Monitoring Log Files for Detecting Intrusions and Failures: A language based approach

by

Mohan-Krishna Channa-Reddy

to

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

Master of Science
in
Computer Science

Stony Brook University
May 2004

Stony Brook University

The Graduate School

Mohan-Krishna Channa-Reddy

We, the thesis committee for the above candidate for the

Master of Science degree,

hereby recommend acceptance of this thesis.

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

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

Professor Erez Zadok,
Computer Science Department

This thesis is accepted by the Graduate School.

Dean of the Graduate School

Abstract of the Thesis

Monitoring Log Files for Detecting Intrusions and Failures: A language based approach

by

Mohan-Krishna Channa-Reddy

Master of Science

in

Computer Science

Stony Brook University

2004

Professor R. C. Sekar

System and network management is one of the most challenging and difficult tasks. One of the tasks an administrator may be required to perform is to monitor systems for signs of intrusions, performance problems and faults. These signs are generally characterized as unusual activity. In a large network, with large amounts of activity, the job of identifying the unusual activity from normal activity, can be like looking for a needle in a haystack. Existing tools target specific types of services for monitoring. This requires administrators to have knowledge about various tools. Managing these diverse tools increases the workload of administrators.

We present a generic and platform independent framework that can simplify the task of administration. This framework has two components- distributed *agents* on systems that need to be monitored and a centralized *detection engine*. Agents are used to extract status of systems or services. Detection engines detect systems or services that are bad or unusual. Agents feed the detection engines with a homogeneous stream of events. We developed a language for developing such agents easily. Detection engines process these events and detect unusual activity. The detection engines should have low false rates and be able to detect any unusual activity. The detection engines employ state-machine specifications of the event patterns and augment the state machines with statistical information that is needed to detect unusual behavior. We developed a state machine specification language to capture the event patterns and the statistical information in a succinct manner.

Table of Contents

List of Figures

ACKNOWLEDGEMENT

I would like to take this opportunity to express my thanks to those who helped me with various aspects of conducting research and writing of this thesis.

First and foremost, Dr. R C Sekar for his guidance, patience and support throughout my research and writing of this thesis. His insights, advice and support have often inspired me during the course of my graduate education.

I would like to thank Dr C R Ramakrishnan and Dr Erez Zadok for their contributions to this work.

I would also like to thank Sandeep Bhatkar for the valuable time he spent with me discussing issues.

I would additionally like to thank my friends Prem Uppuluri, Venkat Krishnan, Ajay Gupta, Zhenkai Liang, Wei Xu, Alok Tongaonkar, Weiquing Sun, Shruthi Murthy and Salil Gokhale for their assistance and valuable inputs at various times during my research.

Chapter 1

Introduction

Today's networks are composed of a large number of heterogeneous systems. Each system may provide diverse services. These services are expected to have the following properties:

- *Availability:* They can be accessed and used in a uninterrupted manner.
- *Reliability:* Their behavior is dependable.
- *Security:* They are infringable to hostile acts.

It is the administrators responsibility to ensure that systems and services satisfy the above three properties. This task is difficult due to the possibility of various types of *failures* that can leave the systems and services unavailable, unreliable or insecure. These failures can be due to bugs in software or hardware or a combination of both. Formally, a bug is an error or a defect in software or hardware that may cause a system to malfunction. The bugs in software can be due to coding errors or configuration errors. The bugs in hardware can be due to design flaws, lack of maintenance or wear-down.

These bugs eventually manifest as failures affecting performance and security adversely.

The major problem with respect to detection of failures in a network is size of network and diversity of activity. The size of the network depends on the number of systems, services and users. The diversity in activity is due to the variety of services offered. Both size and diversity also increase the possibility of failure.

Generally, early detection of failures prevents disruption of service. Unusual activity is an indication of possible presence of bugs or failures. So, our work is based on the idea of monitoring the systems and detecting unusual activity. The detection system can alert the administrators. The administrators can then perform necessary corrective actions before system functionality and performance is affected.

1.1 Our approach

Our work develops a framework for network wide fault detection systems. The framework is a generic, platform independent, configurable framework that makes it effective for fault detection. The basic idea behind our approach is to periodically monitor and determine the state of the system and services in the network. This approach entails two phases:

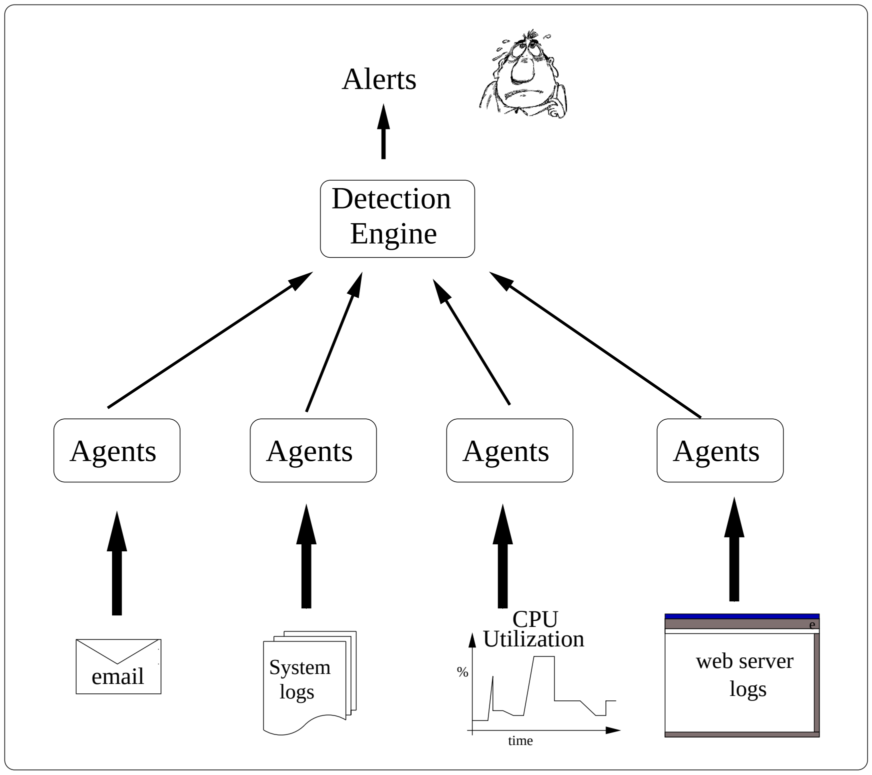


Figure 1.1: Our Approach

- *Monitoring:* Probe the current state of systems and send the probed data to a centralized detection system.
- *Detection:* Process the data and determine if their state is bad or unusual.

This two phase approach provides us an ability to correlate events across systems and services. The following sections briefly explain these two phases.

1.1.1 Monitoring

The goal of the monitoring infrastructure is to monitor systems and services. Systems and services provide mechanisms for monitoring, either implicitly through *log files* or explicitly through *system utilities*. *Log files* are text files that are used to record activity. *System utilities* are commands that give us information about the system. Most of the services use log files to log their activity. Using the log files and system utilities, we can piece together enough information to discover the unusual activity.

