EXTENDED AUTOMATA-BASED APPROACHES TO INTRUSION DETECTION

By

Joel W. Branch

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Rensselaer Polytechnic Institute
Troy, New York

March 2003
(For Graduation May 2003)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In this work, I describe two new approaches to providing computer security by way of intrusion detection. Although the proposed methods find different applications, they both employ extended uses of finite-state automata. The first method lends itself to the real-time detection of denial of service computer attacks using time-dependent deterministic finite automata. Some current network-based intrusion detection systems employ state-transition based methods as a primary mean to detecting system penetrations and misuse. However, this work utilizes the time intervals between certain event occurrences (as defined in our automaton) to improve the accuracy of detecting specific denial of service attacks. The proposed design also lends itself to a distributed detection architecture, supporting non-obtrusive attack signature updates as well as hardware and software platform portability.

The second method is used for the detection of masquerading, a computer attack in which an intruder pretends to be a legitimate user. This approach records and processes sequences of user commands, building probabilistic state finite automata representing users' signatures. The signature is used to distinguish between legitimate and invalid user sessions. The novelty of this approach is the use of transition probabilities in traversing the signature automata. Additionally, a temporal-based method is used to construct more precise ”variable length” signature traces. In contrast, current computer intrusion detection systems use state-transitions to detect compromised program states.

CHAPTER 1
Introduction and Historical Review

A growing rate of incidents of computer system break-ins have increased the importance of information assurance and electronic security. The recent threat of both foreign and domestic terrorist attacks on vital computer systems has called for accelerated research and development in securing both commercial and government system vulnerabilities [1]. One of the various solutions that address these concerns is intrusion detection–a software system that discovers inappropriate or anomalous activity on a computer system.

Traditionally, there have been two main ways to detecting intrusions: anomaly detection and penetration identification. The former method attempts to differentiate "anomalous" activity from the established normal operating behavior of a computer system, application, or user. The advantage of this model is that it allows "unknown" behavior to be identified, so that newer attacks may be caught. A common disadvantage of this method is that sometimes, legitimate behavior is interpretted as being anomalous. Penetration identification (often referred to as misuse detection) makes use of an attack "signature." After a signature is defined that identifies a manifestation of an attack, the attack can be discovered in the monitored network traffic (or host-based audit trails). The advantage with this method is that well-known attacks can be identified in an easier and quicker manner. However, attacks (or behavior) for which there is no defined signature may occur undetected.

1.1 Network-Based Intrusion Detection Systems

Traditionally, there have been two main classes of intrusion detection systems: network-based and host-based. The former type usually monitors the detailed traffic activity of a network of computers and other devices (i.e., routers and gateways). To this end, raw network packet data (usually collected by a utility such as Tcpdump) is analyzed for signs of attacks such as "code-red" or "denial of service." We add that while a network-based system monitors network traffic, it does not necessarily have to monitor an entire network of devices. A network-based IDS may be deployed on a single host to monitor just the network activity of that particular machine. While there have been many different implementations of network-based systems, most are found implemented as penetration identification systems.

1.1.1 Past works

Due to the increasing complexity of computer networks as well as the growing sophistication of network-based attacks, network-based intrusion detection has recently gained more attention from academic, military, and commercial sectors. Consequently, various IDSs have been implemented that address different aspects of network security and use different methods of detection. For example, some tools employ artificial neural networks, trained for misuse detection across a network [7, 2]. Another system, GrIDS, constructs activity graphs from network traffic data to detect large-scale automated attacks in real-time [29]. EMERALD uses a scalable distribution of surveillance monitors throughout a network to apply distributed event correlation models in detecting network intrusions [25]. In [32], the authors describe how DOORS (Distributed Online Object Repositories) can be used to distribute the functions of an intrusion detection system via mobile agents to various selected areas of a network. In turn, the DOORs work spawned [22], which addresses the need for securing mobile agents, further enhancing the deployment of security mechanisms in a network.

The network-based system we propose shares methodologies similar to those of a couple of other IDSs. For instance, Snort is a lightweight network-based IDS that utilizes a rules-based approach, along with network packet-sniffing and logging to perform content pattern matching and detect a variety of attacks [26]. In comparison, we also use a rules-based approach in detecting attacks, but we apply it in the form of a deterministic finite automaton as opposed to the rule chains used in Snort. Another tool comparable to ours is NetSTAT. NetSTAT is an extension of the original STAT design which models an attack as a sequence of actions that progressively take a computer from an initial normal state to a compromised state [14, 33]. NetSTAT applies that design to a networked environment by modeling both the guarded network and the attacks. Based on this model, it determines which network events have to be monitored and where in the network they should be monitored. In our project, we choose to bypass modeling the network and instead focus directly on the representation of various denial of service attacks.

1.2 Host-Based Intrusion Detection Systems

A host-based IDS monitors the detailed activity of a particular host in real-time. The system call traces produced by an auditing mechanism such as the Solaris Basic Security Module (BSM) typically provides the IDS with the data needed to search for attack signatures [15]. When an analysis of the BSM data shows signs of an intrusion, the IDS alerts the system administrator of an attack. Host-based detection systems can detect attacks that network-based detection cannot see. For example, attacks that are made at the keyboard of a critical machine do not travel across the network. More information, including machine and process state, are available for host-based methods enabling coordination of multiple data sources in detection.

1.2.1 Past works

Host-based intrusion detection has emerged as a very important part of the information assurance and security areas, resulting in development of several research and commercial tools. STAT (State Transition Analysis Tool) is a misuse detection system that uses attack signatures to recognize intrusions [14]. Intrusions are represented as a series of state changes from a secure to compromised state. The system uses audit data as evidence of state transitions, causing an alert if a compromised state is reached. The system manager must create state diagrams representing the attacks, so the authors actually point out that their system is very inefficient in detecting masqueraders, which is the focus of the IDS outlined in this work.

The Reliable Software Technologies Corporation designed tools to profile the behavior of applications. They use techniques ranging from memorization to generalization [13]. The former technique, rooted in anomaly detection, uses a simple equality matching algorithm to flag anomalous behavior. Sequences of BSM events are captured online and compared against those stored in a database built from the normal program behavior profile. If the monitored sequence is not found in the database, then an anomaly counter is incremented, and, after a threshold value is exceeded, an alarm is raised.

A seminal research paper on anomaly-based masquerade detection was written by

Forrest et al [11]. In this work, the authors model the "normal" behavior of *sendmail* and *lpr* at the system call level. They use a sliding window approach to record what system calls follow each other. Once a database of these sequences is formed, new traces are collected, using the same sliding window, and checked against the database. Results are expressed as a ratio of found mismatches to all possible mismatches. A popular product implementing some of Forrest's ideas is ImSafe, an anomaly detection tool that monitors the system call traces produced by specific applications. It tries to predict the next system call [10] based on an application profile created during the learning phase. My approach is similar, except that user behavior is modeled instead of application behavior.

In [8], the authors present a method of masquerade detection based on the Bayesian model to test a hypothesis that the observed command transition probabilities come from a profiled transition matrix. In [20], the authors propose the use of the Naive Bayesian classification algorithm and its variant with a special updating feature. Their system computes the probability of each command appearance in the input of a particular user, based on the frequency with which that command was seen in the training data. The authors also compare the performance of their system and the one presented in [27], using the same dataset.

1.3 Deterministic Finite Automata

Conceptually, a deterministic finite automaton (DFA) is an abstract computational model designed to represent an idealized computer. Just as a computer changes states of operation and produces some outputs given particular inputs, so does a DFA. Specifically, DFAs are designed to recognize or accept member strings of a particular regular language [28]. In this case, we want to recognize the language of DoS attacks (to be explained in Section 2.3.2). DFAs possess two important properties: they embody a finite number of states and, they are deterministic, meaning that given a current state and an input, the automaton "transitions" to only one state (which could be the same state or a new one).

Formally, a DFA is defined by:

1. A finite set of input characters (its alphabet)

2. A finite set of states, with one state defined as initial state and a subset of states distinguished as final states

3. A set of transitions that is Cartesian product of “set of states” times “alphabet” times “subset of states”

Hence, in referring to the third component of the definition, each transition is a triple: originating state, transition character, and target state. This represents a move to the target state that the DFA undertakes when it receives the transition character while in the originating state. Informally, DFAs are represented by state-transition diagrams. Circles represent the automaton’s different states and unidirectional arrows labeled with input characters represent transitions between different states triggered by the specific character in the input. Typically, final states, in which the entirety of an input string is accepted, are double-circled. Also, according to the formal definition, for every state and every input character, there must exist an exiting transition. Fig. 1.1 is an illustration of a simple DFA which accepts strings from the alphabet {a, b} containing the sub-string “aba.”

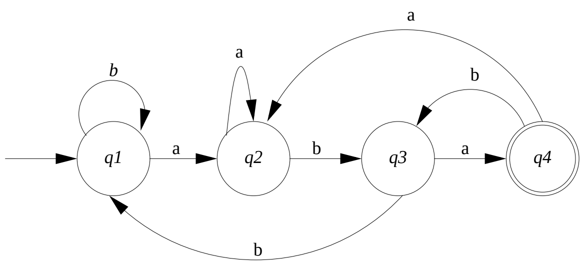


Figure 1.1: An example DFA

In this example, q1 is the start state and q4 is the final state. If the DFA is in state q1 and receives “a” as a current input character, then the DFA moves to state q2. If in state q2 the DFA receives “b” as an input, it moves to state q3. If an “a” is received by the DFA in q3, the automaton moves to the final state and the input string is accepted. An examination of Fig. 1.1 shows that any input disrupting the pattern “aba” moves the DFA back to a state where it can continue to monitor for the desired substring. While the DFA in Fig. 1.1 has only one final state, this is not a requirement for DFAs in general; a DFA can have more than one final state.

