Effective Techniques to Detect Anomalies in System Logs

A Thesis Presented
by

Sandhya Menon

to

The Graduate School
in partial fulfillment of the
Requirements
for the degree of

Master of Science
in
Computer Science

Stony Brook University
August 2006

Stony Brook University

The Graduate School

Sandhya Menon

We, the thesis committee for the above candidate for the

Master of Science degree,

hereby recommend acceptance of this thesis.

Dr. R. C. Sekar, Thesis Advisor,
Computer Science Department

Dr. C. R. Ramakrishnan, Chairman of Thesis Committee,
Computer Science Department

Dr. Rob Johnson, Committee Member,
Computer Science Department

This thesis is accepted by the Graduate School.

Dean of the Graduate School

Abstract of the Thesis

Effective Techniques to Detect Anomalies in System Logs

by

Sandhya Menon

Master of Science

in

Computer Science

Stony Brook University

2006

Computers and their networks are heavily used in every establishment today. The increasing growth of dependence on these systems necessitates efficient administration to ensure smooth functioning. Apart from the tasks of installing and updating software, system administrators need to keep a strict vigil on the activities in the systems. This is important in order to take timely action against failures, performance issues and security breaches.

There are several tools available for monitoring various aspects of a system. Most of these tools are designed for a particular service or look for specific patterns in log files. Their effectiveness depends a lot on the administrator's judgment and knowledge of what to monitor. This increases the possibility of overlooking previously unknown events in the system that need attention.

In this thesis, we explore a different approach for system monitoring that reduces this dependency and is capable of detecting *unknown* events. It involves detecting anomalies in log files. The large amount of information contained in these files can be used to characterize the expected behavior of a system. This property is suitable for the application of the anomaly detection model in order to extract significant events pertaining to performance issues, failure detection and security violations. It is also important that the false alarms are low.

The primary objective of this thesis is to investigate the effectiveness of applying the anomaly detection technique for log analysis, in order to identify potentially significant events for a system administrator. A secondary objective is to develop efficient algorithms to realize this technique.

Table of Contents

ACKNOWLEDGEMENTS

This thesis will not be complete without acknowledging the contributions of the following people during its various phases.

It has been a great learning experience to work under the guidance of my advisor, Dr R. C. Sekar. I thank him profusely for his valuable insights and immense patience.

I extend my gratitude to Dr. C. R. Ramakrishnan and Dr. Rob Johnson for taking the time and interest to be a part of the committee.

I am extremely thankful to Alok Tongaonkar, my mentor and friend, for always making the time to discuss and clarify concepts. His advice and support have been invaluable in completing this thesis.

I would like to mention a very special friend, Gaurav Poothia, who has been by my side in every time of need. I can not thank him enough.

It has been a great pleasure to be a part of the System Security Lab. I thank my friend Janani Natarajan for her constant support. I thank Chetan Maiya, Karthik S, Mohammed Mehkri, Prateek Saxena, Zhenkai Liang, Weiqin Sun and Varun Katta for their encouragement and help in various forms, that helped in the completion of this thesis. I especially thank Wei Xu for promptly supplying the test data whenever they were requested.

I take this opportunity to also thank my dear friends Vijay Arvind B, Divya Gupta and Karthik Tamilmani for always being there.

I finally want to thank the ones who are a very important part of my life, my family. My mother for her prayers and encouragement, my father for his words of wisdom, my sister for providing the lighter moments, and my lovely grandparents for everything.

Chapter 1

Introduction

An important goal of system administration is to ensure healthy systems and networks. Considering that a reliable and problem-free environment is next to impossible, this goal entails a lot of work. Apart from taking backups and installing software, an integral part of a system administrator's duty is troubleshooting unexpected situations and safeguarding against security breaches. The scope of this task is not small and hence a system administrator must depend on a whole suite of support tools and techniques to carry it out smoothly. This dependence has motivated the development of various tools that provide utilities to monitor various components of a system or network to ensure smooth functioning.

Log analysis

Logs are an important set of resources that have the potential to provide aid to system administrators. [14] defines *logging* as the process of recording events or statistics to provide information about system use and performance. Logging is an important activity carried out by most applications and systems. On Unix, services can avail of the `syslog` utility to record their audit data. The details contained in these log files can provide valuable insight into operations of the system and network. The process of analyzing logs to unearth security violations, failures or performance lapses is often referred to as *auditing*.

Log analysis can produce valuable information for system administrators in a number of ways. Lying latent amidst the verbiage in log files, are significant events that deserve attention. A sequence of events that threatens normal functioning of an infrastructure can be determined from analysis of log files. On the other hand, a subset of information extracted from log files can eliminate a particular cause and thus help in further analysis. Another benefit of log analysis is to help establish usage pattern. This would help in reviewing policy and access violations by users within the network.

Despite its obvious benefits, this is an often ignored resource. This trend is not too difficult to fathom considering that log files can run into thousands of lines depending on their source. Most of the entries are innocuous, routine information that are uninteresting. For example, when the application *snort* starts up, it logs a number of copyright related info. A manual audit of a large volume of information can prove to be an extremely cumbersome and error-prone effort. A cursory glance could bring to light some obvious failures and rare events. However, there is much information that could be inferred by correlation of events based on temporal properties or sequence of occurrence. Thus, the task of analysis is a natural application of automation. There are several tools and techniques available that help in achieving varying degrees of automation.

The information contained in log files reflect the conformance of processes and user actions to certain statistical patterns. This trait makes log files suitable candidates for the application of anomaly detection model, that is commonly used for intrusion detection. An anomaly can be defined as a deviation from the normal behavior thus representing something unusual, abnormal or peculiar. The logs of a system that is believed to function normally and reliably could be utilized in order to build a model of the expected behavior of the system. Other logs, that need to be analyzed for anomalies, are compared with this model to detect deviations. This technique could be adapted for carrying out log analysis to unearth anomalies that characterize security breaches, service or system failures, or performance issues.

An overview of the anomaly detection model that is typically applied to intrusion detection is provided in the next section for a better understanding of this model. Section 1.1 will provide a more relevant discussion on how this technique can be applied for extracting significant information from log files.

Anomaly Detection

This technique is modeled on the assumption that usage patterns and process behavior follow a statistical pattern in an attack free environment. A deviation from this pattern signals an intrusion. In other words, any *unexpected* behavior is assumed to be signs of intrusion. [14] defines anomaly detection as a technique that builds a statistical characterization of a system and marks any event that is statistically unusual as *bad*.

There are a few classifications of anomaly detectors based on the metric used to characterize the normal behavior of a system. A violation of this metric will signal an anomaly. The threshold-based model sets a minimum and maximum threshold on the expected frequency of the occurrence of events. The second model establishes an expected interval of statistical moments (mean and standard deviation or higher moments) as the metric. A third model is based on Markov model where a set of probabilities of transitions of a system from one state to another is set as the

metric. These transitions correspond to occurrences of events. Any event that causes a transition having a low probability is flagged to be anomalous.

The biggest advantage of this technique is that it is well suited to discover *unknown* attacks or discover *unexpected* situations. Whereas a model like *misuse* detection, that compares events with a model of *known* signatures of attacks to detect attacks, fail in this respect. However, the anomaly detection model results in the production of a large number of false alarms and requires constant updation of the *expected model* that it uses as reference for detection. The other problem with this technique is that its effectiveness is largely dependant on the training data set. If this data set contains characteristics that are unusual for the system, then these will go unnoticed during detection.

1.1 Objective

As mentioned earlier, log files contain a lot of information that can be utilized to characterize the expected behavior of a system. This property is suitable for the application of the anomaly detection model in order to extract significant events pertaining to performance issues, failure detection and security violations. The objective of this project is two-fold.

- To investigate the effectiveness of applying anomaly detection technique for log analysis in order to identify potentially significant events to a system administrator.
- To develop efficient algorithms to realize the above described technique.

1.2 Organization of Thesis

This section provides an overview of the contents of subsequent chapters.

Related Work

An extensive study of tools and techniques applied for monitoring the system for events threatening its smooth functioning is covered in chapter 2.

Background

Chapter 3 describes the specification language used to model the anomaly detector. It describes the syntax and semantics of the construct that allows specification of the properties that are of interest.

Efficient Algorithms for Realizing Detection Engines

Chapter 4 investigates the computational steps involved in realizing the anomaly detector. It covers a detailed discussion regarding the expensive computations involved and suitable solutions and optimizations to achieve better efficiency.