We employ *agents* to extract information from log files and system utilities. The information gathered depicts the state of the system or service. This information is converted into a standardized self-describing format, called event. An *Event* is an abstraction of information. The agents send these events to the centralized detection engine for processing.

Languages like lex and awk don't suffice for developing these agents. Both these languages have certain limitations. Lex doesn't support extraction of parts of the matched pattern. Awk supports extraction of parts (separated by a field separator) of the matched pattern but restricts a pattern to one line. So, we developed a language called *Agent Development Language* (ADL) for building these agents. This language consists of rules of the form *pattern* $\rightarrow$ *action*. The patterns specify what to extract, while the actions specify what to do with these extracted patterns. Typically, actions construct an event from the extracted pattern and send them over to the detection system. These ADL rules are compiled into an agent.

1.1.2 Detection

The goal of the detection infrastructure is to identify faulty behavior in systems or services. The detection system comprises of *detection engines*. The agents periodically feed the detection engines with events. The detection engines recognize activity that is unusual from the event streams.

The detection of faults is similar to detection of intrusion. Our detection system is derived from intrusion detection approaches. The common approaches adopted in intrusion detection system can be broadly classified [7] into two categories:

- *Misuse-based Approach:* This approach assumes that intrusion attempts can be characterized by sequences of user activities that lead to compromised system states. The misuse-based approaches are characterized by their expert system properties that fire rules when audit records or system status information begin to indicate illegal activity. These predefined rules(specifications) typically look for state change patterns observed in the audit data and compare them to predefined intrusion state change scenarios. Intrusion analysis draws conclusions using these rules to detect the presence of a suspected attack. This approach allows a systematic browsing of the audit trail in search of evidence of attempts to exploit known vulnerabilities. These approaches tend to have low false rates. The limitations of this approach are the difficulty of extracting knowledge about attacks and inability to detect novel attacks.

- *Anomaly-based Approach:* User or system behavior is measured by a number of variables sampled over time and stored in a profile. There are several types of measures in a profile. These include: relative frequency of logins and amount of CPU or I/O for a specific user. The current behavior of each user is maintained in a profile. Anomalous behavior is determined by comparing the current profile with the stored profile. This approach keeps statistics of all these variables and detects whether thresholds are exceeded. The advantage of this approach is the ability to detect novel attacks. The main limitation of this approach is the high false rates.

We borrow concepts from both these intrusion detection approaches mentioned earlier. We combine the two into a hybrid detection approach that mitigates their drawbacks and magnifies their strengths. The detection engines employ the hybrid detection approach. For the development of detection engines, we developed a language called *State Machine Specification Language* (SMSL). This language enhances an existing specification language, *Behavior Monitoring Specification Language* (BMSL) [3], that is used for specifying behavior of processes in host-based intrusion detection systems.

The SMSL specifications of patterns over events are compiled into state machines. We employed state machines as most services can be modeled using them. These state machines are augmented with the learning aspects of anomaly-based approaches. These form the detection engine. The detection engines are trained to learn normal behavior. The learned behavior is the compared with the behavior in the detection phase. Any deviation is flagged as an alarm.

1.2 Thesis Overview

In chapter 2 we explain the agent development language and the state machine specification language. We explain some of the language constructs and their semantics. We also introduce the mechanism adopted to gather statistics and how these statistics are employed in detecting anomalous activity. Chapter 3 deals with the the various components of the framework and the way they interact with each other. We also describe deployment issues of our framework. In chapter 4 we will discuss the current approaches. Specifically, we describe host-based approaches like Swatch [6], SHARP [2], LogSurfer [8] and network-based approaches like RedAlert [10] EMERALD [9] and AAFID [5] [1]. Finally in chapter 5 we conclude by summarizing our approach and explore the future scope.

Chapter 2

Design

We designed the framework with the following objectives:

- *Generic:* The monitoring system should be highly generic, i.e., it should not be bound to any kind of platform, OS, or format. The monitoring system should be able to monitor any service/device in the network. The detection system is generic by design, as it deals with a homogeneous stream of events feed by the monitoring system.

- *Easy-to-use:* Even though, we wanted our framework to be generic, we also want it to be easy-to-use. This was main motivation of developing languages for both monitoring and detection systems. These languages are very intuitive and derive construct from languages like lex, Perl and C, making it easy to learn. Thus, writing the ADL or SMSL specifications are easy. In addition we also took care to ensure that deployment of the system should be an easy task.

- *Configurable:* The decision of what devices to monitor and what service to monitor is entirely in the administrator's discretion. Also the actions to be taken upon finding faulty activities are defined by the administrator.

- *Effective:* For the system to be adopted it should be error free. In addition the system should be efficient, in bandwidth / resource usage, yet responsive.

The rest of the chapter is organized as follows: Section 2.1 describes the features of the *agent development language* (ADL), specifically the pattern matching aspects of the language. Section 2.2 explains briefly the *Behavioral Monitoring Specification Language* (BMSL) [3], a language to capture behavior of programs. This language served as a good starting base for developing the *state machine specification language*(SMSL). Section 2.3 describes the state machine, *mapping* and *timeout* constructs of SMSL. Section 2.4 describes the SMSL features towards anomaly detection, specifically the *on* construct. In this section, we also describe the learning and detection phase of our approach.

2.1 ADL

It is a rule based language, which takes multiple rules of the form *pattern* $\rightarrow$ *action*. The semantics of the rule are to scan the input for the patterns and execute the actions corresponding to matched patterns. In this section we describe the syntax of the language.

2.1.1 Syntax

The syntax of this language is very similar to that of lex. The input files consist of three sections, separated by a line with just '%%' in it:

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

The macros section contains declaration of *macros*, a name that is equated to a text or symbolic expression to which it is to be expanded to at compile time. The macros simplify the writing of rules. The rules section consists of a series of rules of the form *pattern* → *action*. And finally the user code section consists of C++ code that is used in the actions. Each of these sections are described below.

2.1.2 Pattern Matching

The most popular language for pattern matching used today is Perl [11]. This can be attributed to immense features that Perl supports for pattern matching, such as backreferneces, support for non-regular expressions and backtracking. The pattern matching of ADL was derived from Perl. Thus the regular expressions of the patterns are similar to that offered by Perl. However, there are some modifications to backreferences in the patterns, Perl uses implicit variables for backreferences, but we use explicit assignments to variables. For instance, in Perl, the regular expression for matching a decimal number as shown below would extract the real part and fractional part in implicit variables $1 and $2 respectively.

```
/(\d+)\.(\d+)/
```

While the same pattern can be written is ADL as shown below, where the real part and fractional part are extracted into variables *real* and *fract* respectively.

$$/(\textbf{real} = \backslash\textbf{d}+)\backslash.(\textbf{fract} = \backslash\textbf{d}+)/$$

2.1.3 Macros

From our experience of using ADL, there were many occasion where we would use similar patterns, such as date, time, IP, URL, etc. Thus we felt that a macro like feature would help the task of developing these specifications. The use of macros also improves the readability of the ADL specifications. These macros can be defined by regular expressions or by macros themselves.