CHAPTER 2
Denial of Service Intrusion Detection Using Time-Dependent Deterministic Finite Automata

2.1 Introduction

The approach described in this chapter focuses on detecting one type of computer attack: denial of service (DoS). Generally, DoS attacks are classified as a characteristic set of events (i.e., an intentional flood of unwanted traffic) that diminishes or elimates a network's capacity to perform its expected function. As a result, the compromised system looses the ability to handle legitimate traffic/requests, experiences a loss in available link bandwidth, and is overall unable to provide normal service to its users. In many cases, service is denied until the attacker's address is discovered so that further traffic from that source can be blocked. However, often the source address is spoofed so that the true origin of an attack becomes untraceable. A more difficult scenario arises in cases of the distributed denial of service (DDoS) attacks, in which multiple clients are coordinated to simultaneously flood a victim machine or network with requests. In this case, more resources on the targeted system could be in jeopardy and more clients need to be blocked to recover from an attack. As with other attacks, it is most effective to detect DoS (and DDoS) attacks as early as possible.

I propose a novel method to the detection of DoS attacks primarily by exploiting the time-dependent characteristics of DoS attack behavior. Both this method and framework contribute a computationally efficient, non-obtrusive and scalable solution addressing an important subset of network security. In this chapter, we will first discuss the specialized automata-based approach to penetration identification, as well as the motivation for its use. Following that section will be an overview of the system's architecture. We will then present an evaluation of the system's test results. In keeping with the standard measure of testing most other intrusion detection systems, the publically available datasets from both the 1998 and 1999 Defense Advanced Research Projects Agency (DARPA) intrusion detection evaluations [18, 19] were used for testing. Finally, we will conclude with a brief discussion of the project's future applications.

2.2 Time-Dependent Deterministic Finite Automata Approach

2.2.1 Time-Dependent deterministic finite automata

The time-dependent deterministic finite automaton (TDFA) is an extension of the DFA. This extension enables it to consider more than just the sequence of input characters; TDFAs also consider the time intervals between receiving input characters in recognizing members of a language. This becomes very beneficial in the use of automata to recognize DoS attack signatures since many DoS attacks are dependent upon the time intervals between arriving network packets. Fig. 2.1 shows an example TDFA. We can think of this machine as recognizing the pattern "$a, b < 5, a < 5$." In other words, the "b" must occur within five seconds of the initial "a", and the last "a" must occur within five seconds of the "b". All transitions shown without the five second time restraint are default transitions. If the desired input arrives, but not within the specified time constraint, then the default transition for the desired input is used. Notice that in our example (Fig. 2.1), the transition from q1 to q2 is a default transition even though it takes the TDFA closer to the final state. This is because the currently received "a" may be the first character of the attack pattern; it does not make sense for it to come within five seconds of some other input.

A TDFA, as defined here, is a non-trivial extension of DFAs, because the input to the TDFA is a pair consisting of a character from the input alphabet and a time of this character arrival (which we currently represent as a real number). The Cartesian product of these two elements of the TDFA input yields an infinite alphabet, so a TDFA cannot be represented as a product of two DFAs. However, a TDFA uses the actual difference of arrival times between two input characters (this still leads to an infinite set of possible differences). Furthermore, TDFAs use these differences to create boolean values resulting from the comparison of each difference to constant or computed variables defined in the TDFA, which finally yield the finite input to our automata. Hence, formally, a TDFA can be represented as a combination of a token arrival time preprocessor and an ordinary DFA. The token arrival time preprocessor simply yields a boolean value to each inequality defined in transitions exiting from the current state of the TDFA. The states of this particular DFA are the same as those of the TDFA. Each transition in the TDFA with a time constraint is represented by the DFA transition labeled with the same character and a true boolean value. Each transition in the TDFA without a time constraint is represented

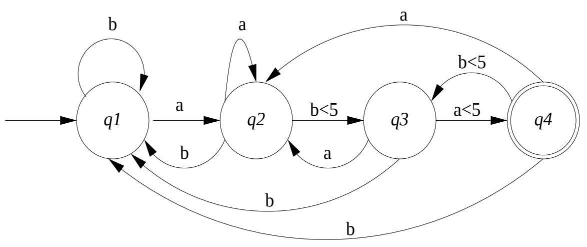


Figure 2.1: An example TDFA

by two transitions in the DFA, one with true and the other with false boolean values.

2.2.2 Denial of service attack representation using time-dependent deterministic finite automata

The very nature of TDFA models makes them a logical choice in representing DoS attacks. DoS attacks manifest themselves as a characteristic series of network events or special packets that progressively render a particular resource inoperable. Therefore, the transitional arcs of TDFAs are used to represent those characteristic attack events. TDFA states are used to represent incremental conditions of a system as it reaches a state of fully successful penetration. The final state(s) of a TDFA then represent state of attack completion.

Serving as another motivation, I observed that the time scale of the various DoS attacks differs dramatically. After analyzing a DARPA dataset for a particular week's worth of intrusion data, a histogram of the attack durations over a period of several days (see Fig. 2.2) was derived. In this figure, because of the large variance of the attack durations, the x-axis is an approximate logarithmic scale of seconds while the y-axis represents the duration count.

While many of the attacks take place over a very small time period, a significant number of attacks take over a minute and quite a few take several times longer than that. DoS detection systems must cope successfully with this wide range of time scales. Incorporating a time component into an automaton structure grants the ability to use temporal differences for accurate detection.

While detection accuracy has traditionally served as a common benchmark for IDSs, the possible circumstances of detection "inaccuracy" should also merit the atten-

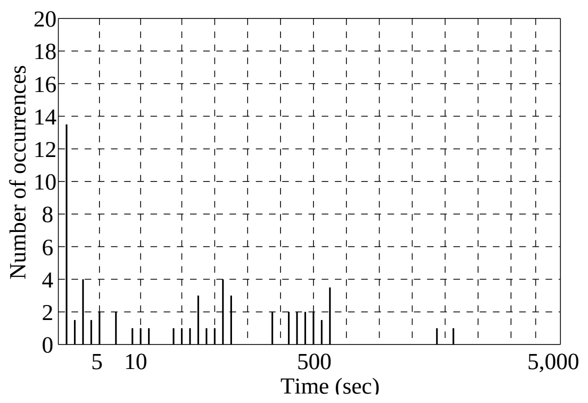


Figure 2.2: Histogram of attack durations from 10 days of attack data

tion of the intrusion detection community. A recent study showed that on average, various commercial IDSs repeatedly "crashed" under the burden of reporting too many false alarms from an ISP's network traffic–meaning that frequent amounts of either low priority or false alarms can cause an IDS to totally miss significant intrusions for which signatures are defined [24]. This definitely serves as motivation for the use of more intelligent attack signature models that intrisically respond to only the events and variations defined by the author of the signature.

Another problem common with most IDSs is the lack of the ability to perform detection on high-speed links. This makes most network-based IDSs situated only for detection coverage on an individual node rather than across an entire network. This concern is being addressed by the development of more intrusion detection appliances (rack-mountable hardware devices) designed to reliably inspect high-speed IP traffic [23]. Even with the advent of more robust hardware, poorly designed inspection algorithms can still throttle detection performance, leaving software as the bottleneck. Because automata-based models are traditioanlly efficient in computation, the proposed detection method should complement intrusion detection appliances used for high-speed network traffic.

2.3 System Overview

In this section, we discuss the full architecture of the system, including its prime functions and its various subsystems and their interactions. Before continuing, it would be useful to note that the system detects DoS attacks occurring in TCP, UDP and ICMP

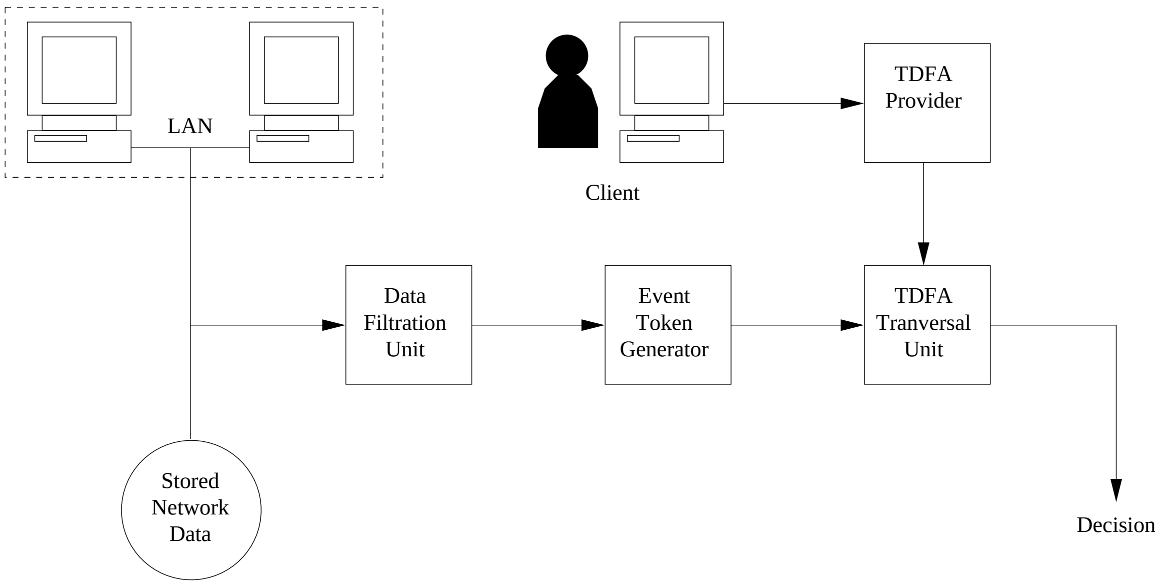


Figure 2.3: Overview of system architecture

network traffic. These are all industry-standard network communication protocols [30]. The following are the key characteristics of the proposed IDS:

- Detects DoS attacks from both real-time and historical data
- Uses time-dependent deterministic finite automata to model and confirm attacks
- Supports the updating of attack models without interruption to other system components

Having an option of operating from real-time or historical data offers a couple of advantages for a site security officer (SSO). Real-time monitoring of TCP/UDP/ICMP network traffic provides the best level of protection because ongoing attacks may be prevented. However, with the addition of pre-recorded (historical) datasets, off-line operation still allows the SSO to see if and when attacks might have occurred while the IDS was down for maintenance. Also, the SSO can use experimental TCP/UDP/ICMP datasets to test and tune the system for newer attacks. Next, while confirmation procedures will be discussed in greater detail later, it is important to further note the significance of time-dependent deterministic finite automata in our system here. Earlier, we discussed how

TDFAs serve as an appropriate model for DoS attacks. However, the additional benefit of TDFAs for site security officers is that they permit the storage of both an attack's base signature and its variation(s) using only one model construct. A SSO can even use one TDFA model to represent multiple DoS attacks, as is done in the prototype implementation. Additionally, using time-based information in attack signatures increases the accuracy of detecting DoS attacks. Lastly, the ability to update attack models without disturbing other system components prevents degradation of the system's performance during detection. As will be discussed later, this feature also presents the opportunity for a distributed architecture.

Also worthy of noting is our selected development language, Java v1.2. We chose this particular language because of its coveted mantra, "write once, run anywhere." Although the prototype was written for Solaris, it can easily be used on any operating system with a Java virtual machine installed.

At present, four distinct components make up the IDS:

1. The data filtration unit
2. The event token generator
3. The TDFA transversal unit
4. The TDFA provider

External components with which the IDS interacts include a local area network (LAN), stored (historical) network traffic data, and a SSO client machine. The connectivity of the entire system is presented in Fig. 2.3.

2.3.1 Data filtration unit

Network packets carry a wealth of information (i.e., sequence number, header length, checksum, etc.), some of which may not be needed for the purposes of network-based IDSs. Since the focus is on DoS attacks, the packet data fields in which we are interested include those pertaining to such information as source and destination addresses and the various flag fields (i.e., SYN and ACK). The function of the data filtration unit (DFU) is to process relevant network packet information for subsequent components of the system

Table 2.1: Current parsed tcpdump fields

Packet type	Timestamp
Source IP address	SYN flag
Destination IP address	ACK flag
Destination port	Echo request
More fragments flag	Echo reply
Fragment offset	Sequence first
IP length	Identification

(according to the IDS flow of data). As seen in Fig. 2.3, network traffic data originates from either of two sources: a local area network (LAN) or a stored data source. It was mentioned before how the Solaris BSM utility supplies audit data for host-based IDSs. However, the utility module that we use here is tcpdmp [16], which can provide a record of network activity for a particular machine in ASCII text form, delimited into various fields. The DFU can be manually configured to parse whatever tcpdump fields the user requests. Table 2.1 lists the fields that were chose for use in the prototype.

Live data originating from connected LAN devices are used in real-time detection while stored data, our second source of extraction, is used in offline mode. As opposed to the prior source, stored data usually resides in a log file and may be in either ASCII or binary form. If the data filtration unit receives the process packet information in binary form, it first converts it to ASCII text for further processing. In this case, the associated overhead of converting binary data into ASCII form constitutes an irrelevant performance hit since we are most concerned with penetration threats present in live, not stored, network traffic. Regardless, the end product is a delimited ASCII text message which contains specific network event information (including TCP, UDP and ICMP packet data).

2.3.2 Event token generator

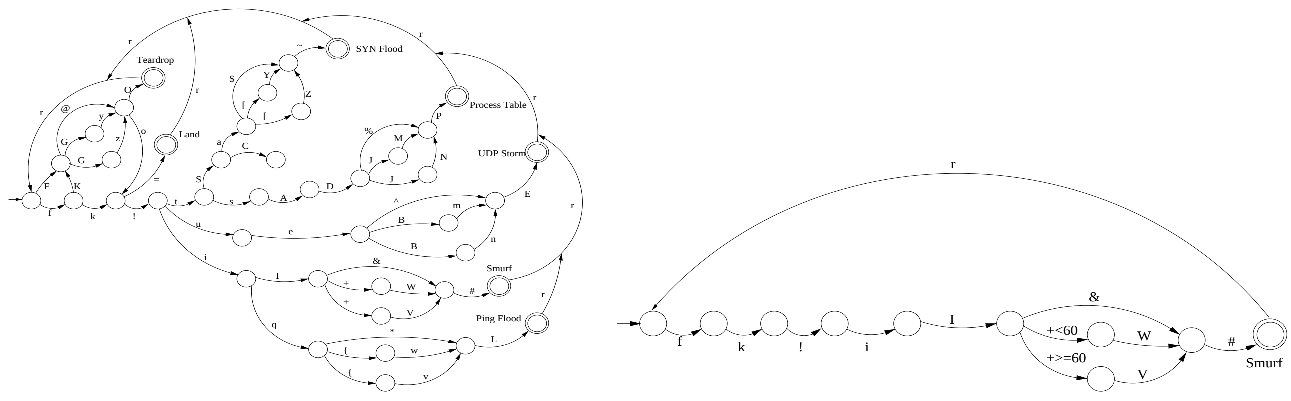
After network event data is processed by the DFU, the corresponding ASCII text (remaining in a delimited format) serves as input for the event token generator (ETG). The ETG is then responsible for translating the DFU text messages, each representing a particular network event, into special tokens. We must note that a relationship between DFU messages and ETG tokens is not necessarily "one-to-one." It is quite possible that the

Table 2.2: Sample ETG tokens and their definitions

Token(s)	Definition
S	Packet's SYN flag is checked
F	Packet's MF flag is checked
$J>5$	Non-initial ACK packet; time interval between this and current packet is greater than 5 seconds
&	First ICMP echo reply packet to a particular destination [address and port]
$B<60$	Not-initial UDP packet to echo port of particular destination; time difference between this and previous is less than or equal to 60 seconds

information in one message will cause the ETG to generate a sequence of tokens. Also, only a finite amount of ETG tokens was defined, each corresponding to network events we observed as being characteristic of DoS attacks. Therefore, many DFU text messages might be ignored by the ETG. In re-addressing the DFU, we are currently researching efficient methods of filtration so as to not only parse out relevant packet information, but also determine which events to send altogether–reducing message-passing overhead between components. Moving on, all of the predefined ETG tokens, each being a string of one or more ASCII characters, compose an alphabet used by our system for recognizing DoS attacks. The efficiency of domain-independent IDS languages has been shown with such languages as STATL [9] (used by both USTAT and NETSTAT), which partially serves as the motivation for using a proprietary language. The other incentive is that it simplifies the definition of our detection engine: the TDFA transversal unit, which is discussed in the next section.

The following is a condensed explanation of how the ETG operates. Suppose that after reading the DFU text message, the ETG determines that the UDP destination port number of the packet is set to an echo port number (a condition indicative of an UDP Storm attack). Subsequently, the ETG generates the token "e" and sends it to the TDFA transversal unit, where it will be used for UDP Storm attack recognition. Table 2.2 highlights just a few of the tokens that compose the language for our system.



(a) Entire TDFA model (b) Smurf attack portion of the TDFA

Figure 2.4: Experimental TDFA model with a detailed view of the smurf attack showing time constraints

2.3.3 TDFA transversal unit

Most intrusion detection systems have some distinguishable core component primarily responsible for recognizing attacks. In this system, the TDFA transversal unit (TTU) acts as the main attack detection engine. The TTU is what actually embodies the user-defined TDFA that represents the various DoS attacks.

The relationship between the TTU and the previous module, the event token generator, is best thought of as that between a physician and patient. The ETG (patient) displays “symptoms” (on behalf of the guarded host) of probable DoS attacks. As discussed earlier, these symptoms materialize themselves as tokens. The intent of the TTU (physician) is to read the tokens and “diagnose” the host as being under, or not under a state of attack. It does this by using the ETG tokens as input characters to traverse the supplied TDFA. In our current prototye, when the TTU finds that its TDFA has reached a final state, it alerts the site security officer that an attack (specified by its respective final state) has occurred.

As mentioned earlier, it would be beneficial for a SSO to verify the effectiveness of the IDS using pre-recorded network traffic data containing traces of successful intrusions. Specifically, this entails verifying the correctness of the TDFA transversal unit. To gain insight as how to design the most effective TDFA models, attack-laced network traffic data provided by the DARPA intrusion detection evaluation was inspected. Since

Table 2.3: Current DoS attacks for which we have written TDFA signatures

Name of attack	Protocol used	Effect
Land	TCP	Operating system loops and eventually freezes
SYN Flood	TCP	Legitimate service requests are denied as CPU resources become totally consumed; operating system may crash or loop
Ping Flood	ICMP	Network slows down; network connectivity may be disabled
Process Table	TCP	Process table is completely filled with network server instantiations; new processes cannot be started
Smurf	ICMP	Host floods both itself and intermediate network with ICMP echo replies
Teardrop	N/A	Host may hang or crash
UDP Storm	UDP	Legitimate service requests are denied as CPU resources become totally consumed; network may become congested

this data would be used for our testing purposes, it was an ideal place to start defining various DoS attack signatures, especially the time-constraints of those signatures. Additionally, we consulted Kendall's work, which contained signature descriptions of the attacks used in the DARPA 1998 evaluation [17]. Also, variations of the attack signatures were included that would likely cover multiple incarnations of certain attacks due to varying time durations or distinct packet characteristics. Again, this highlights the added advantage of using a TDFA to model attack signatures. Fig. 2.4(a) is an abstraction of the TDFA detection model that was used for experimental purposes. Fig. 2.4(b) shows a more detailed portion of the TDFA representing a Smurf attack, complete with time constraints on some of the transitional arcs. Notice the lack of default transitions in the example TDFA. It is noted later how system administrators need only to define "linear" attack models, devoid of default arcs. Table 2.3 lists seven DoS attacks for which the current TDFA was designed to recognize.