Anomaly Detection and Learning

Chapter 5 provides details regarding the process of learning characteristics of the system and anomaly detection. It gives the details regarding the calculation of threshold that is used in determining anomalies.

Agents

Chapter 6 describes the agents used to monitor and process log files in order to extract events of interest. It provides an overview of the specification language used to define the agents.

Results

Chapter 7 explains the experimental setup, and discusses the experiments and their results.

Summary

The thesis is concluded in chapter 8. A discussion on the assessment of the effectiveness and weaknesses of this technique is provided. Some directions for future work is also provided.

Chapter 2

Related Work

The importance of log analysis is well established. A manual effort at it is a cumbersome task and has motivated the application of various techniques to automate it. There are several tools available for log analysis, based on these techniques. This section provides a description of some techniques and tools related to this domain.

2.1 Pattern-matching based detectors

Many intrusion detection techniques rely on looking out for sequence of bytes that constitutes a pattern in log files. Such pattern-matching based detectors mainly help in eliminating known routine log entries. Hence they are successful in reducing the number of entries to be scrutinized. Many of them help in organizing the log entries, facilitating easier examination. However, a reduction from say a 100,000 lines to 3000 lines still doesn't get rid of the struggles of a manual inspection.

To their credit, some of the tools have sophisticated capabilities to group and correlate events, or detect errors based on temporal relationships between events. However, as we visit the details of this category of tools, it becomes evident that these tools are effective in detecting mainly *known attacks*. There is a huge dependence on expert judgment of a system administrator to specify patterns that need attention. Most of the times, it will be based on what attacks or failures have been observed before. Unfortunately, it is just as important to be prepared for the unknown. These tools would be found lacking in this respect.

The tools described below can be categorized as offline analysis tools and online monitoring tools.

2.1.1 Offline analysis solutions

Some of the tools contribute to log analysis by summarizing the logs over a period of time into reports that can be examined by the system administrators at a later hour. They are typically invoked periodically, like once in a day, to analyze the logs. These tools are beneficial in that they can isolate events for further analysis by real-time reporting tools. The reports also provide statistical information that helps system administrators take necessary actions. However, there are situations that require instantaneous reactions which these tools are incapable of providing. For example, if an NFS server is down, a system administrator needs to be alerted immediately so that the computing environment is least inconvenienced. Realizing a day later, that the server was down, will not serve the purpose.

Logwatch

Logwatch [5] is a customizable tool that parses through the logs specified by users to create a report based on the criteria, again specified by users. These specifications are provided as command line options. It is a set of Perl scripts and filters and can be easily setup and configured. The details provided in the reports are configurable by users. It can be used to analyze log output of many popular programs. It can be easily configured to interpret outputs of programs outside this list.

SLAPS-2

SLAPS-2 [6] is a collection of Perl programs used to filter system logs on a centralized log server. This tool produces a series of analysis reports from the log files that can be e-mailed to the specified recipients. SLAPS-2 can also manage rotation of the log files used during analysis.

2.1.2 Online monitoring solutions

The real-time analysis is better referred to as online *monitoring* as these tools are continuously running and monitoring one or more log files. Their obvious advantage over their above described cousins is their ability to generate real-time event driven triggers. Many of the tools described below have advanced features for detection. However, their abilities are limited when it comes to uncovering unexpected situations.

Swatch and 2Swatch

Swatch [4], or *simple watchdog*, was the first well known program available for monitoring log files. It acts as a filter of unwanted data and takes one or more user

specified actions. When it encounters a line matching the pattern specified by users, it can either print it out, or run external programs to notify administrators. It allows for a range of actions that can be specified. It provides support for ignoring duplicate events. However, since it examines only one event at a time, it can not correlate events in time. It is a set of Perl scripts and configuration files. 2Swatch [3] is a variation which achieves some optimizations over swatch. It can correlate records and thus reduces the number of alerts mailed out.

Logsurfer and Logsurfer+

Logsurfer [11] is a more powerful tool that can dynamically change its rules based on events or time. This increases its ability to allow for correlating events based on context. It allows for a lot of options that makes it flexible. Users can specify exceptions, set timeouts for rules, specify patterns that can be ignored, and other useful actions like mailing results to specified targets. It is a single small C program and can be easily installed. Logsurfer+ [10] is an extension of logsurfer with additional features. It can alert when messages stop coming in, and also specify the minimum number of messages required to trigger an alert. The last mentioned feature equips the tool with the ability to detect hyper activity of some events indicating anomaly. However, this would require the judgment and experience of the system administrator to know what is an appropriate minimum number for a particular event.

Simple Event Correlator

SEC [9] is an open source, platform independent tool used for event correlation. It can receive inputs from regular files, named pipes and standard input. It has a list of rules, and the input lines are matched against these rules. Each rule consists of an event matching condition, an actions list and a boolean value that controls whether the rule can be applied at a particular moment. The event matching conditions are expressed as regular expressions and Perl routines. SEC can take various actions like creating contexts, invoking external programs and resetting active correlations, to name a few. Each rule can specify a context name, thus allowing events to correlate based on context.

SEC has static rules unlike logsurfer. However, it provides higher levels of correlation operations like explicit pair matching and counting operations. The *SingleWithThreshold* rule allows counting of event correlation. For example, number of occurrences of event A can be counted in a given window time and the count is compared to a threshold value specified in the rule. If the count exceeds the specified threshold, an action will be taken. An illustration of this would be to count the pattern *login failure* in a window of 60s and threshold of 3. If this threshold is ex-

ceeded, then the action specified is to invoke `notify.sh`, a user defined script. This feature would be very helpful to detect well known failures as illustrated. However, it will not be possible for a system administrator to guess suitable thresholds for many lesser known events.

Open Source Host-Based Intrusion Detection System

OSSEC HIDS [8] is a detection system that provides services like log analysis, integrity checking, rootkit detection, time-based alerting and active response. Its architecture consists of agents forwarding events to a server. The rules are specified in an XML file and are flexible enough to provide a lot of services. The most important ones being the ability to correlate events, and provision for setting of frequency for a specific rule before alerting. It recognizes various input formats like syslog, apache, snort and a few others. Its strength lies in the fact that it provides solutions for more than just log analysis. However, it also faces the criticism that, as far as log analysis goes, it is limited in its ability to capture *unknown events*.

2.2 Miscellaneous Tools

The above mentioned tools are primarily employed to detect security breaches. However, there are a suite of tools that use various techniques to provide system monitoring services. These services include performance monitoring and failure detection. This section provides details for some of the tools that are available for this purpose.

Spong

Spong [7] is a system and network managing package that communicates using simple TCP based messages. Some of its attractive features are as follows:

- Client based monitoring that provide data about CPU usage, disk usage and log monitoring.
- Network based monitoring which involves services such as *ping*, *smtp*, *http*, *dns* and a few more.
- Use of rules to specify and customize messages to report problems.
- Display of results via web-based or text-based interface.

It consists of four main modules – *spong-server*, *spong-network*, *spong-client* and *spong-message*. These modules are written in Perl.

- The server module, *spong-server*, is the core of the Spong package. It listens at port 1998 for status reports from client and network monitoring modules that will be described shortly. The updates contain necessary information like name of service and timestamp. In addition, it also contains a color to indicate the status of the service. The color green indicates that the service is working as normal. The color yellow indicates a warning, and the color red indicates an alert. The server updates its database with the status message and passes the message to the message module which displays the results graphically or textually, as specified in a configuration file.

- The client module, *spong-client*, is deployed on each host that requires monitoring of local system attributes. It issues relevant system commands to run configured checks, and parses the output in order to determine the status of the service. It then conveys the status update to the server. The client can be made to behave as specified by providing the correct information in its configuration file. For example, after it performs one round of checks, it sleeps for a time interval specified in the configuration file. When it wakes up, it repeats the checks again. In order to prevent the clients from overloading the server, the implementation is such that a random amount of time is added or subtracted from the time interval specified. However, this does not exceed 10% of the specified time interval.

- The network module, *spong-network*, checks network service availability and network connectivity to various hosts specified in the configuration file. The network services checked are configured separately for each host. It also sleeps for a time interval specified in the configuration file between every round of checks.

  Consider this example to understand the functioning of the network modules. If we want to ensure that the DNS server is functioning properly, the network module responsible for the DNS check uses the Perl module `Net::DNS` to connect to a DNS server and ask it to resolve its own name. If it does so successfully and returns the expected status message, then the network module will report that the DNS server is up and running.