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

Note that the R.E. above should assign to the macro_args. For example, we illustrate the definition of a macro for date and time and use these to define a macro for timestamp.

```
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 declared macros can subsequently be referred to using $\{macro\}$, which expands into their definition. If the macros take arguments then the expansion replace arguments of the caller in the definition.

2.1.4 Rules

Rules are of the form *pattern* $\rightarrow$ *action*. The patterns can be written as regular expressions or macros or both. The action part of rule is C++ code that can use the explicit variables declared in the pattern. Typically, we use these explicit variables to store extract from the matched patterns and uses these variables in the actions. Typically, the actions construct events, and send these events over to the detection system. We illustrate a rule that would extract a *logind* pattern and return an event of *LogindEvent* type.

```
{timestamp(h, m, s, day, mon) : (server = \s+): logind :(msg= \S+)
→ {
    sendEvent(LogindEvent(Time(mon, day, h, m, s, ), server, msg));
}
```

Here the *LogindEvent* used is a user defined event class, we discuss about these in Section 3.1.1. The code in the action part can make use of external functions provided by the language e.g. *sendEvent* or functions defined by the user in the user code section. The function *sendEvent* is a implemented by the language to send the event to the detection system.

2.2 BMSL

BMSL is a language that was designed for developing security-relevant behavior models. Specification in BMSL consist of rules of the form *pat* $\rightarrow$ *action*, where *pat* is a pattern on event sequences and *action* specifies the responses to be launched when the observed sequence satisfies *pat*. Note that we typically initiate responses when abnormal behavior is witnessed.

As a language that captures properties of(event) sequences, our pattern language draws on familiar concepts from regular expressions. It extends these concepts from sequences of characters to sequences of events with arguments, known as Regular Expressions over Events(REE).

The simplest REE pattern captures single event occurrences or non-occurrences:

- *occurrence of single event:* $\mathbf{e(x_1,\ \cdots,\ x_n)|cond}$ is matched by the (single-event) sequence $\mathbf{e(v_1,\ \cdots,\ v_n)}$ if $\mathbf{cond}$ evaluates to $\mathbf{true}$ when $\mathbf{x_1,\ \cdots,\ x_n}$ are replaced by $\mathbf{v_1,\ \cdots,\ v_n}$

- *occurrence of one of many events:* $\mathbf{E_1||E_2||\cdots||E_n}$, where each $\mathbf{E_i}$ captures an occurrence of a single event, is matched by a sequence if any one of $\mathbf{E_1,\ \cdots,\ E_n}$ is matched.

- *event non occurrence:* $!(\mathbf{E_1||E_2||\cdots||E_n})$, where each $\mathbf{E_i}$ captures an occurrence of a single event is matched by a sequence if none of $\mathbf{E_1,\ \cdots,\ E_n}$ are matched.

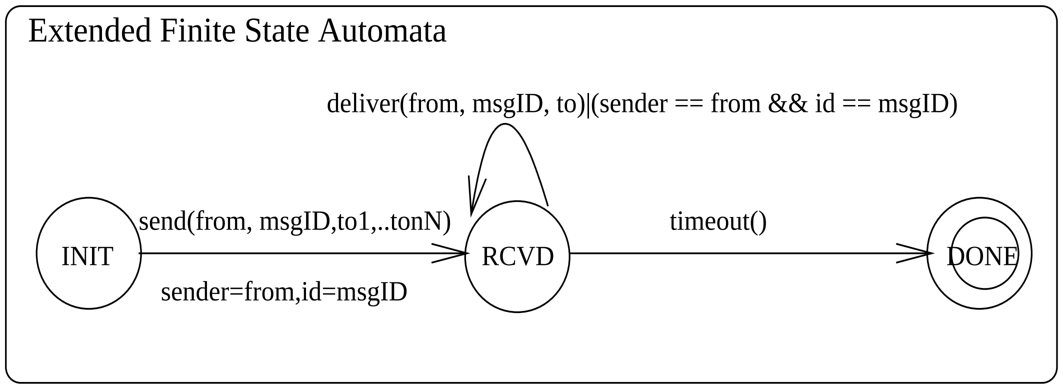
These primitive event patterns can be combined using temporal operators to capture properties of event sequences as follows:

- *sequencing:* $\mathbf{pat_1 \,.\, pat_2}$, is matched by sequence $\mathbf{s_1s_2}$ if $\mathbf{s_1}$ and $\mathbf{s_2}$ satisfy $\mathbf{pat_1}$ and $\mathbf{pat_2}$ respectively.

- *alternation:* $\mathbf{pat_1 \,||\, pat_2}$, is matched by sequence $\mathbf{s}$ if $\mathbf{s}$ satisfies $\mathbf{pat_1}$ or $\mathbf{pat_2}$.

- *repetition:* $\mathbf{pat*}$, is matched by sequence $\mathbf{s_1s_2\cdots s_n}$ if each $\mathbf{s_i}$ satisfies $\mathbf{pat}$.

The REE's have the ability to remember event argument values (for later comparison with arguments of subsequent events). This makes them much



Extended Finite State Automata
deliver(from, msgID, to)|(sender == from && id == msgID)
send(from, msgID,to1,..tonN)
sender=from,id=msgID
timeout()
INIT
RCVD
DONE

Figure 2.1: Sendmail Model

more expressive than regular languages. This expressive power is comparable to that of attribute grammars rather than regular grammars.

The language supports external functions implemented in the BMSL runtime environment, either to assign or test arguments of an event or state variables. The functions can be used to invoke response actions that are aimed at preventing and/or containing damage due to an attack. The functions can also be used to express computations that are not easily described in BMSL, but can be coded in general-purpose programming languages such as C++.

2.3 SMSL

The SMSL language models systems and services as a state machine. A traditional state machine has set of states, a set of input events and a transition function. We extended the state machine to remember values as it makes transition from the start state to the final state. These state machines can model applications and network protocols. For example a simplified model of sendmail

These state machines are represented as *Extended Finite State Automata*(EFSA), which augment traditional FSA with a set of state variable. Formally, an EFSA $\mathbf{M}$ is a seven tuple $(\mathbf{\Sigma}, \mathbf{Q}, \mathbf{s}, \mathbf{f}, \mathbf{V}, \mathbf{D}, \delta)$ where:

- $\mathbf{\Sigma}$ is the alphabet of the EFSA. It is an event alphabet, i.e., elements of $\mathbf{\Sigma}$ are characterized by an event name as well as 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. In our model, it is a sink state that 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.

Below, we describe our language for specifying EFSA.

2.3.1 State Machine Specification

State machines specification follow the EFSA definition given above.

- $\mathbf{\Sigma}$: The set $\mathbf{\Sigma}$ are specified as part of an interface declaration, which lists the events and the argument types. The events are delivered by the runtime to the detection engines via well-defined interfaces.
- $\mathbf{Q}$: These are declared using **states** $\{\mathbf{s_1}, \cdots, \mathbf{s_n}\}$.
- $\mathbf{s} \in \mathbf{Q}$: The startstate of the state machine can be specified using the declaration **startstate** $\mathbf{s}$.

- $\mathbf{f} \in \mathbf{Q}$: The finalstate of the state machine can be specified using the declaration **finalstate** $\{\mathbf{s_1}, \cdots, \mathbf{s_n}\}$.

- $\mathbf{V}$: These variables are declared using syntax similar to variable declarations in typical programming languages.

- $\mathbf{D}$: Since the set of variables in $\mathbf{V}$ are declared with types, the set $\mathbf{D}$ is redundant and not used.