2.3.4 TDFA provider

It is mentioned earlier that this system supports the updating of TDFA attack models without interruption to the rest of the system. The TDFA provider makes this feature possible. When a site security officer (client) wants to replace the resident TDFA, it interacts with the TDFA provider and gives it a description of the new TDFA model. However, models such as those depicted in Fig. 1.1 and Fig. 2.1 are not required. A client has only to specify the attack "signature," meaning only those states and transitions that lead directly to attack acceptance (recognition). All other transitions, such as those moving a TDFA back to its start state, would automatically be added by the TDFA provider. This is convenient for the client as he or she needs only to provide a simple, linear attack model. The TDFA provider then supplies the TDFA traversal unit with the user-defined TDFA description. The substitution of the previous attack model by the updated attack model can occur without degradation to other components of the system because the TDFA provider interacts only with the TDFA transversal unit. It should be noted that at start-up time, the TDFA transversal unit contains a default TDFA model, so detection is possible before interjection by the SSO. As will be described with more detail in Section 2.5, this component makes it possible to develop a distributed modular architecture permitting a more automated approach to updating attack signatures.

2.4 Evaluation

As stated before, we used datasets from the 1998 and 1999 DARPA intrusion detection evaluations in testing the system.

In total, we used eight tcpdump training data files: five from 1998 and three from 1999. Currently, the system has been tested on all but two of the seven attacks that were listed earlier (UDP Storm, and Process Table attacks were not present in the data files that we used from the DARPA web site). The results of the test are presented in Table 2.4. The third column shows the time of the specified DoS attack, according to the DARPA dataset. The fourth column shows when our system recognizes the timestamp of the packet finalizing the specified DoS attack. As is seen in Table 2.4, we were very accurate in detecting various DoS attacks. However, in one dataset, 1998_week6_fri, the system did fail to detect the SYN Flood attack. The attack was successfully detected in most other datasets,

Table 2.4: Test results from DoS attacks in 1998 and 1999 DARPA datasets

Dataset	Attack	DARPA Time	TDFA Time
1998_week4_tues	SYN Flood	11:55:38	8:50:15
	Ping Flood	20:11:31	20:11:31
	Teardrop	23:15:08	23:15:08
1998_week5_mon	Teardrop	08:15:02	8:15:02
	Smurf	12:53:15	12:53:15
	Smurf	15:33:28	15:33:28
1998_week5_fri	SYN Flood	17:27:07	17:27:07
	Smurf	18:00:15	18:00:17
1998_week6_tues	Ping Flood	13:04:56	13:04:56
	Land	17:53:49	17:53:49
1998_week6_fri	Teardrop	08:32:12	8:32:12
	SYN Flood	09:31:52	NO
	Smurf	19:12:37	19:16:27
1999_week2_mon	Ping Flood	08:50:15	8:50:15
	Land	15:57:15	15:57:15
1999_week2_thur	SYN Flood	11:04:16	11:04:16
	Land	15:47:15	15:47:15
1999_week2_fri	Ping Flood	09:18:15	9:18:15
	SYN Flood	11:20:15	11:20:15

although there was also a problem with the 1998_week4_tues dataset. We prematurely detected the SYN Flood attack approximately three hours before it supposedly occurred. Depending on interpretation, this case can be classified as an early correct recognition or a “false-positive” at an earlier time and a “false-negative” at the attack time (just as with the 1998_week6_fri dataset). More detailed information on this attack is needed to see if indeed the first indication of the attack appeared as early as our IDS reported.

To compare our results with other IDSs on similar datasets, we examine the collective results from the DARPA evaluation documents for the 1998 and 1999 attacks. Because this system only functions on DoS attacks for which signature automata have been created, we only focus on the results others have obtained on old (or previously seen) DoS attacks. One should note that different IDSs and different attacks were used in the 1998 and in the 1999 evaluations. The five attacks on which the prototype TDFA was tested constitute a subset of those tested in the evaluations. A small graph summarizing

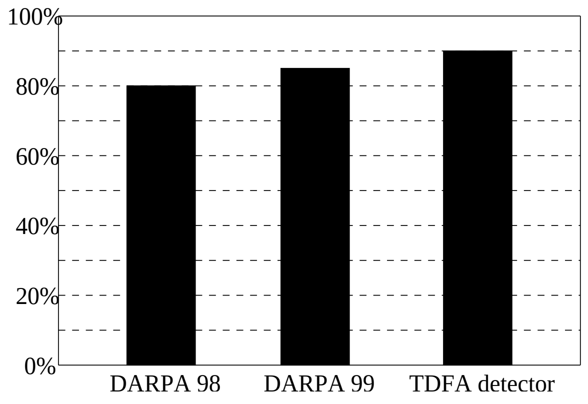


Figure 2.5: Graph summary of detection results from DARPA 1998 and 1999 evaluations for DoS attacks

the collective results from the 1998 and 1999 DARPA evaluations is given in Fig. 2.5. While not perfect, our current results are very competitive when compared to the results achieved by the systems used in the DARPA evaluations.

2.5 Future Work

It was mentioned earlier the opportunities that the TDFA provider component can offer in the area of distributed attack detection. Currently, this is the focus of our future developments pertaining to this system.

The proposed application, as well as other intrusion detection systems, use “sniffed” network traffic to determine significant intrusion events. However, sniffed data traditionally can only be obtained from a packet sniffer running on the same network as the monitored host machines. When the monitored hosts are in different networks, multiple sniffers are needed. The growing degree of segmentation in networks has caused problems in many intrusion detection applications relying on network sniffed data [21]. To facilitate distributed detection, it is important to recall the modular framework of our application such as that portrayed in Fig. 2.3. Modularity such as this would be necessary to distribute the functionality of our system. Our intent is to use the DOORS system (Distributed Online Object Repositories) in which our detection process would be encoded into mobile agents and sent to the network in need of monitoring [4, 3]. Fig. 2.6 illustrates the topology of such a network middleware architecture. This figure is a portrayal

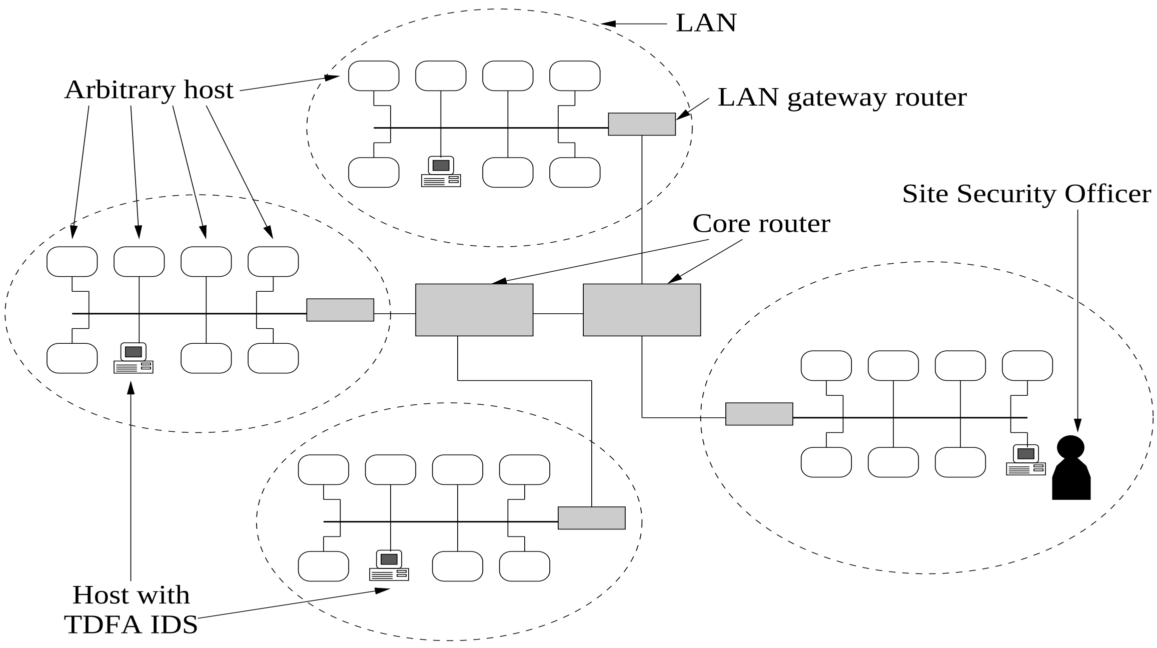


Figure 2.6: Overview of distributed modular architecture

of a large network in which smaller LANs are connected by gateway and core routers. In this case, a host in each monitored network will be equipped with a software “docking port” to receive an agent containing our TDFA IDS. A site security officer will update the host running the IDS with a new TDFA structure when necessary by sending a new TDFA to the TDFA provider module shown in Fig. 2.3. This module would then immediately replace the TDFA structure used by the TDFA transversal unit to recognize unfolding attacks. We believe that this solution would effectively distribute the detection while providing a centralized management for continual TDFA updates.