This package is suitable for a sanity check of the hosts and network. It achieves a limited extent of log analysis by scanning the specified log files for a particular pattern expressed as a Perl regular expression in the configuration file. The status to be reported is also specified in the configuration file. It lacks in flexibility as every service that needs to be checked requires a module implemented for it. Its functionality is also limited by the services provided by Perl as all the checks are carried out using Perl modules like Net::DNS.

Chapter 3

Background

Any anomaly detector can effectively detect unusual behavior only after it is trained to know the properties that depict usual behavior. For this purpose, feature selection plays an important role. Typically, experts, by virtue of their knowledge and judgement, might end up selecting features influenced by known attacks. While they might select some useful features, there is no guarantee that this would serve well to detect unknown attacks. A specification-based approach to anomaly detection proposed in [1] provides a high level of automation in the process of feature selection. In this approach, this is achieved by mapping statistical properties of system behavior to statistical properties of transitions of the state machines. Deviation from the previously observed properties of the transitions can be interpreted as an anomaly in the system behavior.

The anomaly detector is modelled as an *Extended Finite State Automata (EFSA)*. EFSA is specified using SMSL, a language that models systems and services as a state machine. EFSA and SMSL will be dealt with in greater detail in section 3.1.

The support to specify the statistical properties of interest is provided by the *on statement* and is described in detail in section 3.2.

3.1 SMSL

EFSA is similar to a finite-state automaton, with an additional set of state variables that can be used to store values. It consists of a set of states, a set of input events, a transition function and a set of state variables that can be used to remember values as transitions are made from one state to another. An EFSA can be formally expressed as a septuple $(\mathbf{\Sigma}, \mathbf{Q}, \mathbf{s}, \mathbf{f}, \mathbf{V}, \mathbf{D}, \delta)$ where:

- $\mathbf{\Sigma}$ is the alphabet of the EFSA. The elements of $\mathbf{\Sigma}$ are events characterized by an event name and event arguments.

- $\mathbf{Q}$ is a finite set of states of the EFSA.
- $\mathbf{s} \in \mathbf{Q}$ is the start state of the EFSA.
- $\mathbf{f} \in \mathbf{Q}$ is the final state. It has no outward transitions.
- $\mathbf{V}$ is a finite tuple $\mathbf{v_1}, \cdots, \mathbf{v_n}$ of state variables.
- $\mathbf{D}$ is a finite tuple $\mathbf{D_1}, \cdots, \mathbf{D_n}$, where $\mathbf{D_i}$ denotes the values for the variable $\mathbf{v_i}$.
- $\delta : \mathbf{Q} \times \mathbf{D} \times \mathbf{\Sigma} \rightarrow (\mathbf{Q}, \mathbf{D})$ is a transition relation.

SMSL is a language comprising of a set of declarations and rules [2]. It is used to specify EFSA as follows:

- The events that form the alphabet of the EFSA are declared as part of an interface declaration. It has the following syntax:

  `event` *eventName*`(`*parameterDecls*`)`

  The *parameterDecls* is a list of the declarations specifying the types of the parameters in the event.

- The set of states are defined as `states` $\{s_1$`, ...,` $s_n\}$.
- The start state is specified as `startstate` *s*.
- The final state is specified as `finalstate` *f*.
- The state variables are declared using the common variable declaration syntax of *type variableName*, as seen in typical programming languages.
- The transitions are expressed as rules and take the following form:

  pattern –>*action*

  The *pattern* is a *regular expression over events (REE)* that is a sequence of events with arguments. It has the syntax *eventName*$(v_1, \ldots, v_n)|$*cond*. The condition *cond* evaluates to a boolean value. It can consist of relational or arithmetic operators.

  The *action* refers to the actions that need to be taken when *pattern* is matched. This part is where assignments to the variable *state* and assignments to the state variables are carried out. Even invocation of external functions can be specified in *action*. Thus, the transition of the state machine from one state to another can be specified.

The `map` construct is used to enable efficient look up of state instances that need to make a transition on the arrival of an event. At run time, several state transition instances will come into existence. For every incoming event, this would cause searching through all those instances to find the ones that are qualified to make the transition. The state machine instance can easily be determined by the event parameters. This is done by using the following syntax:

`map` *eventName*`(`*arguments*`)` `when` *cond*

The component *cond* is a conjunction of equality tests where *arguments* form the left hand side value of the expression and a state variable forms the right hand side. In addition, it is mandatory that the number of state variables used in these conditions should be the same. This constraint facilitates in implementing the look-up of the relevant state machine instances as a hashtable look-up.

There is also a special kind of transition called *timeout* transition. This transition is associated with a special event called *timeout*. This feature is useful in capturing temporal properties. When a state machine takes a *timeout* transition, it indicates that it has timed out. The language allows us to specify this feature as follows:

`timeout` t `in` $\{s_1, \ldots, s_n\}$

This statement specifies that a state machine existing in any of the states represented by s_1 through s_n must time out after t seconds.

3.2 The On Statement

It would be useful to understand the *properties* one might identify for the system. While analyzing firewall logs, one might be interested in the frequency of requests for a particular service. This can be easily captured by studying how often a transition representing this event is taken. An example of a different category would be the interest to monitor the scripts being accessed on a webserver. This can be mapped to the values of the state variable representing the url accessed. Thus we can summarize the categories of properties as follows:

- Frequency with which a particular transition is taken.
- Values of a state variable on a transition.

The frequency of the occurrence of an event can be captured by counting the number of times a transition is taken over a period of time. While some attacks or failures are visible in short time intervals, there are others that come to light only over longer periods of time. For this reason, the mechanism allows for specifying a range of time intervals.

In most cases, just measuring the frequency of all events does not provide much information. It would be of greater interest to gain more specific information. For example, it would be very useful to focus on events on a per-host basis. Consider a second example of observing how often authentication failures occur on a per-service basis. Thus, the mechanism is aimed at providing flexibility to specify a subset of transitions and state variables to be monitored.

The syntax and semantics of the *on statement* are described in detail in the following sections.

3.2.1 Syntax

The *on statement* is a specification language used to express the statistical properties to be learnt and later used for detection by the anomaly detector. It is a simple and flexible interface which captures all the design features expressed above.

The syntax of the *on* statement is as follows:

on (**all**|**any**|([**all of**|**any of**] **{***trans* {**,** *trans*}**}**))[**frequency**|**value**]

[**when** *cond*]

{**wrt** var{**,** var} **size**[*size*]}*

timescales (*ts* {**,** *ts*})

falseAlarmRate *rate*

The next section deals with a detailed explanation of the semantics of the *on statement*.

3.2.2 Semantics

The semantics of the five clauses defined in the previous section are explained below.

- **on** clause specifies the set of transitions which will be monitored in order to collect statistical information. The statistics can be a collective measure for all the listed transitions or it can be collected per transition by using keywords *all* and *any* respectively. The keywords *frequency* or *value* is specified to indicate that frequency or value distributions need to be maintained, respectively.

- **when** clause is useful to filter the transitions further, based on the result of condition *cond* over state variables. As the name suggests, it specifies when to consider the listed transition(s).

- **wrt** clause contains a list of state variables. The statistical information collected is a distribution of the frequency of the occurrence of events over these state variables. It could also be a distribution of the values for the state variables. The state variables in the list form a key, and statistics is collected for each unique key.

  It is possible to have multiple **wrt** clauses. The usefulness of this feature will become clearer in later chapters.

  The *size* variable indicates the maximum number of such unique keys for which statistics will be collected at a time. This provides a mechanism to purge older entries. This system is designed to purge entries based on least frequently updated entries.

- **timescales** clause provides the timeperiod over which the statistics are collected. Smaller timescales enable faster fault detection. However, slow anomalous activities can be detected over longer time frames. Hence, multiple timeperiods can be specified in order to not miss either fast or slow anomalous activities. Each timescale will have its corresponding frequency distribution.

- **falseAlarmRate** clause provides the maximum desired false alarm rate. It should be specified as a number between 0 and 1. This, alongwith the distribution learnt during training, will be used to calculate an appropriate threshold for the counter. An alarm will be raised, during detection, when the counter exceeds this threshold.

  There is an obvious trade-off when choosing a false alarm rate. A low rate will mean less alarms and therefore less work. However, some intrusive behavior could be missed. While a high rate will mean more alarms, but a greater likelihood of catching suspicious activity.