- δ : The transition relation is specified using rules of the form $pat \rightarrow action$. pat, here refers to a REE pattern, over the event alphabet $\mathbf{\Sigma}$. The rules make transition from one state to another, by assigning to a special state variable **state**. The domain of this variable is $\mathbf{Q}$. This variable, just like any other state variable can be tested or assigned in the **cond** part of primitive event. The action part of the rules can invoke external functions or make assignments to state variables including the variable **state**.

These state machines are non-deterministic. We simulate non-determinism by cloning k copies of the state machine whenever it can make one of k different transitions. The cloning operation duplicates not only the states, but also all of the variables in $\mathbf{V}$. Clearly, we cannot have a situation where the number of state machine instances increases forever. To deal with this problem, we automatically delete state machine instances that reach their final state. Note that final states are some what different form "accepting states" of an FSA - they are similar to "sink" states from which no progress can be made.

In addition we added a couple of more construct to the language, these are described below:

2.3.2 Mapping

There can be many instances of a state machine at runtime. Thus, for each incoming event, we may have to search through all of these state machine instances to discover those that can make a transition. This operation can be very expensive, so we use a mechanism that speeds up this operation in a situation that arises frequently: often we use a state machine instance to track a particular "service" and the service to which an event applies can be computed efficiently from the event parameters. The following language construct is used to specify such mapping:

$$\textbf{map event}(\textbf{eventArgs})\ \textbf{when condition}$$

The **condition** component must be of a special form: it should be a conjunction of equality tests, where the left hand side of the test is an expression on **eventArgs** and the right-hand side is a state variable. Also, an additional constraint is that number of state variables used in these conditions should be the same. This restriction is imposed so that the identification of the right state machine instance can be implemented using a hash-table lookup.

2.3.3 Timeout

The language also permits timeout transitions to be delivered. Timeout values can be declared using one or more declaration of the form

$$\textbf{timeout t in}\ \{\mathbf{s_1},\ \cdots,\ \mathbf{s_n}\}$$

This declaration states that a state machine will stay in one of the $\mathbf{s_1}, \cdots, \mathbf{s_n}$ for at most **t** seconds. At the end of this period a transition associated with the special event *timeout* will be taken. This feature enables the state machine to be a timed, i.e., state machine can now capture temporal properties.

2.4 Anomaly Detection

Anomaly detection is dependent on good feature selection. Currently feature selection is driven by human expert's knowledge and judgment regarding what constitutes "useful information" for detecting attacks. While human experts are often in a position to identify some useful features, but surely not all. Often, their notion of a useful feature is influenced by their knowledge of known attacks. Consequently, they may not necessarily select features that are useful in detecting unknown attacks. Thus our approach attempts to bring a high degree of automation in the process of feature selection.

2.4.1 Learning

Anomaly detection is concerned with detecting "unusual behaviors". With our state machine models, we are ultimately mapping behaviors to transitions of the state machine. Thus, unusual behaviors can be detected if our approach learns how frequently a transition is taken, or the commonly encountered values of state variables on a transition. One obvious way to represent this information is as an average, e.g., the average frequency with which a transition is taken, However, it is well-known that network phenomena tend to be highly bursty, and hence averages do not provide an adequate way to characterize such phenomena. Therefore, in our approach, we focus on capturing *distributions* rather than averages. For this we support both frequency distribution (frequency of transitions) and value distribution (distribution of values for the state variables).

The representation of distributions differs, depending on the nature of the values in the distribution. If the values are categorical, then a distribution simply counts the number of times each distinct value occurs in the distribution. Often the number of possible categories that occur may be too large, so the distribution may represent only those categories that occur most frequently. If the values represent a discretized quantity, then the distribution can be represented compactly as a histogram.

Often, we are interested in properties that hold across a subset of traces. One way to select traces is based on recency, e.g., traces witnessed during the last **t** seconds. This would enable us to focus on recent behavior , as opposed to behaviors observed a long time in the past. A second way to select traces is based on values of state variables. For instance, we may be interested in:

- trace corresponding to a particular service.
- trace involving a particular host.

Statistical properties to be learned can be specified conveniently in our specification language using the *on* statement. We illustrate its use in below section.

2.4.2 On Statement

The *on* statements have the following syntax:

on {**all** | **any** | {$trans_1, \cdots, trans_k$ } }
[**when** cond]
{**wrt** $v_1, \cdots, v_m$ [**size** <size>]}*
timescales {$ts_1, \cdots, ts_n$}
falsealarmrate <rate>

As we can see the *on* statement has five clauses: *on, when, wrt, timescale* and *falsealarmrate*. The semantics of each of these clauses are as follows:

- *on :* This clause is used to specify a set of transitions $\{trans_1, \cdots, trans_k\}$ on which statistics will be collected. The keyword **all** indicates that statistics should be collected for all the transitions aggregated together, where as, the keyword **any** indicates that statistics should be collected for all transitions individually.

- *when :* This clause is used to control when the statistics should be collected. Only when the *cond* evaluates to *true* will it affect the distribution. The *cond* is a condition over state variables.