Another area of future development will focus on the Data Filtration Unit. A fully configurable parser will be developed so that the input source, the output format and the packet data fields to be captured can all be specified inside a single configuration file. As was mentioned earlier, the input source can either be an off-line tcpdump file or the real-time network traffic. Besides sending the output to the ETG, the next version of DFU will also be able to store the delimited text output into a plain ASCII file or into the database server for future references and further analyses.

Finally, this would be an appropriate section to discuss actual attack prevention.

Currently, our prototype has only the ability to detect an attack, but not to actually prevent it. Therefore, another direction in which we would like to take this project is to either add DoS attack prevention methods ourselves, or to extend the system's interface to interact with currently available tools that can offer preventative measures. Either way, a common strategy would be to identify probable scenarios of "no return." In other words, preventative measures (i.e. blocking requests from an attacker's address) should be invoked when the supplied TDFA reaches a state where the probability of attack completion is very high.

2.6 Contributions and Conclusion

My major contributions regarding this project dealt with (or currently deal with) background research, architecture design, and testing. I performed most of the initial background research concerning this project, which helped us identify and assess the feasibility of the contributions we could make in network-based intrusion detection. As far as contributing to the project's architecture, I helped in deciding the most effective use of the deterministic finite automata in the detection process. This involved the definition of default transitions and what states what states of operation they should return the automata to upon execution. Most importantly, I am invloved in further evaluation of the prototype. The current results from our system evaluation are encouraging. However, we are concerned about the lack of SYN flood attack detection in the 1998 datasets which could have been caused by limitation of our TDFA that currently does not embody all variations of this attack signatures. I am still studying the DARPA datasets in attempts to improve our TDFA model. I am also in the process of testing our system for the remaining attacks present in the DARPA datasets: process table and UDP storm.

In general, we acknowledge that our IDS, being a misuse detection system, has a significant flaw common to all such systems: it can only detect attacks for which it knows a signature. Detecting malicious activity by recognizing legitimate use signatures (patterns), however, has been a widely supported practice. However, misuse systems such as ours, have much lower numbers of false-positives in the detection process, which is very important for overall system security.

In general, network-based intrusion detection is still in early stages of development. As networks increasingly become more complex, the need for sophisticated security tools will rapidly grow in importance. While this IDS does not detect all malicious network penetrations, it does a great job detecting a significant subset of these attacks: denial of service. DoS attacks are widely publicized as they inflict damage on many substantial commercial, educational and government web sites. Our IDS detects these attacks in an accurate and efficient manner and is compact enough to be coupled with other IDSs (perhaps even anomaly detection systems) to build a complete suite of general attack detection/prevention tools on multiple platforms.

CHAPTER 3
Host-Based Masquerade Detection Using Probabilistic State Finite Automata

3.1 Introduction

In this chapter, we focus on detecting masquerade attacks. These are attacks in which an intruder gains access to another user's account. Once access is gained, the intruder will usually find a way for repeated access. These are particularly dangerous attacks because if a user account with high privileges is compromised, sensitive information and vital business/military components could be exposed. These attacks can also last for a very long period of time if gone undetected. This chapter proposes a method of detecting masquerade attacks by generating a "signature" of a user's normal behavior on a computer system. We assume that commands entered in a rapid sequence are characteristic for a typical behavior of a legitimate user of the account and they differ from such sequences entered by an intruder. Typically, such sequences contain a lot of "navigational" commands (such as listing or changing a directory) that indicate familiarity with an account. Hence, we create and store a signature of all the frequent (and infrequent) commands that a user types in a rapid sequence. Future command traces executed in the user's "account" are then compared with the stored signature. The novelty of this method lies in the use of finite automata to represent signatures. First, we use time of the command entry to create input sequences for a signature automaton. Second, we compute the probability of each transition in the automaton. As a result, we can build an accurate model of user behavior and perform dynamic detection algorithms–as described in detail later.

3.2 Probabilistic State Finite Automata Approach

This work outlines an anomaly detection system that generates signatures of normal behavior for each user of a computer system. The assumption is that rapidly entered sequences of commands are characteristic for a normal behavior of the legitimate user. Typically, such sequences include "navigational" commands that indicate a user's familiarity with a structure of the account. By storing such sequences and comparing them with the actual input, the belief is that we can discover masquerading attacks quickly and efficiently.

Anomalous behavior is defined as any behavior that sufficiently deviates from the model. Thus, the detected anomalies may or may not correspond to an actual intrusion. In some instances, the user may simply be experimenting with a new set of commands, or the user's behavior may be different due to fatigue or stress.

A probabilistic state finite automata (PSFA) can be viewed as an extension of a finite state machine in which a probability is assigned to each state. An example of a PSFA is shown in Figure 3.1. For each node m, $P(m)$, is the probability of reaching m from the predecessor nodes. It is computed from a sample population while building the PSFA.

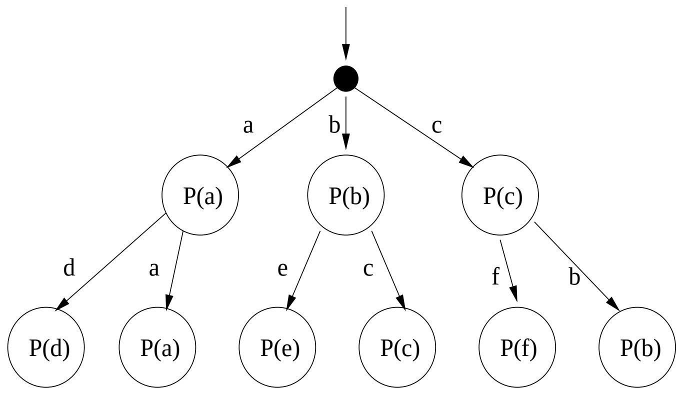


Figure 3.1: PSFA example

3.3 System Overview

Our system consists of four basic modules as shown in Figure 3.2. The first module, the Filter Module, is responsible for extracting user commands from log files and converting them (including their timestamps) into command traces. The traces are then either passed to the PSFA Builder, which constructs the PSFA, or to the PSFA Tester for anomaly detection. The PSFA Provider is responsible for passing the appropriate PSFA to the PSFA Tester.

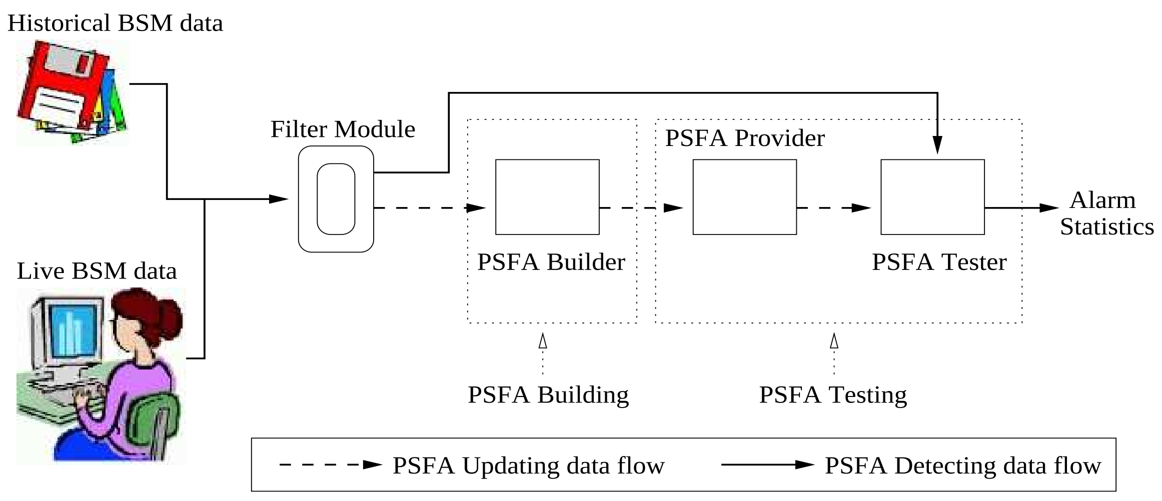


Figure 3.2: PSFA system architecture

3.3.1 Filter module

The proposed system can use either Solaris BSM (Basic Security Module) [31] or the UNIX *acct* accounting mechanism [27] for collecting audit data. BSM reports user sessions at the system call level while *acct* does the same at the user-typed command level. The Filter Module parses BSM output (*acct* output is already in the proper format) for system calls created by the user actions (typed or entered via a GUI). For every user action, many system calls are generated. Our system extracts just three system commands: *execve* used by the shell to execute most user-entered commands, *login* and *chdir*.

The string format produced by the Filter Module contains the timestamp followed by the name of the command, as shown in Figure 3.3.

$$Trace_{user} = < ts_1 > \textvisiblespace < c_1 >, \textvisiblespace < ts_2 > \textvisiblespace < c_2 >, \textvisiblespace \cdots < ts_n > \textvisiblespace < c_n >$$

where: $\textvisiblespace$ = space character
ts_t = timestamp at time t
c_t = command entered at ts_t

Figure 3.3: String representation of a user command trace

3.3.2 PSFA builder

Command traces are formed based on the time intervals between successive commands in the user input. Two successive commands are considered to be in the same command trace if the difference between their timestamps is less than the predefined threshold, δt. Figure 3.4 shows a user signature containing q commands.

$$Signature_{user} = <ts_1> \textvisiblespace <c_1>, \textvisiblespace <ts_2> \textvisiblespace <c_2>, \textvisiblespace \cdots <ts_q> \textvisiblespace <c_q>$$

where: $\{\forall i \in 2, 3, .., q\} | ts_i - ts_{i-1} \leq \delta t$
$\textvisiblespace$ = space character
ts_t = timestamp at time t
c_t = command entered at ts_t
δt = command inter-arrival threshhold

Figure 3.4: String representation of a user signature

The value of threshold δt defines an average command trace length and therefore strongly impacts the structure of the PSFA. Table 3.1 shows an example of two sets of command traces derived from the same command listing, based on different command inter-arrival thresholds.