Chapter 4

Efficient Algorithms for Realizing Detection Engines

The *on statement* described in the previous chapter provides a mechanism to capture either the frequency with which a set of transitions are made, or the most frequently occurring values of state variables.

If a *wrt* clause is specified, then it is required to maintain a counter for each unique value of the variables to build a distribution. Its absence results in maintaining only a single distribution for the set of transitions specified. The occurrence of either a transition or a unique value of state variables will result in incrementing the corresponding counter.

The frequency of a transition is maintained over a range of time periods as specified in the *timescales* clause. The computation and maintenance of distributions can prove to be expensive. Section 4.1 covers a detailed discussion on the issues seen and suitable solutions.

Another area of concern is the maintenance of a collection explained shortly. For each *wrt* clause, a collection of the unique values for the state variables specified, along with their corresponding counts needs to be maintained. There is also the additional restriction that this collection can not contain more than a maximum number of elements as specified by the *size* parameter. These collections will be accessed frequently. Hence, this collection must be represented by a data structure that allows an easy access mechanism to its entries. The solution to this is discussed in section 4.2.

4.1 Computation of Frequency Distribution

As mentioned earlier, the specification allows frequency distribution to be maintained for several timescales. This can be implemented by maintaining counters for

each time interval. The counter is incremented with the occurrence of every event that contributes to the distribution. The distribution can be represented as a histogram. As frequencies are scalar quantities, they can be conveniently expressed in terms of the bins of the histogram.

Typically in detection, the statistics are computed again to compare with the statistics developed during the learning phase. However, this can be an expensive operation in terms of time and space. However, this can be avoided. During initialization, a threshold is calculated based on the statistics collected during learning and user-defined false alarm rate. Details regarding the calculation of threshold is discussed in chapter 5. From there on, only a count needs to be maintained and a check to see if it exceeds the threshold. The need to maintain a histogram is eliminated, thus saving space. This is a significant gain in terms of storage and time complexity as the system would typically be run in the detection mode more often than in learning mode.

An issue that must be looked into, is the presence of quantization errors owing to maintenance of counts for discrete intervals of time. A solution to this is described below.

4.1.1 Sliding counters

Sliding counters can reduce effect of quantization error. These counters are used to count such that the counts within the past t-seconds contribute more towards the total count than counts that are older. Let us consider an example where we keep track of counts in the past 10 seconds. Hence at $t = 15$, counts sampled from $t = 6$ through $t = 15$ will contribute towards the total count in this window of 10s. Then again, at $t = 20$, counts measured from $t = 11$ through $t = 20$ will be included in the total count. At this point, the counts measured at $t = 6$ through $t = 10$ are deducted as they are considered as expired time units.

This solution, however, poses a problem. There arises the need to store counts of every time unit that contributes to the total count for a particular time interval t. This is a problem especially when t is large, which is very likely. To overcome this issue, an approximation can be achieved using sub-windows. The counter can be viewed as a window of duration t which represents the time interval in *timescales* clause. This window is divided into n sub-windows of duration d each, such that $d * n = t$. Each sub-window accumulates counts for a duration of d. The total of all these counts will be the count for t. For every lapse of duration d, the window slides ahead by a duration of one sub-window. The count in the expired sub-window is eliminated from the total count. A circular array is used to implement the sub-windows. The value of n chosen for our implementation is 4. The counting algorithm can be stated as follows.

```
1.  Let t be the duration of the window
2.  Let n be the number of sub-windows
3.  Let t_beg denote the beginning of the window
4.  Let t_cur denote the current time
5.  Let totalCount be the count for the time t
6.  Let count[i] hold the counts for sub-window[i]
7.  When an event arrives at t_cur,
8.  for every t/n that has occurred in (t_cur − t)
9.  begin
10.   Slide t_beg past the expired sub-window i
11.   Deduct count[i] from totalCount
12.end
```

From the description above, we see that steps 8 through 12 is a loop and will incur some cost especially if t_{cur} is a much larger value than t. Let us calculate the upper bound on this operation. If t_{cur} is a larger value than t, then all the counts corresponding to the expired sub-windows must be deducted from $totalCount$. This would require checking each of the expired sub-window's count. However, these sub-windows are implemented as a circular array and hence this operation will have an upper bound equal to the number of sub-windows n. Hence, at the worst case, if more than n subwindows have expired, then all the counts maintained for the n sub-windows will not contribute to the $totalCount$ for t_{cur}. Hence there will be no need to check more than n sub-windows. Thus the complexity is O(n), where n is the number of sub-windows. In our implementation, n is a constant value and does not depend on the number of events.

4.1.2 Optimization

While computing the frequency distribution, a counter is maintained for each timescale. In this case, for every event, the sliding counter has to be incremented for every timescale specified. An optimization can be introduced wherein the counting operation is limited to only one timescale. This can be achieved if the higher timescales are implemented over the smallest timescale specified in the *timescales* clause. Before we visit the details of this optimization, we must note that this would impose the restriction on the user to specify timeperiods of the form t, k_1t, k_2k_1t, $k_1...k_nt$. This compromise will not result in reducing the flexibility of the *timescales* clause. The optimized algorithm is described below:

```
1. function inc() begin
2.   Let t[n] represent n timescales
3.   Let c[i] represent count for t[i]
4.    for i = 0 to n-1
5.    begin
6.      if t[i] elapses then
7.      begin
8.        c[i+1] += c[i];
9.        if learning mode then
10.         put c[i] in appropriate histogram bin;
11.       reset c[i];
12.     end
13.   end
14.   increment c[0];
15. end
```

The timer used for the operations is based on the timestamp of the events. Interval between occurrences of events serve as indication of the time elapsed.

From the above algorithm, we see that for every event that occurs, almost every instruction has a constant cost with one exception. In step 10, updation of the histogram will require a traversal to find the appropriate bin that $c[i]$ belongs to. The bins of the histogram are arranged in geometric progression.

Let the total number of events be N. Now consider the case when the total duration of the N events is equal to T, the only timescale specified. Hence the largest count possible will be N for T. Since the bins are in geometric progression, the traversal will take $logN$ iteration and is the upper bound. Next consider the case where we introduce more timescales that are smaller than T for the same set of N events. For any timescale value $t_i \leq T$, the traversals can not exceed $logN$. Hence, for N events, with N operations, the complexity will be $0(N)$.

In the case of detection, a significant gain is achieved by not maintaining the histogram. The use of a simple sliding counter to keep a count of the occurrence achieves better space and time efficiency.

4.2 Maintenance of Table of Most Frequently Used Values

In this section, we will discuss the appropriate choice of data structure for the collection discussed above. The collection is represented as a table designed to store the key-value pairs for the set of variables in a *wrt* clause. It is similar to a hashtable, where the distinct values of the state variables act as a key. This representation is

best suited as the complexity of a lookup is $O(1)$, and thus making the frequent accesses efficient. It can hold key-value pairs of arbitrary types. However, the maximum permitted entries to this table is limited to the argument *size* provided in the *wrt* clause. This limitation is imposed so that *stale* values are purged. Thus, the table contains the most frequently accessed entries.

The purging action is invoked when a new entry has to be inserted into a table that has reached its maximum permissible limit. At this point, k oldest elements are removed from the table. This scheme has its advantage over simply deleting the oldest entries in that it will result in requiring purging less frequently. The purging action is invoked atmost once in k times. The value k represents a fraction of the maximum size, N where $k = \lfloor f * N \rfloor$.

It is possible to find the k oldest entries by a procedure having expected complexity of $O(N)$, i.e. linear. The algorithm is the partial quicksort [12]. Quicksort creates two partitions around a pivot such that one partition contains elements lesser than the pivot and the other contains elements greater. For this application, we only require the partition containing the *kth* oldest entry.

Chapter 5

Anomaly Detection and Learning

The anomaly detector, or detection engine (DE) can detect an anomaly only after it is trained to know the properties that depict usual behavior. Hence the DE is first executed in a learning mode wherein it collects statistics based on events that are observed normally in the system. Once it is sufficiently trained, i.e. the statistics are believed to represent usual behavior of the system, it is ready to detect anomalies. Thereafter, when the trained detection engine is run in the detection mode, it checks to see if the properties of the currently observed events deviate from the learnt statistics. In such an event, an alarm is raised informing the system administrator of a possible anomaly.