- *wrt :* **wrt** is an abbreviation for "with respect to". The statistics will be collected separately for each combination of values that these state variables $v_1, \cdots, v_m$ take, i.e., these variables form the key and statistics will be collected for each unique key value encountered. Each *on* statement can have multiple *wrt* clauses, this can be visualized as multiple levels of tables. The **size** part of each **wrt** indicates the maximum number of unique values of key to expect.

- *timescales :* This provides the ability to specify the time periods over which the number of times the transition was taken is counted. Use of short timescales enables faster fault detection. However, slow anomalous activity tend to be missed at shorter time scales. They can be detected by observing statistics over larger time scales, but those time scales imply longer latencies before attack detection. By using multiple time scales that range from a milliseconds to a thousand of seconds, will combine the benefits of fast detection of rapidly progressing anomalous activity, with delayed detection of slowly progressing anomalous activity. Each timescale $ts_i \in \{ts_1, \cdots, ts_n\}$ has its corresponding frequency distribution. Frequency distribution is obtained by splitting the range into $k + 1$ geometrically progressing bins(classes), e.g., $[0 - 1), [1 - 2), [2 - 4), [4 - 8), \cdots , [2^{k-1}, 2^k)$.

- *falsealramrate :* This provides the maximum desired false alarm rate. This will be used to calculate the appropriate counter threshold for generating an alarm based in distributions learned. A low rate means less alarms which might miss some anomalies, while a high rate will mean more alarms thus likelihood of catching most of the anomalous activity.

We illustrate an example of the *on* statements in the later chapter.

2.4.3 Detection

Typically in the detection phase, the statistics specified for learning are computed again, and compared with the values learned during the training phase. If the statistics vary substantially from what was learned, then an anomaly is raised.

Maintaining these distributions in the detection phase and comparing these distributions would impact the responsiveness of our system, thus during the detection phase we don't maintain frequency distributions nor do we have do we compare them with the learned distributions. But instead, we employ a simple yet very effective approach for detecting anomalies. We compute the thresholds for each timescale, from the distributions gathered in the training phase. The highest bin with nonzero count **Max Non-Zero Bin** (MNZB), for each timescale serves as the threshold $\mathbf{Thres_T}$ for that timescale.

And during detection phase we use simple counters $\mathbf{C_T}$ for each timescale $\mathbf{T}$. These counters maintain the count of the number of times the transitions were taken in the time period $\mathbf{T}$. To eliminate the quantization errors we make use of windowing. If these counters, have $\mathbf{n}$ windows and the duration of each window be $\mathbf{d}$ seconds, then they should satisfy $\mathbf{n} \times \mathbf{d} = \mathbf{T}$. Typically, the counters have 4 windows, so the duration of each window would be $\mathbf{T/4}$ seconds.

In detection phase, alerts are raised when a counter $\mathbf{C_T}$'s count exceeds the corresponding threshold $\mathbf{Thres_T}$. This simplifies the entire detection mechanism, as both the time and space complexities are very low compared to those of the training phase. But, the more important aspect for this system to be effective is low false rate.

Chapter 3

Implementation

The entire framework is composed of two major components: *agents* and *detection engines*. The monitoring system is deployed on all systems that need some kind of monitoring. The basic functionality of agents is feeding the detection with the events as they occur, while that of managers is to assimilate these event sent from the various detection system and detect anomalous patterns in the event stream.

The monitoring system is composed of two entities: *agents* and *agent managers*. Each system being monitored, has one agent manager (AM). The AM manages multiple agents (AG). Each AG monitors one service or application. So, each system may be configured to run many AG's, depending on the number of services that need monitoring. There is no communication between the AM's and detection system, instead the AG's communicate with the detection system directly.

The detection system is composed of two entities: *Detection Manager* and *Detection Engine*. The detection manager (DM) manages multiple detection engines (DE). The DM is configured to communicate with multiple AG's on various systems. The DM waits for events from the AG's, upon receiving the events, it feeds them to the appropriate DE. The DE's processes these events and flags an alarm on detecting anomalous patterns.

3.1 Agents

The specification of the agents upon compilation produces a C++ and Perl file. The C++ program is the core of the AG, this get compiled into a dynamically loadable library. The AM initializes the library and controls the behavior of the AG. This library manages and uses the Perl script for the purpose of pattern matching and extraction. The Perl script is employed for pattern matching since pattern matching in Perl is far superior than any pattern matching library that exists for high level languages such as C++. So the sole purpose of the Perl script is to match patterns in the input and return these matched patterns. The Perl script is invoked from the library, which in turn returns the matched patterns. The library then constructs the explicit variables of the patterns as specified in the specification, with values from the matched pattern and then executes the corresponding action part for each of the matched patterns.

These libraries support a few options:

- How often to invoke the script? Periodically or when input is modified.
- How is the input scanned? Incrementally or from a fixed point(beginning of input).
- How often should it communicate with the DE? As and when patterns are matched or periodically.

3.1.1 ADL Runtime

The language runtime provides a few classes to represent data, such as string, date, time, timestamp, IP address, MAC address. It also provides a base class to represent a basic event. The specification writer can derive from this class to specify a specialized event. The base *Event* class is shown below:

```
class Event {
      TimeStamp ts_ ;
      String server_ ;
      String service_ ;
      String msg_ ;
   public:
      Event(TimeStamp& t, String server, String service,
           String msg) ;
      Event(const String& buf) ;
      String serialize() ;
}
```

Apart from the constructor, the user needs to implement a function to serializes these classes into a buffer. This function is called by the ADL runtime when the event has to be sent over the network to the DE.

The runtime in addition to these classes also provides utility functions such as *sendEvent*, sends the event to the DE, this function buffers the events and sends them to the DE when the buffer is full or when the timeout occurs. The user can also specify other utility functions.

3.2 Agent Manager

The basic functionality of the AM is to link and load these AG libraries. After loading they initialize each of these AG libraries based on the configuration are specified in a *conf* file. The configuration file maps the AG library to the input source and the port on which the agents would accept connections from the detection system. The configuration file also specifies the mechanism by which the Perl script is invoked, i.e., whether they are invoked periodically or on modification of input, whether the input should be scanned from a fixed point or incrementally and how often to send the extracted events. These parameters are passed to the initialization function of the AG libraries.

3.3 Detection Engine

The detection engine (DE) is the core component of our system. The SMSL specifications are compiled into these detection engines. These DE's are dynamically linked library. The DM links and loads these DE's and delivers events received from servers.

The DE manages multiple Finite State Machine (FSM). The FSM comprises of the *state* and set of *state variables*. The *state* of the FSM indicate their position in the EFSA and the *state variables* remember values as the FSM makes transitions in the EFSA. The DE stores these FSM's in a hash table, with the key based on the mapping construct, as described in Section 2.3.2.

The DE has no FSM's to start with. As an event is delivered to the DE the DE lookups the hash table based on the key formed by the event arguments, if no corresponding FSM's exist, one FSM is created with the *state* as the *startstate*. For each of these FSM's, the EFSA is checked to see if there exists a transition on the delivered event, from the current *state* of a FSM. If there exists a transition then the condition involving event arguments and *state variables* is checked. If the condition evaluates to true then, this FSM makes a transition to new state by updating the FSM, based on the actions specified in the rule. This operation is illustrated in the figure 3.1 and 3.2.

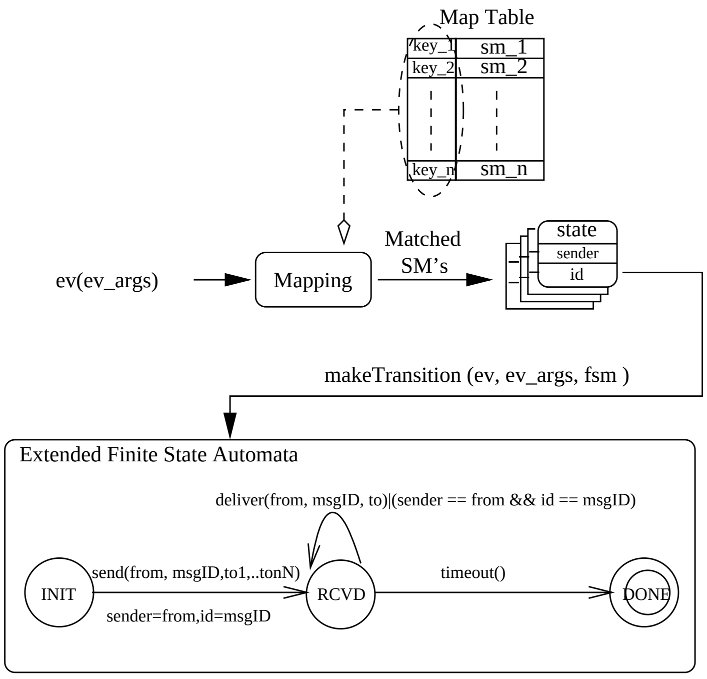

Map Table
key_1
sm_1
key_2
sm_2
key_n
sm_n
ev(ev_args)
Mapping
Matched SM's
state
sender
id
makeTransition (ev, ev_args, fsm)
Extended Finite State Automata
deliver(from, msgID, to)|(sender == from && id == msgID)
send(from, msgID,to1,..tonN)
sender=from,id=msgID
timeout()
INIT
RCVD
DONE

Figure 3.1: Detection Engine Working

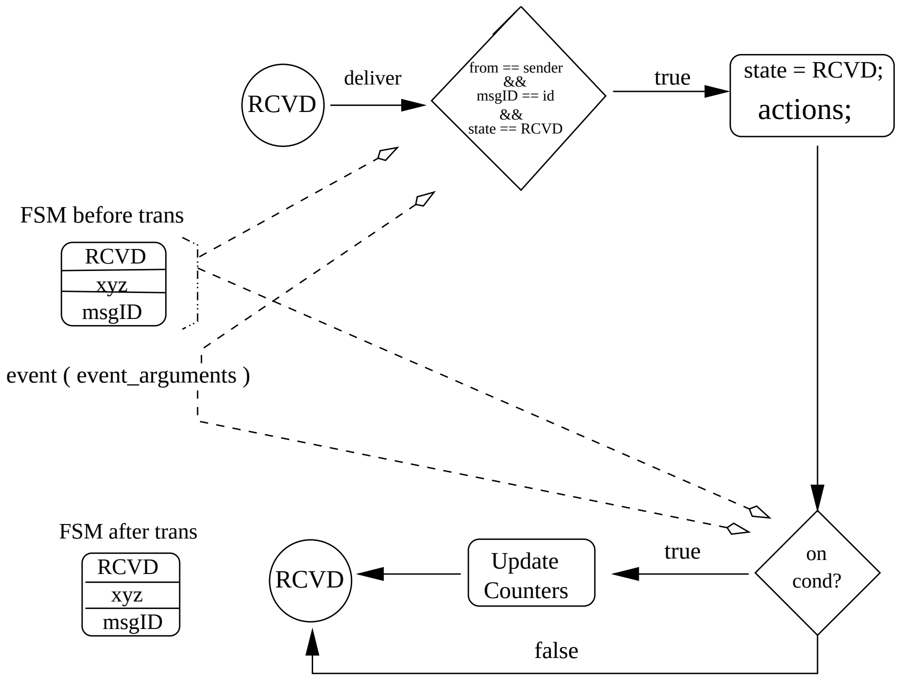


Figure 3.2: Detection Engine Working

In addition, to updating the FSM, the statistical counters of the *on* statements need to be updated. The transition in figure 3.2 presents a high level view of how these on statements are interpreted. We explain this in detail in the section 3.3.1.

3.3.1 Statistical Counters

In this section we explain the interpretation of the *on* statements.

$$\begin{array}{l}\textbf{on all}\\ \textbf{wrt x},\textbf{y size}[\textbf{n}]\\ \textbf{wrt u},\textbf{v size}[\textbf{m}]\\ \textbf{timescale}\ \{\textbf{1},\textbf{2},\textbf{4},...,\textbf{1024}\}\end{array}$$

The figure 3.3 shows the runtime structures that corresponds to the *on* statements in the learning phase. The *on* statement in the example has only three(*on, wrt* and *timescale*) of the five constructs mentioned in section 2.4.2. The on clause indicates that this structure is shared by all transitions and each transition in the specification should update this structure. The first of the wrt clause is represented by a hash table with the size n and the state variables x, y as the key. The value points to another hash table, that represents the second wrt clause. This hash table has a size m and the state variables u, v as the key. The value points to frequency distributions for each timescale based on the timescale clause. The only difference in the structure between the learning and detection phase is that, instead of the frequency distributions we would employ simple counters for each timescale. The hash table is little different from the conventional hash tables. The number of entries in this table is fixed, based on the specified size, this helps us purge *stale* entries.. Thus we need to keep a check on the size at runtime. So, we have implemented a *Most Frequently Used Table* (MFUTable). This table is similar to the hash table as each entry has a key and value. In addition each entry also has a count. When the number of entires in the table has exceeded the specified size we purge elements from them based on this count. The smallest 30% of the entries are removed. This is done so that the removal operation is amortized.

The frequency distribution for each timescale is implemented using counters. We need as many counters as the number of bins, typically 10 bins. Each frequency distribution has a windowed counter, the windowed counter is very similar to the counters used in detection phase as described in the section 2.4.3.

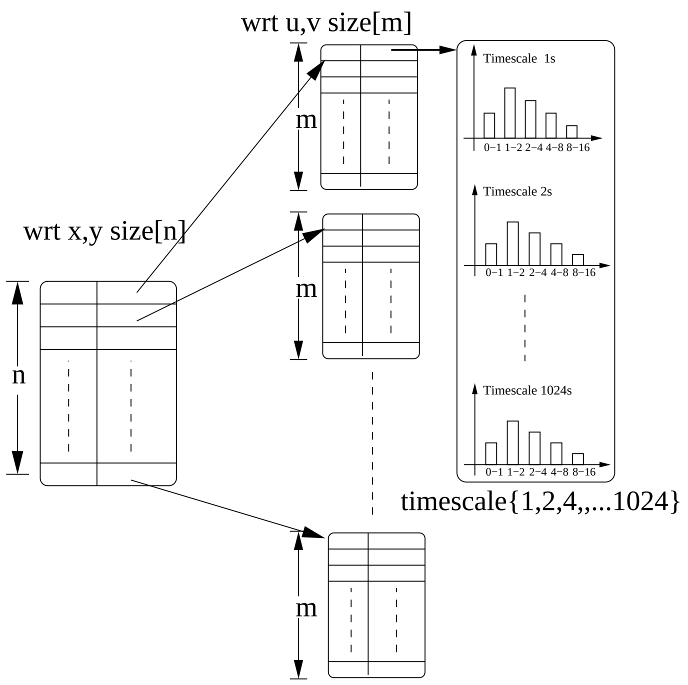


Figure 3.3: Statistical Counters

3.4 Detection Manager

The functionality of the DM is very similar to that AM. The DM manages and initializes multiple DE's. The basic functionality of a DM is to map a set of AG's to one DE. An AG is refereed by the system it is running on and the port it is listening on. There is a configuration file that instructs each DM to load a set of DE's and associate a DE with corresponding AG's.