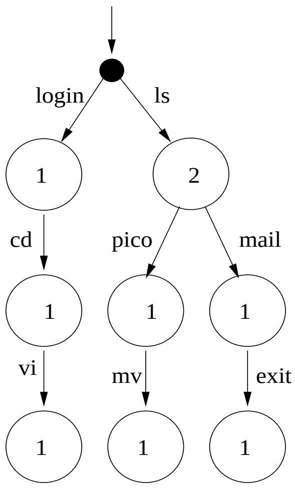
Time	Command
10:00	login
10:01	cd
10:02	vi
10:04	ls
10:05	pico
10:06	mv
10:08	ls
10:09	mail
10:10	exit

δt	Command Window
δt=1	[login, cd, vi] [ls, pico, mv] [ls, mail, exit]
δt=3	[login, cd, vi, ls, pico, mv, ls, mail, exit]

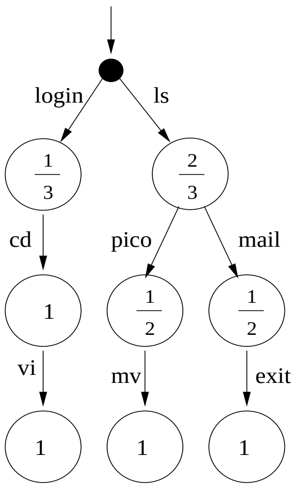
Table 3.1: Sample host command dataset

3.3.2.1 Building the PSFA from command traces

The PSFA represents all command traces extracted from the given user log files. It has a tree structure. Each node represents the user command and each edge shows the apperance of the given pair of commands in the command trace. Command traces are added to the PSFA one after another. Figure 3.5 shows the formation of a PSFA consisting of the command traces from the Table 3.1 where $\delta t = 1$. To calculate the probability $P(m)$ during the training stage, we first keep a count of the number of times the node in the PSFA was reached (Figure 3.5(a)). $P(m)$ is computed as a ratio of its reach count to the sum of reach counts of its siblings in the tree. Figure 3.5(b) shows the automaton with $P(m)$ values computed from the data shown in Figure 3.5(a).



(a) Sample PSFA totals created from sample host data shown in Table 3.1.



(b) The PSFA with probabilities in the nodes of the automata.

Figure 3.5: Formation of PSFA from sample data in Table 3.1 where $\delta t = 1$.

The probability of a command trace, P_m, is simply the product of the probabilities $P(m)$ of nodes encountered during the PSFA traversal induced by this trace. We also compute the average probability, P_a, from all of the $P(m)$ values associated with each node we encounter. We use P_m and P_a to distinguish frequently seen command traces from infrequent ones. Significantly low P_m and P_a indicate that the tested command trace is anomalous.

3.3.2.2 Updating the probabilities using standard deviation

The use of P_m and P_a as described above introduces an unintentional bias against two types of advanced users.

- **Case 1:** PSFA for users who frequently use a large variety of commands yield small probabilities for each individual command and yield low P_m and P_a values.
- **Case 2:** PSFA for users who type commands quickly yield long command traces that have low P_m.

To address this bias, we introduced a standard deviation based indicator. For each trace, we compute $P_\sigma(m)$ for each node. This value measures how many standard deviations is the probability of reaching that node away from the average probability of reaching all the children of the parent node. The exact formula is shown in Equation 3.1.

$$P_\sigma(m) = \begin{cases} 1 + C_{reward} * \frac{P(m) - P_{avg}}{\sigma} & if\ P(m) > P_{avg}(m) \\ 1 + C_{penalty} * \frac{P(m) - P_{avg}}{\sigma} & if\ P(m) < P_{avg}(m) \\ 1 & otherwise \end{cases} \tag{3.1}$$

$P(m)$	the probability for a particular state
$P_\sigma(m)$	standard deviation-based probability for a particular state
C_{reward}	reward coefficient
$C_{penalty}$	penalty coefficient
$P_{avg}(m)$	the average breadth probability found in the level and branch of $P(m)$
σ	the standard deviation found in the level and branch of $P(m)$

For very long traces, we compute an additional reward based on the average trace length. Specifically, this reward is based on how many standard deviations away is the current trace's length from the average trace length. The formula for computing this reward is shown in Equation 3.2.

$$DepthOffset(m) = C_{reward} * \frac{L(m) - L_{avg}}{\sigma_{len}} \quad if\ L(m) > L_{avg} \tag{3.2}$$

$DepthOffset(m)$	the offset used to account for fast users
$L(m)$	length of the particular trace
C_{reward}	reward coefficient
L_{avg}	the average length of traces in the PSFA
σ_{len}	the standard deviation of trace lengths

3.3.2.3 Mutation events

If the next command entered, while traversing the PSFA, matches none of the commands in the next level of the PSFA, a mutation event has occurred. This action may be the sign of a new mutation of the same signature. Therefore, traversal of the PSFA does not stop. We simply determine a probabilistic value for the command in question and proceed to the next command. When developing a probabilistic value for the command missing from the signature, the goal is to make a distinction between commands which the user rarely or never uses and those the user uses frequently. To do this we keep all of the commands the user uses in a table, along with the number of times the command was entered. A probabilistic value is returned reflecting the frequency of the command's use.

3.3.3 PSFA tester

Once a PSFA is created for a specific user, future command traces can be tested against the PSFA to determine if the behavior is anomalous or not. The traversal of the PSFA for testing proceeds as follows:

- Get the next command from the test command trace and look for a transition containing the command in the next level of the automaton. In the beginning, this would be the first command in the trace and the first level of the automaton.
  - If there is an associated transition, follow the transition and keep track of the probability $P_\sigma(m)$ at the encountered node.
  - If there is no transition for the command, assign it a probability of its usage by the given user (or a default low value if it is the first use).

3.4 Evaluation

To evaluate the performance of our method, the IDS was tested on data used by Maxion and Townsend [20]. The dataset was created by Schonlau et al. [27] and has thus been named the SEA dataset.

3.4.1 The SEA dataset

The masquerade detection dataset consist of user commands captured by the UNIX *acct* accounting mechanism. To protect privacy, only command name and user fields were used. 70 users were monitored and their first 15,000 commands were recorded. For some very active users, 15,000 commands were generated in only a few days, whereas other users took months to generate the necessary number of commands. The data was then organized into blocks of 100 commands, creating 150 blocks of data for each user.

After recording the library of commands for each user, 50 (of the 70) users were randomly chosen as intrusion targets. The remaining 20 users served as masqueraders whose data were interspersed into the data of the 50 targets. The first 50 blocks (5000 commands) of all users remained unaltered to serve as a training dataset. The next 100 blocks of intrusion target datasets were contaminated.

While the exact values of the new masquerade and continuing masquerade probabilities were arbitrary, they reflect the author's three requirements: (1) there should be an arbitrary number of masqueraders in the data (including the possibility of none), (2) the length of the actual masquerading attack should be varied, and (3) most of the user data should not be contaminated.

3.4.1.1 Conforming to the test data

As mentioned earlier, the SEA dataset used captured user commands from the UNIX *acct* accounting system. From this accounting system, only the command and user names were recorded. However, this method is built on temporal differences in the data. With no timestamps given in the SEA dataset, we had to adjust our method slightly. Instead of a command trace being comprised of all commands entered within a certain time δt of each other, we had to fix the length of the command trace to an arbitrary number of commands. To this end, all command traces lengths were fixed to be five commands (using a non-overlapping windowing method), making each user block contain 20 command traces.

If timestamps were available, we would expect our technique to generate more accurate results. However, testing on the SAE dataset can still give us an idea of how our method performs and present a basis for comparison with other methods which never consider the temporal variance of command entries in their techniques.

3.4.2 Results

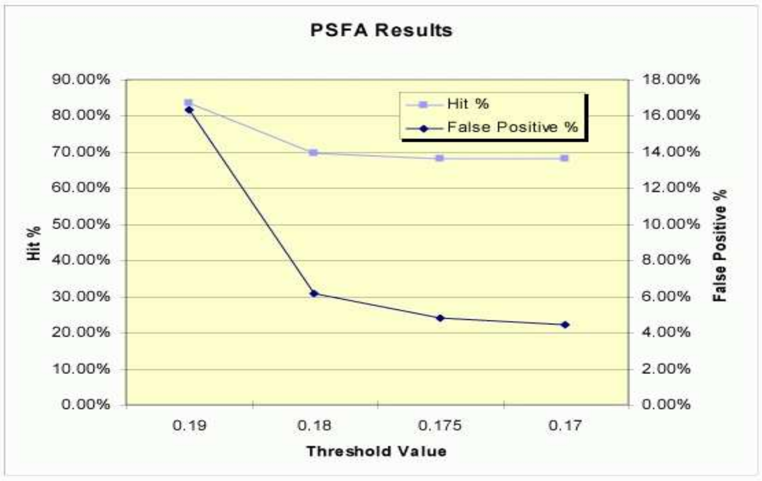


Figure 3.6: Performance with varying threshold values

Figure 3.6 shows the results of running our method on the SEA dataset for different P_a threshold values. This graph shows that the hit percentage achieved by most of our methods were in the high 60th percentile (with the exception of the very high hit percentage of 83% achieved when using 0.19 as the threshold value). This graph also shows that the false alarm rate for our methods is between 4% and 6% (except for the case with the threshold value of 0.19, which got a very large false alarm rate of around 16%).