In the next sections, descriptions of the activities carried out during the training and detection phase are given.

5.1 Training phase

The statistical properties of the system are mapped to the statistical properties of transitions. The properties learnt would typically be the frequency with which a transiton or a set of transitions are taken, or values of state variables that occur. See chapter 3 for a detailed discussion. These statistical properties are learnt and represented as frequency and value distributions respectively. These distributions are represented differently based on their nature. The use of histograms is most appropriate for representing the former, as frequency is a scalar value. Scalar values can be conveniently represented by the histogram bins. On the other hand, the latter is better represented as a set of unique values as they tend to be *categorical*. For example, the services running on a system.

For a new, untrained detection engine, the initialization process involves instantiation of top level data structures for every *on statement* in the specification. An alternate option is to start the detection engine in a mode that allows it to continue its training. This option is very useful as anomaly detectors need to be constantly trained regarding valid changes in behavior. In this case, the data structures should be brought to a state as in the end of the previous training phase. This data is made available from a file that contains the saved results of previously learnt statistics.

During the training phase, every time a transition of interest occurs, the related statistical counters need to be incremented. *Statistical counters* refer to the entities maintaining a count of the occurrence of the transition or the value of a particular state variable in that event (as specified by the *on statement*.

At the end of the learning phase, it is important for the learnt information to be stored persistently. This data will be required either to continue learning or to detect anomalies. The results are stored in two different files. One file contains the details of the distribution for each key contained in the tables. The details of the distribution also include the values stored in the histogram bins in case of frequency distribution. This file is then used to provide the details to restore values when the system is again started in the continued learning mode. The second file contains only as much information as is required for detection. For detection, only information pertaining to the values of the state variables seen and some extra information required for calculating the threshold for every timescale is required.

5.2 Detection phase

Detection is typically carried out by computing the statistics specified for the current stream of events, and comparing to see if the statistics deviate from what was computed during the training phase. However, it would impact performance to maintain the distributions and carry out the comparisons. Instead, we describe an alternate solution that is simple, yet effective.

During detection, maintaining a sliding counter to keep track of the number of occurences of an event for each timescale T will suffice. There is no need to store the distribution. This way, additional storage can be avoided. From the statistics learnt previously, we can compute a threshold C_t. During the detection, when the count C pertaining to a particular event or value, exceeds C_t, then an alarm should be raised. The false alarm rate, specified in the *on statement* plays a role in the calculation of thresholds. A detailed discussion on this is presented in the next chapter.

This simplified solution helps in reducing the time and space complexities seen during the training phase.

5.2.1 Threshold Calculation

An integral part of the system is to calculate a threshold from the observed statistics. This threshold serves as the representative of expected behavior of the system. The *falseAlarmRate* clause specifies a value that is used to determine the threshold. The following discussion will throw light on how the threshold is determined.

Let X^N represent a valid value in the distribution and T represent the threshold. As described earlier, an alarm is generated when a value $X > T$. However, the alarm is said to be a false alarm (also known as false positive), if X is a valid value. If FA denotes the false alarm rate, this situation can be mathematically defined as

$$P(X^N > T) = FA \tag{5.1}$$

Chebyshev's inequality and its one-tailed variant [16], quantitatively describes that all the values in a probability distribution are close to the mean value. This theorem is proven to hold for any kind of distribution. The following is the mathematical statement of the one-tailed variant of Chebyshev's inequality.

$$P(X \geq \mu + k\sigma) \leq \frac{1}{1+k_2}), k > 0 \tag{5.2}$$

μ and σ represent the mean and standard deviation respectively Applying (5.2) to (5.1), it can be stated that

$$FA \leq \frac{1}{1+k_2}), where T = \mu + k\sigma$$

Using the above information, we can deduce that

$$T \leq \mu + \sigma \sqrt[2]{\frac{1}{FA} - 1}$$

During the learning phase, a histogram is maintained to facilitate the calculation of μ and σ. These values are stored in persistent storage to be used during detection. In the detection phase, the threshold is calculated as explained above with the statistical moments provided from the files, and the false alarm rate determined from the specification.

Chapter 6

Agents

Information needs to be extracted from the data sources and converted to a standard form that can be easily processed by the anomaly detector. This function is carried out by programs referred to as *agents* [13].

Agents basically extract information representing the state of the system or service and transform it into *events*, a standard self-describing format. Any set of information derived from the log files, the data sources in this case, can be abstracted as an *event*.

An *event* is defined by a name and a set of parameters.

$$eventname(param1, param_2, param_3, \ldots, param_n)$$

Every agent is associated with a particular logfile and contains the definition of events to watch out for. An agent recognizes the information for an event, and then creates the event alongwith its parameters from the corresponding entry.

It is upto the system administrator to determine what events she would be interested in and which log entries would constitute such events. A detailed description of the specification of these agents will be provided in later sections.

6.1 Developing Agents

Agents are specified by means of *Agent Definition Language* (ADL), a rule based language. This specification language allows users to define events and their parameters. The next section will describe the syntax of this language in greater detail.

6.1.1 ADL Syntax

ADL is a rule based language containing rules of the form *pattern–>action*. The input is scanned for the *pattern*. A matched pattern results in the execution of the

corresponding *action*.

ADL syntax is very similar to that of lex. Its source consists of regular expressions and corresponding program fragments. The format is as shown below

```
macros
%%
rules
%%
user C++ code
```

Similar to lex, any text enclosed within % and % from column 1, is copied to the generated code without modifications. This section typically contains preprocessor statements and comments. The semantics of each of these sections can be described as follows:

Macros

This section contains definitions of macro-like variables that can be used in the rules section. A *macro* is a name equated to a symbolic expression to which it is expanded at compile time. The use of macros greatly simplifies the task of writing rules. It also enhances the readability of the specifications.

A macro can be defined by either regular expressions (R.E.) or other macros.

$$macro(macro\_args) = (\text{macro\_args} = \text{R.E.}) \text{ — } \text{macro}(\text{macro\_args})$$

The following example should make the above described concepts clearer. In this example, we define macros for date and time using regular expressions. We define another macro for timestamp using the already defined macros date and time.

```
date(dd,mm) = (mm = [A-Z][a-z]*)(dd = [0-9]+)
time(h,m,s) = (h = [0-9]+):(m = [0-9]+):(s = [0-9]+)
timestamp(H,M,S,DD,MM) = date(DD,MM)time(H,M,S)
```

The regular expression is assigned to the arguments in the macro. Also, when an earlier declared macro is used in subsequent macros, it expands into its definition. If the macros take arguments, then the expansions replace arguments of the caller in the defintion. While using earlier declared macros, the syntax used is {*macro*}.

Rules

The rules are expressed using the following syntax:

pattern –>*action*.

Here, *pattern* comprises of a regular expression or a macro, or both. The regular expressions are similar to Perl regular expressions as ADL derives its pattern matching from Perl. The main difference between the two forms of regular expression is that Perl specifies implicit variables to carry the value of backreferences. However, ADL allows explicit variables to be expressed that are assigned the values of the matched group. For example, to extract the hour, minute and seconds component from the timestamp using ADL, we express as follows:

$$(h=\texttt{[0-9]+}):(m=\texttt{[0-9]+}):(s=\texttt{[0-9]+})$$

On the other hand, Perl would use implicit variables \$1, \$2 and \$3 respectively to denote hour, minute and seconds from a pattern that matches the following regular expression:

$$\texttt{([0-9]+):([0-9]+):([0-9]+)}$$

The explicit variables used in ADL are used to extract subgroups of the matched pattern, to be used as event arguments.

The *action* component denotes what needs to be done when a match is found for *pattern*. It consists of program segments (C++ code). These program segments, typically perform construction of events to be sent across to the detection engines.

User subroutines

This section is used to implement any C++ routine that can be used in the *action* component of a rule described above.

Just extraction of events is not sufficient. It is important to be able to access values in the log entries as parameters to be used later for learning the properties. The availability of classes like Message, Event and Param allow us to encode the matched pattern arguments. Each matched argument can be represented as a `Param` object. The `Param` object contains the value of the argument and the data type as assigned by the user. For example, the host name should be of type `STRING`. Hence, the object `Param` representing the host name, will contain its value and associate the data type `STRING` to it. Some other examples of the data types that can be specified are `HOUR`, `YEAR`, `INT`. All the `Param` objects from a particular log entry is bundled into an `Event` object. Each event has a user specified name associated with it which is assigned in the `Event` object. All the events extracted from the log file and represented as `Event` objects are contained in a single `Message` object that can be processed by the anomaly detector.