The DM reads the configuration file, it loads the library(DE) and attempts to connect to the corresponding AG's. This is repeated for each DE in the configuration file. Then the DM waits on these connections for events. When any event is received, the DM multiplexes them to the corresponding DE.

3.5 Deployment

We illustrate how these ADL specifications and SMSL specifications are compiled into agents and detection engines respectively.

Once the administrators comes up with specification policies for the services and systems that they intend to monitor, they can be compiled into agents and detection engines. The effectiveness of the entire system depends directly on the comprehensiveness of the specifications. Such comprehensive policies need a good understanding of the service and systems. We don't discuss any methodology to write specifications.

3.5.1 Compiling a Agents

The ADL specifications are compiled into an agents as shown below

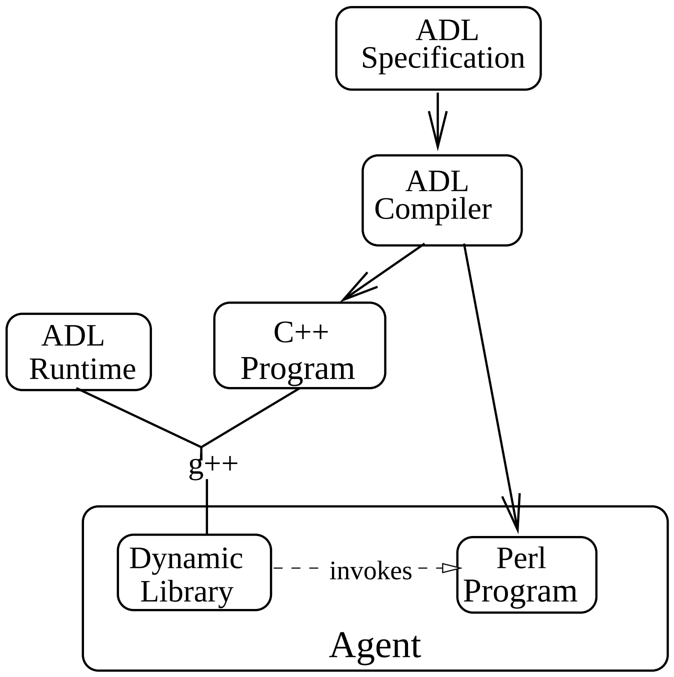


Figure 3.4: Compilation of Agents

Once the agents have been obtained, the configuration of the the agent manager has to be modified to map the agent with the corresponding input. A syntax of the configuration file is as follows:

<lib> *<i/p>* *<port>* {*I*|*F* *<pos>*} {*M*|*P* *<read_time>*} [< *send_time*>]

lib : agent library.
i/p : source of data for the agent.
port : port that agent will wait for connection.
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 sent.

For each agent the agent manager manages there is an entry in the configuration file.

3.5.2 Compiling a Detection Engine

The SMSL specifications are compiled into the detection engines as shown below

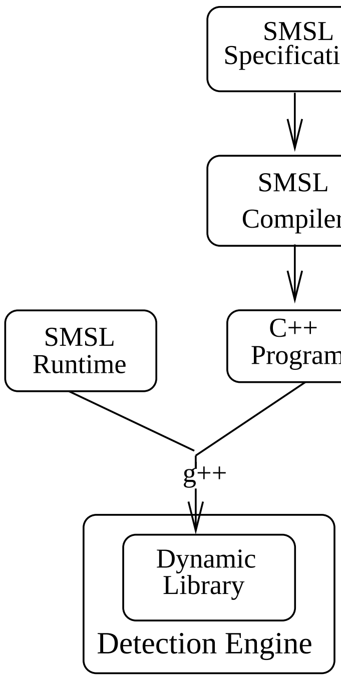


Figure 3.5: Compilation of Detection Engine

These detection engines then have to be mapped to the agents in the configuration of the DM. The syntax of the configuration file is as follows:

$<lib>\ \{<server>,\ <port>\}+$

lib	: detection engine library.
$\{server, port\}+$	: list of agents (server, port combination identifies an agent) that feed the detection engine.

3.5.3 Guidelines

We intend to implement a framework that is flexible and has no rigid architectural constraints. However, there are a few association primitives that need to understood before we could deploy the setup. These are:

- Each agent is associated with one data source (application/service). Patterns are extracted from the data source and converted into events.

- Each agent is associated with one detection engine. The events extracted from the agent are sent over to one detection engine.

- Each detection engine can be associated with multiple agents. These agents can all belong to one system or can be spread over the network. Thus a detection engines can monitor one system, which includes various the services the system offers or one kind of service across the entire network.

- Each detection manager can have multiple detection engines. These detection engines are independent of each other.

- Each Network can have multiple detection managers. These detection managers are independent of each other.

The figure 3.6 is an example to explain the above primitives. AM_1, AM_2 and AM_3 are agent managers, running on systems S_1, S_2 and S_3 respectively. AG^A, AG^B, AG^C and AG^D are agents for services A, B, C and D respectively. DM_1 and DM_2 are detection managers running on systems S_4 and S_5. $DE^{A,B}, DE^C, DE^D$ and DE^{S_3} are detection engines. $DE^{A,B}$ is monitoring A and B on S_1, S_2 and S_3. DE^C is monitoring C on S_1. DE^D is monitoring D on S_2 and S_3. DE^{S_3} is monitoring S_3.Note in the figure 3.6, system S_3 has two of each of the services A, B and D as A and B is monitored by $DE^{A,B}$ and DE^{S_3} and D is monitored by DE^D and DE^{S_3}.

With these primitives in mind, the administrators can decide on a setup that suites them best.

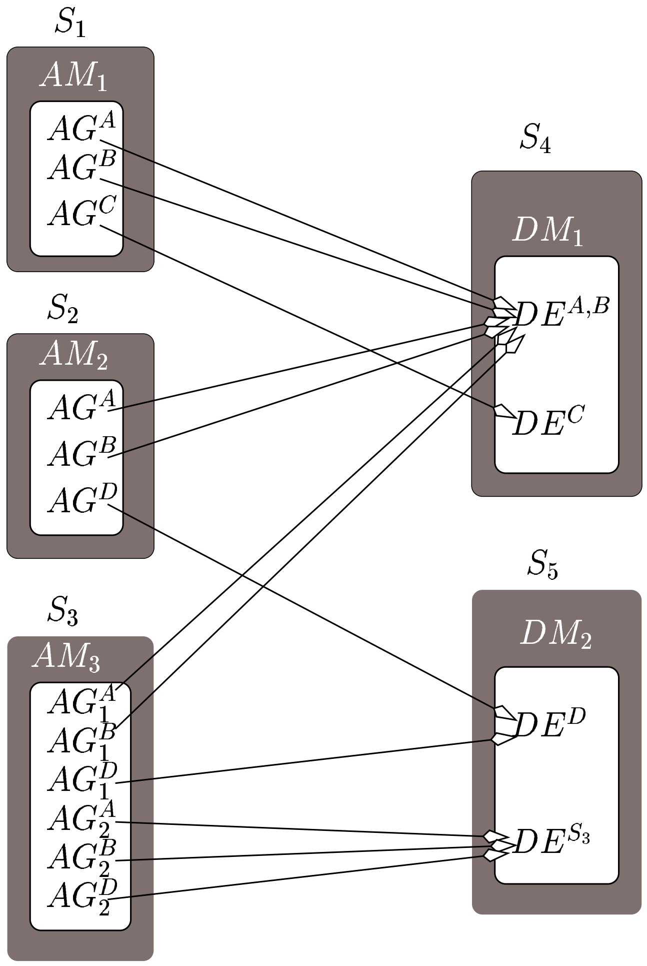


Figure 3.6: An example setup

Chapter 4

Related Work

Most of the approaches for system and network management, have a primarily goal of detecting faults or intrusions. We can broadly classify these approaches based on their architecture either host-based or network-based.

4.1 Host Based Approach

Swatch [6], the Simple Watch Daemon is used to monitor log files. When it sees a line matching a pattern you specify, it can highlight it and print it out, or run external programs to notify administrators through mail or some other means. It is a set Perl scripts that read from the .swatchrc file for patterns and actions.

Syslog Heuristic Analysis and Response Program (SHARP) [2] improves the monitoring of systems by extending the existing syslog infrastructure with programmable modules. These modules use a library with a simple API to perform analysis based on the messages they register to receive. They use the modified syslogd, *nsyslog* [4] for network message delivery. *nsyslogd* communicates all system log to *sharpd* by means of a UNIX domain socket. *sharpd* passes them to the interested module. The modules are written in C. These modules need to register with the sharp deamon. The need to specify the service they are interested in. Based on this the sharpd will direct the logs that go through syslogd to the interested modules.

Swatch and SHARP works only with one message line at a time. LogSurfer [8] overcomes the limitations of these earlier approaches, especially the limitation on single lines. Uses contexts by collecting messages instead single lines. Matching of lines is done by two regular expression. A log line must match the first expression but must not match the optional second regular expression, i.e., one can specify exceptions.

These approaches expect to administrators to classify the patterns that indicate bugs or faults and thus cannot detect unknown faults. In addition, they lack the ability to correlate activity across services on one host. They base their detection on activities logged in one log file.

