by any security-flow policy in the legal-flow matrix, the intrusion will pass by undetected. An intrusion could also not be detected if it does not produce any indirect relations or involve any perceptible data or privilege flows. This limitation severely effects the capacity of the detection method in overcoming false negative error types. Misconfiguration of the legal-flow matrix due to the incorrect specification of a flow policy might also let an intruder slip through the system without being detected. Therefore, stringent data and privilege flow rules must be derived from the known intrusion patterns. Once again, care must be exercised to guarantee that the rules aren’t
too rigid and would potentially create false positive errors as was mentioned previously, so the IDS administrator must strike a balance in controlling allowable privilege flows. Fortunately, intrusion detection by using pattern-oriented modeling has a unique way of letting IDS administrators handle yet unknown intrusion techniques. IDS administrators can hypothesize future intrusion techniques that can potentially occur in a system by mapping out their hypotheses on the protection graphs. Given a knowledge of publicly accessible file or directory areas that could be shared by any user type regardless of privilege, the protection sets that separates user types and their data, and a general suspicion on which process might potentially be the source of an intrusion, the IDS administrator can discover potential indirect relations that might originate from the intruder. New flow policies can then be generated from these rules and checked against currently allowable data and privilege flows to see if any incursions to existing user or group privileges can occur. If there are no problems, the new flow policy can be added into the legal-flow matrix in anticipation of a possible future intrusion attempt that exhibit this pattern. This added function would be greatly beneficial in reducing the overall false negative rate faced by this intrusion detection method.

Both false positive and false negative error types can further be reduced if the legal-flow matrix contains not only flow policies of allowable data and privilege flows, but also restrictive flow policies that explicitly prohibit privilege flows between subjects and objects of various protection sets. However, to do this, the IDS administrator must be careful to not include two conflicting rules that might cause system failures.
3.4.4 Performance and Impact of Detection Method in Practical Implementation

Pattern-oriented modeling was designed to allow an IDS to be deployed immediately into a computer system without any interruption to regular user activity. The security flow policies stored within the legal flow matrix can be added without having to halt production and therefore makes the IDS package very versatile. To assess the effectiveness of pattern-oriented modeling as a detection method, the five categories of effectiveness outlined by Debar et. al. [10] that was described earlier in this research report will be employed once again.

1) Accuracy – From the analysis given in the last section on the methods capacity in handling error types, this method can be said to have a fair level of accuracy in correctly labeling an ongoing attack as an intrusion. Unlike other knowledge-based methods, which tend to have a very high level of accuracy due to intrusion signature analysis, pattern-oriented modeling requires that the IDS administrator be extremely careful in specifying the allowable data and privilege flows in the legal-flow matrix so that the IDS does not see normal user activity as intrusive. Mistakes in the configuration are more likely to happen using this detection method because the complexity of the permissions structure of the users and groups must be taken into account. But if the configuration is done correctly, then false positive errors will not happen and the method will have a higher level of accuracy.

2) Performance – Although this detection method captures the system states from audit files to analyze the data and privilege flows currently taking place, it requires very little information from them. The most important items that would be required by the IDS would be subject information from the process ID, object information from the file or directory owner property,
timestamp of the process or any spawned executions, and the process primitive to determine the type of relation taking place between the subjects and objects. All this information can be derived from a single audit report, or a single customized report if the operating system does not provide it. The relations between the subjects and objects can be extracted very quickly from the logs, and any indirect relations that appear as a result of the user interactions can be immediately derived. Therefore, due to the speed of processing, pattern-oriented modeling has very high performance.

3) Completeness – The previous section on handling error types shows that pattern-oriented modeling faces much difficulty in reducing the number of false negative errors, or completely detecting all intrusion attempts. There are two reasons for this. First, it is because the method is a knowledge-based system, one that depends on a repository of recognized intrusion techniques to positively identify an intrusion. The absence of a rule, or security flow policy in this case, would prevent the IDS from recognizing that intrusion activity is taking place. The second reason for the lack of completeness in this method is that this method is unable to detect intrusions that do not produce any indirect relations or data and protection flows that do not cross protection sets. This shortcoming was proven in the method’s inability to detect masqueraders, misfeasors, and certain virus types that was detailed in Section 3.4.2. Each of these intruder or intrusion types do not produce any indirect relations. Therefore, even though the intrusion signature has been recognized, it cannot be defined or broken-down into a protection graph, rendering this detection method useless in detecting this intrusion. In other words, pattern-oriented modeling has a very low level of completeness.
4) **Fault Tolerance** – Apart from the existence of the legal-flow matrix as a rule-base storage mechanism, Sheih and Gligor [43] have not given much description about the architecture of an IDS based on this method in their research. The degree of fault tolerance in an IDS based on pattern-oriented modeling would most likely be dependent on how secure the legal-flow matrix or rule-base is against attacks. Assuming the rule-base is stored in a secure area of the system, its integrity is dependent on the prevention of intrusions into that area, or the prohibition of read-write accesses to the rule-base. This is where pattern-oriented modeling is one step ahead. A security flow policy that allows only direct read-write accesses by the administrator will send an intrusion alert any time anyone else tries to gain access to it either directly or indirectly by assuming the administrator’s identity. In addition, because the audit logs have to go to a form of pre-processing to extract the data and privilege flow information, the IDS would not be susceptible to “buffer overrun” type of attacks. However, as mentioned before, this detection method is unable to detect resource monopolization or DoS attacks, therefore the IDS can be expected to fail if these types of attacks are used.