Figure 3.7 shows the functional relationship between the hit rate and the false alarm rate in a graph called a ROC (Relative Operating Characteristic) curve. In this curve, the higher and more leftward a point is, the better the corresponding method is. Deciding which metric is more important, the vertical height or the horizontal length, is based on the preference of the application. For example, if an application needs to have a very small number of false alarms, it may choose not to employ the **PSFA(0.19)** method even though the method has the highest hit rate. On the other hand, if the goal is to discover all possible intrusions, this would be the method of choice.

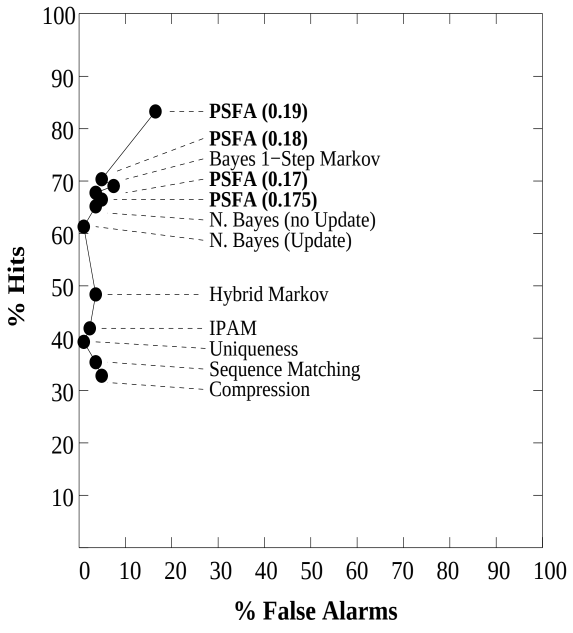


Figure 3.7: Relative operating characteristic curve

Following [20], we use the cost formula shown in Equation 3.3 to compare our method with others.

$$Cost = \frac{\alpha M}{A} + \frac{\beta F}{N - A} \tag{3.3}$$

α	miss multiplier ($\alpha = 1$ in Maxion evaluation)
M	number of attacks that were missed
A	the total number of attacks
β	false alarm multiplier ($\beta = 6$ in Maxion evaluation)
F	the number of times the alarm was raised on the normal user session
N	the total number of sessions (N=10,000 for SEA data)

Figure 3.8 shows the cost of our detection methods as well as other masquerade detection methods outlined in [20] and [27]. We should point out that setting $\beta = 6$ places a large bias against false alarms. In a more realistic environment, the penalty for false alarms may not be quite so high. However, even with value of β, although we are not the top method, we do occupy 2 of the top 3 places and 3 of the top 5 places as shown in Figure 3.8.

Another comparison could be based on the following observation. There is a certain cost of missing an attack as well as another cost of a false alarm that interrupts users and system managers work unnecessarily. In our opinion, a ratio, let us call it ρ, of the first

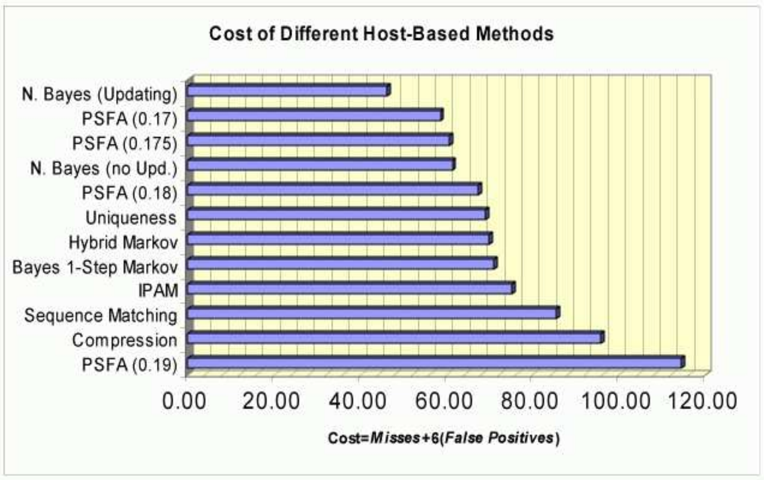


Figure 3.8: Cost of different host-based methods

cost to the second one, can range from about 10 to 100. Hence, we propose the following in Equation 3.4.

$$C(M,F,\rho) \quad = \quad \rho\frac{M}{N}+\frac{F}{N} \tag{3.4}$$

The advantage of the new formula is that it does not depend on the frequency of attacks, as the Maxion formula does, but only on the relative cost of missing an attack versus false alarm. Indeed, let $f = \frac{A}{N}$ be attack frequency. In Equation 3.5 we express Maxion's cost formula in terms of the new cost formula and ρ from Equation 3.4.

$$\begin{aligned} &\frac{\alpha M}{A}+\frac{\beta F}{N-A} \\ &\frac{\alpha M}{fN}+\frac{\beta F}{N(1-f)} \\ \frac{\beta}{1-f}\left(\left[\frac{\alpha(1-f)}{f\beta}\right]\frac{M}{N}+\frac{F}{N}\right) \quad &= \quad C\left(M,F,\frac{\alpha(1-f)}{f\beta}\right)\frac{\beta}{1-f} \end{aligned} \tag{3.5}$$

Note in Equation 3.5, ρ is computed to be $\frac{\alpha(1-f)}{f\beta}$. The α and β compunents of ρ are both constent coefficients, leaving the dominating factors of ρ to be $\frac{(1-f)}{f}$. Noticing that $\frac{(1-f)}{f} \approx \frac{1}{f}$ for $f << 1$ and $1-f$ for f close to 1, we see that the Maxion formula changes the relative cost of missing an attack depending on the frequency of attacks, from about $\frac{\alpha}{f\beta}$ for infrequent attacks to about $\frac{\alpha(1-f)}{\beta}$. Less frequent attacks are more costly, compared to

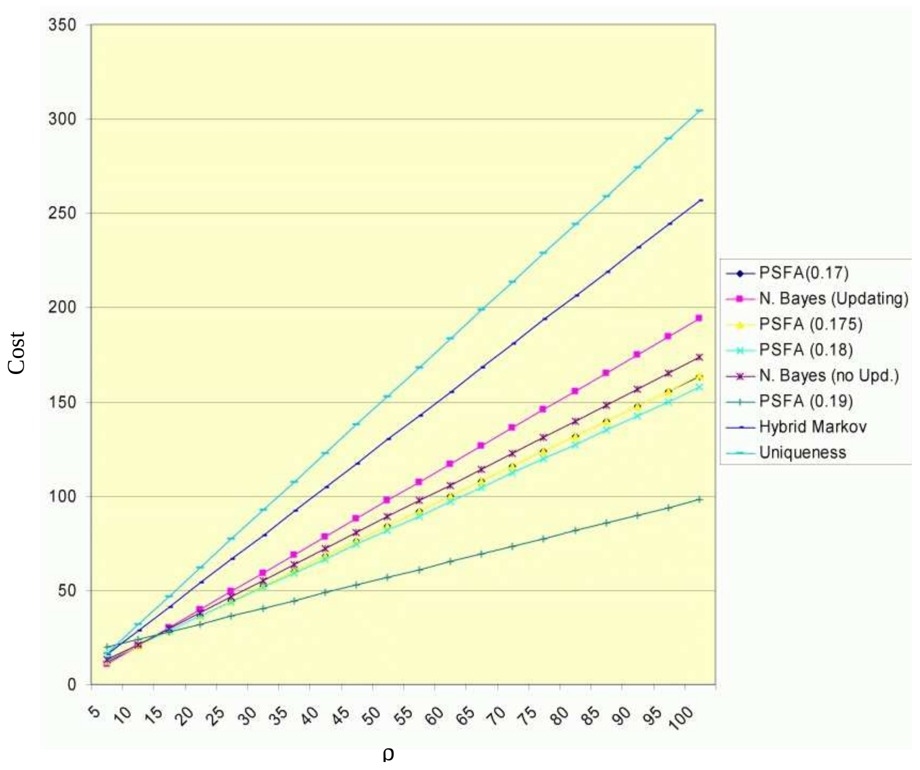


Figure 3.9: Cost of different host-based methods for different ρ values

the cost of raising a false alarm, then more frequent ones. In contrast, our formula keeps the ratio of costs of missing attacks to those of false alarms independent of frequency of attacks or false alarms.

To evaluate the other methods according to the new cost formula we needed to determine the number of attack blocks used in the SEA dataset. To develop an expected value for this statistic, a detailed analysis of how the dataset was formed must be conducted. Recall the details of the SEA dataset's block organization described in Section 3.4.1. Every user had 100 blocks of possibly contaminated data for testing purposes. A block of user data has a 1% chance of being contaminated. If the current block is contaminated, the next block will be contaminated with an 80% probability. Adding attack blocks extends the total number of blocks for a particular user, however the user's data remains capped (in many cases, truncated) at 100 blocks. This dependant distribution leads us to compute the expected value for the attack frequency in a particular dataset in Equation 3.6.

$$E(f) = \frac{(0.80)^{100} + \sum_{i=1}^{i<100}(0.80)^{i-1}(0.20)}{100} \approx 0.05 \tag{3.6}$$

To use Maxion's evaluation coefficients in our new cost formula, we would compute ρ according to Equation 3.7.

$$\rho = \frac{\alpha(1-f)}{\beta f} = \frac{1(1-f)}{6f} = \frac{1(1-0.05)}{6(0.05)} = 3.1667 \tag{3.7}$$

Figure 3.9 shows the new cost comparison for the top 8 methods using varing values of ρ. Note that all methods may be close at low values of ρ, but as the importance of insuring that a minimal number of missed attacks grows, the cost of different methods becomes more pronounced.