6.2 Agent Manager

The *agent manager* is responsible for loading and initializing the agents for a particular host. This information is provided to it in a configuration file.

Configuring the AM involves providing it with a configuration file. This configuration file is read when the AM starts up and is all set to initialize the agents. The configuration file contains details required to map the agents with the corresponding input data source. Syntax of the configuration file is as follows:

<i/p ><lib >{ I | F <pos >} { M | P <read_time >}

<send_time>

i/p : location of input data source
lib : agent library
I : scan for patterns incrementally
F : scan for patterns from a fixed point
M : scan when input is modified
P : scan in periods of read_time
send_time: how often the events need to be sent

The configuration file must contain a line for each agent to be managed by the Agent Manager.

6.3 Deployment

Once the agents are specified using ADL, as described above, they are compiled to generate a C++ and Perl file. The agent is driven by the C++ source which is compiled into a dynamically loadable shared library. It invokes the Perl script to extract the events from the log files by matching the *pattern* specified. The matched patterns are returned to the library. These matched patterns are assigned to the explicit variables specified in ADL, by the library. Then the corresponding routines specified in the *action* component are executed for each of the matched patterns.

An *agent manager* is deployed on every host that requires monitoring. When the AM is up and running, it first loads and initializes all the agents as per its configuration file. It functions like a listening server by listening for requests from Detection Managers for the extracted events. When it receives a request, it forks a child process to initialize and load the relevant agent library to take care of the request.

Chapter 7

Results

7.1 Experimental Infrastructure

This section describes the setup used to carry out the experiments.

7.1.1 Creation and compilation of agents

Agents need to be created to extract events from the log files.

The agents are specified using the ADL. Its rules section allows us to specify the patterns that need to be extracted from the data source along with the actions to be performed on that information. Each pattern is expressed as a regular expression. The use of macros makes it all the more convenient to express the patterns. In the *action* component of a rule, we can specify what data types should be assigned to the arguments extracted from the log record. The arguments can be represented using the data structures explained in chapter 6.

The agent definitions are specified using ADL in files with extension `.pat`. These are then compiled using the *LFM compiler*. From the specification, the compiler generates a Perl file and a C++ file which is then compiled into a shared object. The agent uses the Perl script to match the specified patterns with the entries in the log files. The agent also conducts periodic polls for recently created log files that need to be monitored. The shared object and Perl script are maintained in a particular directory that needs to be specified to the Agent Manager. This compilation completes the creation of agents.

7.1.2 Configuration of Agent Manager

The next step is to configure and get the **Agent Manager** (AM) running on every host that needs monitoring. For details regarding configuration, refer to chapter 6. The Agent Managers are responsible for loading and initializing the agents at

runtime. Agents and Agent Managers need to be deployed on every host machine containing the data source. If the logs are collected at a central system, then these entities need to be deployed only at the central site.

7.1.3 Setting up of Detection Engines

This is the key step in our experiments along with specifying agents. The **Detection Engines** are statistical anomaly detectors. They are defined by using the specification language *SMSL*. This language allows us to easily specify the transitions, the state variables, and most importantly the properties we want to learn and use in detection. For more details of *SMSL* refer to chapter 3.

Each DE is interested in a certain set of events that it can obtain by contacting one or more agents. Though the DE is flexible enough to include all and sundry set of events, it would seem more logical that one DE should be set up to handle a particular category of events. The word *category* again has a very loose definition. In the experiments described in section 7.2, the DEs were designed to work with only related sets of events. For example one DE was setup to detect anomalies in the web server's error log files. While another DE was set up to detect anomalies in the *messages* log file used by `syslog` utility.

Using the specification language, the events are declared. Each event has a name and a set of parameters associated with it. It has the following syntax:

`event` *eventName*`(`*param1*`,` *param2*`, ...`*paramN*`)`

As the properties of the system are mapped to the properties of transitions of state machines, these transitions need to be defined. Events or sequence of events are used to define a transition. Thus, if the aim is to study the frequency with which an event or a sequence occurs, it suffices to study the frequency with which the transition happens. The transitions can be defined using the following syntax:

transName`:` *eventName*`(`*param_list*`)` | *cond* `-->` *action*

Finally, the properties to be observed must be specified. This support is provided by the *on statement*. A combination of transitions and state variables can be used to express a *property*. Using this statement, the system can be instructed to measure the frequency with which events occur or record the values of specified state variables that are often seen. In our experiments, we have only specified frequency measurement. Anomalies in new values can easily be detected by measuring frequencies for state variables using the *wrt* clause. During detection, if a new value is seen, then the system assigns its threshold as 0 as it was not seen during learning. Thus an alarm is raised since its count (at least 1) crosses the threshold. Many of the anomalies detected in our experiments have been in new values not observed during training.

The specifications are given in a file with extension `.asl`. The *SMSL compiler*,

generates a C++ file which is compiled into a shared library that can be dynamically loaded by the Detection Manager. The compiler must be provided with a file that maps the expected event names with numerical event identification. The event names must be the same as the names that Agents use to tag the events they extract. This concludes the creation of the Detection Engines.

7.1.4 Configuration of Detection Manager

The **Detection Manager** is responsible for loading and initializing the Detection Engines. The DM also carries out the administrative function of contacting the AMs depending on its configuration. When it receives the events from the AMs, it forwards it to the appropriate DEs to be processed. It also carries out some pre-processing of the events to extract the parameters that are to be provided to the DEs. It also determines the timestamp of the events from the time-related arguments (recognized based on data type) and provides this value to the DEs. This is an important piece of information for the DEs as all their timing information depend on event timestamps.

The configuration of the DM also involves a configuration file that contains details about the DEs, the name of the machine hosting the agent managers to be contacted and a file listing the events that the DE is interested in. This file must contain the names that will be used by the Agents to tag the events that they extract. For every detection engine to be managed by the DM, there should exist a line in the configuration file with the necessary details. The syntax of the configuration file is as follows:

<host ><detection engine ><eventMap >
host : name of the machine hosting agent manager
detection engine : the detection engine library
eventMap : location of file containing the events for *detection engine*

The DM can be started in either the learning or the detection mode.

The configuration file and the mode in which the detection engines should operate are run-time options and need to be specified as command line arguments.

7.1.5 Experimental Data Set

Before describing the experiments, it would be useful to visit some background details of the logging syntax of the files used in the experiments.

System service and kernel logs

Syslog is a protocol for collecting log information from the kernel and various programs and devices. The logging daemons (syslogd and klogd) based on this proto-

col form a dedicated logging subsystem that collects messages from heterogeneous sources and puts them in a central repository. This helps in imposing a basic structure to the log entries facilitating automated analysis. Every log entry will, at the minimum, have a timestamp and hostname associated with it. Using *syslog.conf*, it can be configured to collect the logs of different programs in different files. For example, all mail logs can be saved in *maillog*. The usual location for these log files are */var/log*. Also, most log messages from different sources are, by default, collected in the file */var/log/messages*. We can depend on this file to provide us operational information of various sources. It is the most suitable place to look for information regarding failures and intrusions as well.

Mail logs

Mail servers form an important part of the network infrastructure of any establishment. With the emergence of email as an important and daily form of communication, it is not difficult to imagine either the volume of logs generated or the amount of valuable information contained in these logs. While there are several mail transfer agents, this experiment was carried out on logs generated in machines that used *sendmail* as its MTA.

Sendmail also uses syslog facility described above for logging. The messages that are logged are determined by the log level specified in the configuration file *sendmail.cf*. A log level of 0 will result in no messages being logged. As the level is increased, lesser critical messages are also logged. For the anomaly detector to be knowledgeable of the expected behavior, it is important that a reasonable amount of information is logged. Values from 1-10 contain useful information for system administrators. In this experiment, a log level of 9 proved to be sufficient to determine a whole range of significant events.

Each log record has the following format:

`<timestamp ><host name >sendmail[<pid >]: <qid >: <message >`

For a detailed description of each field in the log record, refer to [17].

The log entries could pertain to message transfers or various other events like configuration errors.

In the mail logs, a log record is created for each of the following:

- The receipt of a message.
- Every delivery attempt made.
- Miscellaneous situation.

The miscellaneous messages can either be error messages or non-errors. These messages can provide valuable insights to performance issues and failures in the host as was discovered in this experiment.