5) **Timeliness** – Although the method of intrusion detection using pattern-oriented modeling has a very high level of performance due its speed in detecting intrusions, its timeliness in preventing any corruption as a result of the intrusion is very questionable. For an indirect relation from the intruder to the victim’s protection set to be identified, the intrusion itself must be completed. This can be seen in the intrusion demonstration given by Sheih and Gligor previously. In that example, if the indirect write from the rouge ls program did not take place to modify the contents of the victim’s .login
file, then an indirect write relation from the intruder to the .login file could not be established. This means that pattern-oriented modeling will not allow any preemptive measures to be taken by the IDS, even though the intrusion is positively identified. The only timely response the IDS can possibly take is to alert the IDS administrator or lock down both the intruder's and victim's account. To some degree, this is helpful in preventing further damage to the system.
3.5 Summary of IDS Detection Method Effectiveness

Sections 3.1 through 3.4 have detailed the four detection methods under analysis in this research. The overall effectiveness of these methods can be summarized in the tables provided below. To simplify the reading of these tables and comparison process, the detection methods will be referred to as follows:-

Method 1 - Behavior-based intrusion detection by observing the Sequences of System Calls
Method 2 - Behavior-based intrusion detection through File Integrity Checking
Method 3 - Knowledge-based intrusion detection using State Transition Analysis
Method 4 - Knowledge-based intrusion detection using Pattern-Oriented Modeling

Table 1: Effectiveness of detection methods against intruder types

<table>
<thead>
<tr>
<th>INTRUDER TYPE</th>
<th>Behavior-Based</th>
<th>Knowledge-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method 1</td>
<td>Method 2</td>
</tr>
<tr>
<td><strong>External Predators</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attempted break-in (Intruder)</td>
<td>Fair</td>
<td>High</td>
</tr>
<tr>
<td><strong>Internal Predators</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Masquerader</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>Clandestine User</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Misfeasor</td>
<td>Low</td>
<td>Fair</td>
</tr>
<tr>
<td><strong>Malicious Process</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virus</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Trojan Horse</td>
<td>High</td>
<td>Fair</td>
</tr>
<tr>
<td>Denial-of-Service (DoS)</td>
<td>Fair</td>
<td>Low</td>
</tr>
</tbody>
</table>

Table 1 above compares how each detection method deals with the various types of intruders. The following extrapolations can be made as a summary:-

1. All four detection methods are highly effective at detecting intrusions that cause clearly observable and auditable behavior characteristics, especially intrusions from clandestine users and computer viruses.
2. Observation of system calls is slightly less effective at detecting intrusions through attempted break-ins because an external attack may not produce discernable system calls.

3. File integrity checking may not be able to detect some attacks from Trojan horses, especially if the destructive payload does not involve any modification of file characteristics.

4. All other detection methods, apart from the two mentioned previously, are generally highly effective at detecting attempted break-ins and Trojan horses due to their scope of detection.

5. File integrity checking provides a fair chance of detecting misfeasors if the "accessed time" characteristics of files are controlled and monitored.

6. Observation of system calls has a possibility of being effective against DoS attacks if the attack results from a vulnerability in a particular privileged process.

7. All other detection methods, apart from the two mentioned previously, are generally highly ineffective at detecting misfeasors and DoS attacks due to the method of intrusion.

8. All four IDS detection methods analyzed in this report have a very low possibility of detecting masqueraders, especially if the intruder displays no behavioral anomalies or refrains from performing any further intrusive activity.
Table 2: Effectiveness of detection methods against classification errors

<table>
<thead>
<tr>
<th>CLASSIFICATION ERROR TYPE</th>
<th>Behavior-Based</th>
<th>Knowledge-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method 1</td>
<td>Method 2</td>
</tr>
<tr>
<td>False Positive Errors</td>
<td>Fair</td>
<td>Fair</td>
</tr>
<tr>
<td>False Negative Errors</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 2 shown above compares how the two classes of IDS detection methodology handles classification errors. The points below summarize the comparison:

1. Behavior-based detection methods are highly effective in dealing with false negative errors because they look for deviations from a known acceptable behavior.

2. Knowledge-based detection methods are highly effective in dealing with false negative errors because they only look for exact matches with known intrusive activity.

3. Behavior-based detection methods have a fair chance of preventing false positive errors if the training period to create the legitimate use database is long enough and allowances are made to accept changes in user operational behavior.

4. Knowledge-based detection methods are not effective in dealing with false negative errors especially if the intrusion technique has never been encountered or recorded previously.
Table 3: Effectiveness of detection methods in practice

<table>
<thead>
<tr>
<th></th>
<th>Behavior-Based</th>
<th>Knowledge-Based</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Method 1</td>
<td>Method 2</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Fair</td>
<td>Fair</td>
</tr>
<tr>
<td>Performance</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Completeness</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Fair</td>
<td>Fair</td>
</tr>
<tr>
<td>Timeliness</td>
<td>High</td>
<td>Fair</td>
</tr>
</tbody>
</table>

Table 3 above summarizes how effective the detection methods are in actual implementation. The following points summarize the analysis:

1. Knowledge-based detection methods generally have better accuracy than behavior-based methods because it depends on a database of known illegal activity for detection.

2. Behavior-based detection methods are more complete than knowledge-based methods because all deviation from a known legal behavior is alerted.

3. Behavior-based detection methods are less fault tolerant than knowledge-based detection methods because the detection source or database could possibly be trained to accept illegal activity.

4. Performance is slightly lower for intrusion detection through state transition analysis especially if the number of audit records to be processed increases.

5. Timeliness for all detection methods are mixed and is largely dependant on the stage in which detection takes place. Detection methods which track user activity and behavior in real-time have a higher potential of reacting to the intrusion than those detection methods that require a complete set of activity records before the detection process can even start.