3.5 Future Works

A direction we would like to pursue is to classify users into distinct groups. Most likely, these user groups can be defined to reflect the policies and/or hierarchy of a guarded organization (e.g., a group of senior level programmers might constitute one group). We believe that while a user's behavior will eventually deviate from his normal activity, it is least likely to deviate from that of the group whose job functions are overall very similar. This could be one possible method of reducing false-alarm rates, a serious problem amongst most anomaly detection tools.

3.6 Contributions and Conclusion

Similar to the project outlined in Chapter 1, I contributed much in the way of background research and testing. I conducted the majority of the historical review for this project, once again helping us identify what advancements we could make in the masquerade detection community. My background research also lead to the discovery of Schonlau's experimental data. Schonlau's data was essential in collecting early test results because the time-stamped data we needed to test the novel features of our project did not yet exist or was (and still is) under restriction from use. The tests I designed allowed us to prompty evaluate our system's performance against other published works and under various operational configurations (i.e., using different trace lengths or using overlapping windows). Although the SEA datasets provide a solid basis for the evaluation and comparison of masquerade detection systems, we require datasets containing both the user commands and their timestamps. Datasets using this specific format along with information of exactly which command trace locations are contaminated (similar to the SEA datasets) will enable effective testing of our method. Another important extension of our work would be to enable our tool to process unsuccessful as well as successful user commands (e.g., the *mkdir* command accidently typed as *mkdor*). I am currently pursuing both of these objectives.

Additionally, I am in the process of further evaluating the system using different analytical models. I am employing different mathematical formulas to detect intrusions with the same (or greater) amount of accuracy while lowering the amount of false-positives. We do this by considering the frequency of occurence of a command which causes a particular command trace to score low. Such a command should not cause a user to be heavily penalized because its misplacement in the command trace should not carry great significance, since it is rarely seen in the training data. Most of this work speaks to the mutaion events discussed earlier in the chapter.

Another extension to this work which I am helping to pursue involves the application of bioinformatics to the masquerade detection process. Various genomic algorithms are used to identify similarities between multiple gene sequences. Viewing our command traces as gene sequences, we can apply genomic algorithms, such as Smith Waterman, to measure similarities between a user's training data and his or her test data. This technique has not been widely published, thus may serve as yet another novel contribution in masquerade detection.

In this chapter, we have presented a host-based masquerade detection tool which is very successful at identifying anomalous user behavior. The strength of the tool results from the way it creates and stores the signatures of the frequent commands and their order. These signatures provide accurate models of typical user behavior. Any user behavior that clearly detours from the signature will raise an alarm.

LITERATURE CITED

[1] R. K. Ackerman. Government enlists industry for information security. In *Singal*, volume 56, pages 17–20. Armed Forces Communications and Electronics Association, 4400 Fair Lakes Court, Fairfax, VA 22033, August 2002.

[2] A. Bivens, M. Embrechts, C. Palagiri, A. Smith, and B. Szymanski. Network-based intrusion detection using neural networks. In *Proceedings of Artificial Neural Networks in Engineering Conference (ANNIE'02)*, pages 579–584, St. Louis, Missouri, November 10-13 2002.

[3] A. Bivens, P. Fry, L. Gao, M. Hulber, B. Szymanski, and Q. Zhang. Distributed object-oriented repositories for network management. In *Proc. 13th Int. Conference on Software Engineering*, pages CS169–174, Las Vegas, NV, August 1999.

[4] J. A. Bivens, L. Gao, M. Hulber, and B. Szymanski. Agent-based network monitoring. In *Agent based High Performance Computing*, Seattle, Washington, May 1999. Autonomous Agents: Sponsered by ACM SigART.

[5] J. Branch, A. Bivens, C. Y. Chan, T. K. Lee, and B. Szymanski. Denial of service intrusion detection using time dependent deterministic finite automata. In *Proceedings of The First Annual Walter Lincoln Hawkins '32 Graduate Research Conference*, pages 45–51, Troy, New York, October 2002.

[6] J. Branch, A. Bivens, S. Freeman, and B. Szymanski. Host-based masquerade detection using probabilistic state finite automata. Submitted to The International Conference on Dependable Systems and Networks.

[7] J. Cannady. Artificial neural networks for misuse detection. In *1998 National Information Systems Security Conference*, pages 443–456, Arlington, VA, October 1998. (NISSC'98).

[8] W. DuMouchel. Computer intrusion detection based on bayes factors for comparing command transition probabilities. Technical Report TR91, National Institute of Statistical Sciences (NISS), 1999.

[9] S. Eckmann, G. Vigna, and R. Kemmerer. Statl: An attack language for state-based intrusion detection. In *Proceedings of the ACM Workshop on Intrusion Detection*, Athens, Greece, November 2000.

[10] L. Eschenauer. ImSafe - host based anomaly detection tool. http://imsafe.sourceforge.net/, February 2001.

[11] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[12] S. Freeman, A. Bivens, J. Branch, and B. Szymanski. Host-based intrusion detection using user signatures. In *Proceedings of The First Annual Walter Lincoln Hawkins '32 Graduate Research Conference*, pages 33–38, Troy, New York, October 2002.

[13] A. K. Ghosh, A. Schwartzbard, and M. Schatz. Learning program behavior profiles for intrusion detection. In *Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, CA., April 1999. Reliable Software Technologies Corporation.

[14] K. Ilgun, R. A. Kemmerer, and P. A. Porras. State transition analysis: A rule-based intrusion detection approach. *IEEE Transactions on Software Engineering*, 21(3):181–199, 1995.

[15] S. M. Inc. Sunshield basic security module guide, solaris 7. Sun Part No. 8052635-10, October 1998.

[16] V. Jacobson, C. Leres, and S. McCanne. Tcpdump, June 1989. Available via anonymous FTP from ftp.ee.lbl.gov.

[17] K. Kendall. A database of computer attacks for the evaluation of intrusion detection systems. Master's thesis, Massachusetts Institute of Technology, 1998.

[18] R. Lippmann, R. K. Cunningham, D. J. Fried, I. Graf, K. R. Kendall, S. E. Webster, and M. A. Zissman. Results of the darpa 1998 offline intrusion detection evaluation. In *Recent Advances in Intrusion Detection 1999*, West Lafayette, Indiana, September 1999. Second International Workshop, RAID 1999.

[19] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das. The 1999 darpa off-line intrusion detection evaluation. *Computer Networks*, 34(4):579–595, October 2000.

[20] R. A. Maxion and T. N. Townsend. Masquerade detection using truncated command lines. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN-02)*, pages 219–228, Washington, D.C., June 2002. IEEE Computer Society Press, Los Alamitos, California.

[21] S. McCloghrie and J. Scambray. Once-promised intrusion detection system stumbles over switched networks. In *InfoWorld*, volume 22, page 58. InfoWorld Media Group, Inc., December 2000.

[22] I. McLean, A. Bivens, and B. Szymanski. Methodology of risk assessment in mobile agent system design. submitted to the 4th Annual Information Assurance Workshop.

[23] E. Messmer. Intruvert inspects high-speed ip traffic. NetworkWorldFusion, September 2002. http://www.nwfusion.com/news/2002/135068_09-09-2002.html.

[24] D. Newman, J. Snyder, and R. Thayer. Crying wolf: False alarms hide attacks. In *NetworkWorld*, volume 19, pages 57–64. Network World, Inc., 118 Turnpike Road, Southborough, MA 01772, June 2002.

[25] P. Porras and P. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings 20th National Information Systems Security Conference*, pages 353–365, Baltimore, Maryland, October 1997.

[26] M. Roesch. Snort: Lightweight intrusion detection for networks. In *Proceedings of the LISA'99 conference. USENIX*. USENIX, November 1999. URL http://www.snort.org/lisapaper.txt.

[27] M. Schonlau, W. DuMouchel, W.-H. Ju, A. F. Karr, M. Theus, and Y. Vardi. Computer intrusion: Detecting masquerades. *Statistical Science*, 16(1):58–74, February 2001.

[28] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[29] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – A graph-based intrusion detection system for large networks. In *Proceedings of the 19th National Information Systems Security Conference*, volume 1, pages 361–370, October 1996.

[30] W. R. Stevens. *TCP/IP Illustrated Volume 1: The protocols*, volume 1. Addison-Wesley Publishing Company, One Jacob Way; Reading, Massachusetts 01867, 1994.

[31] I. Sun Microsystems. Answerbook2: Sunshield basic security module guide. http://docsun.cso.uiuc.edu/cgi-bin/nph-dweb/ab2/coll.47.11/SHIELD/@Ab2PageView/3507, 2000.

[32] B. Szymanski, A. Bivens, Y. Liu, K. Madnani, and A. Sastry. The genesis project: Network decomposition in monitoring and simulation for network management and intrusion detection. In *Proceedings of the Virtual Worlds and Simulation Conference*, San Antonio, Texas, January 27-31 2002.

[33] G. Vigna and R. A. Kemmerer. Netstat: A network-based intrusion detection approach. In *Proceedings of the 14th Annual Computer Security Conference*, pages 25–36, Scottsdale, Arizona, December 1998.