Web server logs

Web administration is another challenging task as internet services have sprung up everywhere. Feedback regarding the activity and performance of a web server is important for its smooth functioning. The Apache HTTP server provides flexible logging capabilities. There are two forms of logs that are of interest here:

Error Logs These logs contain a lot of information as the apache http daemon logs all the errors it encountered during processing of requests, in this file. It is the place to look when something goes wrong. The log entries are descriptive and have the following format:
<timestamp >[error] [client <source-ip >]: <message >

A wide variety of messages are logged here. Some of these messages are debugging information written out by CGI scripts. Some of the entries in the error files have corresponding entries in the access log.

Access Logs The server logs all requests processed in its access log. The format can be easily configured unlike the error log format. The common log format is as follows:
<remote host ><ruser ><auth-user >[<timestamp >] <request ><resource ><return-code ><bytes-sent ><user-agent ><referer >

7.2 Observations

The experiments were conducted to observe whether this technique was effective in identifying significant events related to the state of a system, while at the same time suppressing events that do not have much importance. The latter clause is an important measure, as the main motivation for automated log analysis is to extract meaningful events from the large volume of routine entries.

The results are quantitatively expressed in terms of the number of false positives and false negatives generated.

The following sections discuss the choice of log files selected for the experiment, the events monitored for, and the observations and results.

7.2.1 System Logs

As explained earlier, most of the processes in a system usually log their activities in */var/log/messages* using the *syslog* utility. This is an appropriate data source to detect anomalies in the system.

For this experiment, *messages* file from a local host as well as from a server in the intranet was chosen for analysis.

In order to identify events, it helped to categorize the messages that were observed in many samples of this file. The events identified were as follows:

- **Type 1:** Successful starting up of services.
- **Type 2:** Successful shutting down of services.
- **Type 3:** Failed startup of services.
- **Type 4:** Failed shutdown of services.
- **Type 5:** Authentication failure.
- **Type 6:** Opening and closing of sessions.
- **Type 7:** All other messages were tagged as miscellaneous events.

The above were mapped into seven different transitions which will be referred to as `trans1`, `trans2`, `trans3`, `trans4`, `trans5`, `trans6` and `trans7` respectively.

The statistics collected were as follows:
`ts{1, 100, 10000, 100000}`

1. The amount of activity observed on a per-service basis on each host in `ts`.
   `on all frequency wrt (host) size[25] wrt (service) [100] ts`
2. The number of authentication failures observed on a per-service basis on each host in `ts`.
   `on trans5 frequency wrt (host) size[25] wrt (service) [100] ts`
3. The number of sessions opened and closed on a per-service basis on each host in `ts`.
   `on trans6 frequency wrt (host) size[25] wrt (service) [100] ts`
4. The number of shutdown and startup failures observed on a per-service basis on each host `ts`.
   `on {trans3, trans4} frequency wrt (host) size[25] wrt (service) [100] ts`

In order to study how effective this technique was to detect failures, two scenarios were created.

Multiple failed accesses In the first one, I tried to log into the local host using different user names. Wrong passwords were provided so that each attempt would result in an authentication failure. Similar attempts at access to the host was tried through ssh, again resulting in authentication failures. This scenario is similar to the real world scenario of either brute force and dictionary attacks, or some internal system problem.

This suspicious chain of events was detected as an alarm was generated for statistic (2). It reported the occurrence of 11 authentication failures logged for `sshd` in 10000 seconds. Since the access attempts were made manually, this event would not have been detected in 1s or 100s. Thus the provision to specify multiple timescales help to detect anomalies that manifest in different scales of time.

Disconnection from the network The second scenario was one in which the local host was disconnected from the rest of the network. This was done by simply disconnecting the ethernet cable. This resulted in an error message informing that the eth0 link was down. However, this went undetected as it was a single event that was tagged as a miscellaneous event. Since the occurrence of this event did not cross the threshold, no alarm was generated resulting in a false negative.

The host was running an nfs client and when a request for a remote file was submitted, messages were logged informing that the nfs server was unreachable. However, even though the nfs client continues to contact the server, no further messages are logged. Since this did not contribute to an increase in the count of this event for nfs, it went undetected. This exposes the limitation of the anomaly detector to spot events that result in less number of log messages. However, the network connection failure was detected in a different experiment that is described in the next section.

When the detection engine analyzed the logs from the server, an interesting set of events resulted in an alarm for statistic (3). On examination, a series of sshd sessions were seen to be opened and closed within a very short interval of time, by user with id 0 (root). The alarm reported 12 such events in 100s. Clearly, this could only be achieved by an automated process and not by manual effort. On investigation, it was found that this was because one of the users has used *rsync* over *ssh* to synchronize data from the laptop and the home directory. *Rsync* opens an ssh session for every directory specified. This turned out to be a false positive. However, it is useful for events of this nature to be brought to the attention of system administrators.

The above mentioned results were based on anomaly in the frequency of occurrence of events. Another category of anomalies detected involved the appearance

of new services that were not encountered during the training phase. This resulted in a lot of false positives. During the analysis of the server's log files, alarms for twenty-two new services were generated. A postmortem revealed that many of these services were run rarely. For example, an alarm was generated for the service *md* as it was a new value seen. The *md* driver is used to provide virtual devices that form a *RAID*, created from independent underlying devices.

The reporting of new values seen has its benefits. Some of the services detected might be ones spawned by applications installed by users in the network. This way a system administrator is aware of the possible applications owned by users, and can make informed decisions whether a user is violating a policy. However, this benefit will be masked by the potentially large number of false positives it can result in.

7.2.2 Mail Logs

The mail logs for this experiment were collected from two different sources. The first source was a host machine that is a part of our intranet (small network). The other source is a mail server used for the faculty and employees of the entire university.

The events defined for the mail log agent corresponded to receipt of a message, attempt to deliver a message and others. Three types of events were defined for the mail log agent.

- **Type 1:** The receipt of a message.
  `mail_from(timestamp)`
- **Type 2:** The attempt to deliver a message. This event has a parameter indicating the status of the attempt. For example, a message sent successfully will have a status *Sent*.
  `mail_to(timestamp, status)`
- **Type 3:** All other messages that did not belong to the above types.
  `mail_misc(timestamp)`

The above events were mapped into three different transitions in the detection engine. In the following description, these transitions are referred to as `trans1`, `trans2` and `trans3` respectively.

The statistics collected were as follows:
`ts {1, 100, 10000, 100000}`

1. The number of messages received per time period in `ts`.
   `on trans1 frequency ts`
2. The number of messages of a particular status observed over time periods in `ts`.
   `on trans2 frequency wrt (status) size[100] ts`

3. The number of miscellaneous events observed over time periods in `ts`.
   `on trans3 frequency ts`

4. The amount of activity observed by the mail server over time periods in `ts`.
   `on all frequency ts`

In order to test if the detection engine would be able to detect large volumes of mail leaving the system, a script was written to generate over 10000 mails. This was run on the local host that would typically be used by a single individual at any point in time. Hence the generation of a large number of mails within a small interval of time, on this system, should be cause for concern. This was immediately detected as an anomaly in statistic (2). During learning, not more than 1 mail per second was sent. However, in the anomalous state, 17 outgoing mails were detected in a second.

A more interesting set of experiment were ones that were conducted to discover system resource related failures by detecting anomalies in mail logs. The following situations were created in the local machine:

Low memory By creating a very large file, the total available disk space was reduced to less than 10MB. The above mentioned script was executed. This scenario would be similar to the activity seen by a mail server in a large network.

During detection, a large number of alarms were generated for statistic (3). An event count of 50 per second was detected. During the learning phase, it was observed that the usual miscellaneous messages did not exceed 3 per second. These messages usually referred to rebuilding the alias database. However, in this case, the miscellaneous events generated indicated low disk space. The mail server requires space for queuing its messages that are ready for delivery. In this situation, when the system's disk space was already low, the mail server was unable to queue its messages, thus generating several such messages.

High Load While conducting log analysis of a mail log from the local host, alarms for statistic (3) were generated. On examination of the log files, several messages logged regarding refused connections due to high load in the system was seen. Sendmail can be configured to prevent sending or receipt of mails if the system load is above a threshold.