4.2 Network Based Approach

RedAlert [10] is a application monitoring system which consists of a stateful server daemon and extensible Perl client API. IP-protocol service is a candidate for RedAlert monitoring: the clients determine what error condition they have discovered, convert that information into a standard message format, and transmit the alert to the server. The major drawback of this approach is that the administrators should possess a clear understanding of the working of all applications that need to be monitored.

Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) [9] is a tool suite for tracking malicious activity through networks. It uses a blocked approach to network surveillance, attack isolation and automated response. They use a layered approach. At the lowest layer are service monitors that interact with the environment, correlate the information and can disseminate to other EMERALD monitors through subscription based communication scheme.

Autonomous Agents For Intrusion Detection (AAFID) [1] [5] consists of agents distributed across the network, working collectively to detect intruder-like activity. The agents monitor for interesting events occurring in the host, they report their findings to a single transceiver, these are per-host entities. These can perform data reduction and report to monitors. These monitors oversee the operations of several transceivers, so they can correlate network wide activities.

Both EMERALD and AAFID approaches propose an architecture for intrusion detection systems. They don't deal with any specific mechanism for detection.

Chapter 5

Summary

5.1 Conclusions

In this thesis, we use high-level specifications to describe the monitoring and detection policies. These specifications are intended to capture behavior of services an systems. Deviations from the learned behavior indicate faults. Thus, faults can be detected even though they may not have been encountered previously. In addition it can also detect well known faults.

We notice that there are some signs to every failure in the network. Careful monitoring can detect these signs that indicate deviations from the expected behavior in real time. Thus our approach gives us the ability to detect problems before they cause damage and can thus be preventive.

We designed a high-level language called the Agent Development Language used to develop agents. This language has a very intuitive syntax, making it easy to use. The ADL specifications are compiled into a Perl pattern matching program and C++ program that manages the Perl program.

We also designed another high-level language called State Machine Language to specify state machines. This language is powerful enough to express a range of behaviors over time. Our compiler will translate state machine specifications into an Extended Finite-State Automaton (EFSA). These EFSA are represented by optimized C++ programs. This language is intended to simplify the specification of state machine. At the same time, be able to gather comprehensive statistics, that can detect deviations efficiently with a low false rates. These feature greatly simplifies the administrator's work.

5.2 Future Work

Currently we have the framework in place. But we haven't deployed and tested the system yet. So we intend to work on deploying this framework by building agents to monitor a services in our lab, such as httpd, ftpd, system logs across all machine in our network. Also we will employ agents for system status that will monitor memory, CPU, HDD usage. We are hopeful that this framework will prove to be useful.

In addition, currently the only kind of response action the detection system supports is notification. We are looking at providing some kind of mechanism so that the detection manager can instruct the agents to take some response action, such as stopping, restarting the service.

The development of the specifications even with such intuitive languages might not be completely acceptable to administrators. So we want provide a methodology that can be adopted in writing such specification. We also can develop an expert system that employs these methodologies to help the user write the specification. We may also develop a graphic interface to the user that helps manage and modify the entire system visually.

Bibliography

[1] J. S. Balasubramaniyan, J. O. Garcia-Fernandez, D. Isacoff, Eugene H. Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. In *ACSAC*, pages 13–24, 1998.

[2] Matthew Bing and Carl Erickson. Extending unix system logging with sharp. *Proceedings of the Seventh Systems Administration Conference (LISA XIV) (USENIX Association: New Orleans, LA)*, pages 101–108, 2000.

[3] R. Bowen, D. Chee, M. Segal, R. Sekar, P. Uppuluri, and T. Shanbag. Building survivable systems: An integrated approach based on intrusion detection and confinement, 2000.

[4] Darren. nsyslogd.

[5] R. Gopalakrishna. A framework for distributed intrusion detection using interest driven cooperating agents, 2001.

[6] S. E. Hansen and E. T. Atkins. Automated system monitoring and notification with swatch. *Proceedings of the Seventh Systems Administration Conference (LISA VII) (USENIX Association: Berkeley, CA)*, page 145, 1993.

[7] Anita K. Jones and Robert S. Sielken. Computer system intrusion detection: A survey. Technical report, University of Virginia Computer Science Department, 1999.

[8] Wolfgang Ley and Uwe Ellerman. Logsurfer, 2000.

[9] P. A. Porras and P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *Proc. 20th NIST-NCSC National Information Systems Security Conference*, pages 353–365, 1997.

[10] Eric Sorenson and Strata Rose Chalup. Redalert: A scalable system for application monitoring. *Proceedings of the Seventh Systems Administration Conference (LISA XIII) (USENIX Association: New Orleans, LA)*, pages 21–34, 1999.

[11] Larry Wall and Mike Loukides. *Programming Perl.* O'Reilly & Associates, Inc., 2000.