Unavailability of network In this experiment, the host was disconnected from the network. Following this, an attempt was made to send several mails. During analysis of the logs, the detection engine threw an alarm for statistic (2) for queued messages. On examination of the log files, it was obvious that a lot of mails were queued and not sent. Sendmail would queue its messages when an immediate

delivery is not possible. In most cases, the messages are queued for a very short duration owing to small network delays. The reason for so many messages being queued for a long time indicated a more serious problem. Even though the real problem was not evident here, the existence of one in the system was established. More information about system related issues is available in the */var/log/messages* file. This entries in this file was scanned for any useful message around the time the alarm was generated. A message stating that the ethernet link was down was logged around that time. This helped in diagnosing the problem.

An easier way to diagnose this problem would have been by noticing a number of miscellaneous messages indicating timeout of connection while awaiting response from the destination. However, these events were not detected by the detection engine despite an increase in the count for statistic (3). This was mainly because the false alarm rate was specified as 0.01 resulting in a higher threshold.

There was another event that did not get reported, thus contributing to the false negative count. This event was an authentication warning issued when the user invoked sendmail with the mail client *pine* resulting in the use of *-bs* switch option. This warning is generated by sendmail because pine attempts to change the *from* address in the mail header. An administrator should be informed of this warning as it could be a case of *mail forgery* wherein the sender's address is changed. The detection engine did not catch this warning as it was a single warning that did not cause a significant increase in the number of miscellaneous events.

7.2.3 Web Server Logs

For this experiment, the http access and error logs of the web server in our intranet facility were utilized. Both these types of logs do not exhibit potentially varied events as each log record corresponds to a request processed or an error encountered in the processing, respectively.

HTTP Error Logs

In the case of error logs, only one event was defined in the agent. Every log record was an instance of this event. The parameters of the event consisted of the timestamp, the source ip initiating the request, and the url that was requested.

`http_error(timestamp, ip, url)`

The statistics learnt were:

`ts {1, 100, 10000, 100000}`

1. The number of requests per url over timeperiods in `ts`.
   `on httpe_t frequency wrt (url) size [500] ts`

The number of urls accessed is usually very large. This will result in a lot of false alarms being raised as new urls are encountered that was not seen during training.

To reduce this, it was made sure that the agents extracted only messages that were not accesses to *.html, *.txt, *.css, and images.

The analysis of these logs helped in discovering some attack scans. However, there were many false alarms generated unlike the previous two experiments.

Scans for *main.php* These scans attempt to find access to *main.php* in various directories trying to gain access to *mySql* via the *phpMyAdmin* interface. This interface is used to administer *mySql*.

This scan resulted in a number of alarms for each directory that was accessed to find *main.php*. The alarms were generated because these urls were not encountered during training. The corresponding log entries showed that these scans originate at three different ip sources. Using this information, the http access logs were scanned. No successful hits were discovered, thus confirming that the scans had not been successful.

Scans for *xmlrpc.php* Any system that is affected by the *Linux.Plupii* worm, issues requests for various urls generated by the worm. These requests are issued to find another victim that can be compromised. It tries to access resources that are known to have vulnerabilities. For example, the *XML-RPC* is known to have the *PHP Remote Code Injection* vulnerability.

This case was similar to the previously described scan. Analysis of the log showed that these scans were generated from a single ip source. The absence of a successful hit in the http access logs, for this ip source, indicated that the web server was not compromised. This is expected as the anti-virus software is knowledgeable about this worm.

Scans for *guestbook.pl* Scripts like *guestbook.pl* and *guestbook.cgi* are freely available for download. These scripts are written to provide a guest book on any home page. They can be customized to suit the user's need. These scripts are vulnerable to *cross-site scripting* attacks.

These scans originated from a small number of ip sources. However, there was a successful hit for this resource in the http access logs. This was because one of the users in the intranet did host a guest book. There were no signs of any successful attack. This could probably be attributed to the fact that the anti-virus utilities on the web server had the necessary updates to prevent an untoward action.

In general, these logs have too much information and can give rise to a lot of false positives as was observed.

Chapter 8

Summary

8.1 Conclusion

From the results observed in chapter 7, this technique can be effectively applied to indicate the possible existence of a problem. It does not directly indicate the nature of a failure or attack, in most cases. This is consistent with the objective we had stated. Information in a log file typically indicate the possibility of an unexpected situation. This information helps system administrators in further analysis.

It must also be stated that while some log files are suitable for anomaly detection, it is not convenient to use it on some other kinds of log files. Anomaly detection on mail logs provided very interesting results with a small number of false alarms. Analysis of */var/log/messages* was not as successful as that of mail logs. Yet, it also produced certain results that normally would have not caught the eye of an administrator. With a little more information logged on behalf of some of the services, some of the undetected anomalies could have been unearthed. For example, if the *nfs client* logs a message for every attempt to contact the server, the anomaly detector would have been able to capture this information and raise an alarm.

This technique was less effective for monitoring web servers. The false alarms generated were very high. On the other hand, attack scans were easily detected and did not require the system administrator to be already aware of these vulnerabilities.

From these observations, it can be stated that the effectiveness of this technique depends on the nature of the service and the amount of logging information available. Failures, performance issues or security breaches that result in generation of a large number of events, can be effectively detected.

The effectiveness of this technique also depends a lot on the quality of the training data. Some of the false negatives observed in the experiments were because they were present in the training data as well. For example, an analysis of the university's mail logs did not detect the presence of local configuration errors. This was mainly

because the training data also had several such errors. The training data did not actually represent an error-free system.

A merit worth mentioning is the reduction in dependency on the system administrator's experience and knowledge to select properties to be observed. With just a basic understanding of each log file, and specification of simple rules, a fair amount of significant information could be extracted by this technique.

8.2 Future Work

The current algorithm for learning, takes into account all the events extracted from the training data. This might not be necessary. Since a distribution is being built during learning, the possibility of convergence exists. In this case, convergence of a distribution refers to that point in time after which the values of the distribution do not vary significantly. The algorithm can be further optimized by adding this check for convergence. Thus, for very large training sets, it need not unnecessarily continue learning after convergence.

Since the effectiveness of this technique depends a lot on the training data, we intend to have a feedback mechanism in place. During detection, some events that are rare, generate alarms. However, these rare events might not be very significant. For example, the events that are generated when a server is restarted, would result in alarms. If the system administrator could indicate the insignificance of these events in the next iteration of training, the false alarms generated would be greatly reduced. The confidence in the system using this technique will increase.

Bibliography

[1] R. Sekar, A. Gupta, J. Frullo, T. Shanbhag, A. Tiwari, H. Yang and S. Zhou. Specification-based Anomaly Detection: A New Approach for Detecting Network Intrusions. Proceedings of ACM Computer and Communication Security Conference, 2002.

[2] R. Sekar, Y. Guang, T. Shanbhag and S. Verma. A High-Performance Network Intrusion Detection System. Proceedings of ACM Computer and Communication Security Conference, 1999.

[3] 2Swatch ftp://ftp.sdsc.edu/pub/security/PICS/2swatch/README

[4] S. Hansen and T. Atkins. Automated System Monitoring and Notification with Swatch. USENIX Systems Administrator (LISA VII) Conference Proceedings, pp. 145-156.

[5] Logwatch. Kirk Bauer. http://www.logwatch.org/.

[6] SLAPS-2. http://www.openchannelfoundation.org/projects/SLAPS-2/.

[7] Spong - Systems and Network Monitoring. http://spong.sourceforge.net/.

[8] Open Source Host-Based Intrusion Detection System. http://www.ossec.net/.

[9] R. Vaarandi Simple Event Correlator http://kodu.neti.ee/ risto/sec/.

[10] Logsurfer+. http://www.www.crypt.gen.nz/logsurfer/.

[11] Logsurfer. http://www.cert.dfn.de/eng/logsurf

[12] M. Blum, R. Floyd, V. Pratt, R. Rivest and R. Tarjan. Time Bounds for Selection. Computer and System Sciences, 1972.

[13] M. Channa-Reddy. Monitoring Log Files for Detecting Intrusions and Failures: A language based approach. Master of Science Thesis, Computer Science, Stony Brook University, May 2004.

[14] M. Bishop. Computer Security: Art and Science. Addison-Wesley Publishing Company, 2003.

[15] R. Jain. The Art of Computer System Performance Analysis. John Wiley and Sons, Inc., 1991.

[16] http://en.wikipedia.org/wiki/Chebyshev_inequality

[17] http://www.sendmail.org/ ca/email/doc8.10/op-sh-2.html#sh-2.1