A Methodology for Testing Intrusion Detection Systems[1]

Nicholas J. Puketza, Kui Zhang, Mandy Chung,
Biswanath Mukherjee*, Ronald A. Olsson

*Correspondence Author
Department of Computer Science
University of California, Davis
Davis, CA 95616
Tel: (916) 752-4826
Fax: (916) 752-4767
E-mail: mukherje@cs.ucdavis.edu

Original submission date: October, 1993
First revision date: September 8, 1995
Second revision date: September 27, 1996

Abstract

Intrusion Detection Systems (IDSs) attempt to identify unauthorized use, misuse, and abuse of computer systems. In response to the growth in the use and development of IDSs, we have developed a methodology for testing IDSs. The methodology consists of techniques from the field of software testing which we have adapted for the specific purpose of testing IDSs. In this paper, we identify a set of general IDS performance objectives which is the basis for the methodology. We present the details of the methodology, including strategies for test-case selection and specific testing procedures. We include quantitative results from testing experiments on the Network Security Monitor (NSM), an IDS developed at UC Davis. We present an overview of the software platform that we have used to create user-simulation scripts for testing experiments. The platform consists of the UNIX tool *expect* and enhancements that we have developed, including mechanisms for concurrent scripts and a record-and-replay feature. We also provide background information on intrusions and IDSs to motivate our work.

[1]This work has been funded by the National Security Agency INFOSEC University Research Program under Contract Number DOD-MDA904-93-C-4084. A short, summarized version of this work was presented at the 17th National Computer Security Conference in Baltimore, MD, in October, 1994.

1 INTRODUCTION

An intrusion detection system (IDS) is a system that attempts to identify *intrusions*, which we define to be unauthorized uses, misuses, or abuses of computer systems by either authorized users or external perpetrators [27]. Some IDSs monitor a single computer, while others monitor several computers connected by a network. IDSs detect intrusions by analyzing information about user activity from sources such as audit records, system tables, and network traffic summaries. IDSs have been developed and used at several institutions. Some example IDSs are National Security Agency's Multics Intrusion Detection and Alerting System (MIDAS) [31], AT&T's ComputerWatch [9], SRI International's Intrusion Detection Expert System (IDES) [24, 25] and Next-Generation Intrusion-Detection Expert System (NIDES) [1], UC Santa Barbara's State Transition Analysis Tool for UNIX (USTAT) [15, 16], Los Alamos National Laboratory's (LANL's) Network Anomaly Detection and Intrusion Reporter (NADIR) [14], and UC Davis' Network Security Monitor (NSM) [13] and Distributed Intrusion Detection System (DIDS) [33].

As more and more organizations depend on IDSs as integral components of their computer security systems, techniques for evaluating IDSs are becoming more important. IDS users need to know how effective their IDSs are, so that they can decide to what extent they can *rely* on their IDSs, and to what extent they must rely on other security mechanisms. However, evaluating an IDS is a difficult task. First, it can be difficult or impossible to identify the set of all possible intrusions that might occur at the site where a particular IDS is employed. To start with, the number of intrusion techniques is quite large (e.g., see [28]). Then, the site may not have access to information about all of the intrusions that have been detected in the past at other locations. Also, intruders can discover previously unknown vulnerabilities in a computer system, and then use new intrusion techniques to exploit the vulnerabilities. A second difficulty in evaluating an IDS is that an IDS can be affected by various conditions in the computer system. For example, even if an IDS can ordinarily detect a particular intrusion, the IDS may fail to detect that same intrusion when the overall level of computing activity in the system is high. This complicates the task of thoroughly testing the IDS.

We have developed a methodology for testing IDSs which confronts these difficulties. As the basis for the methodology, we have identified a set of general IDS performance objectives, such as the ability to detect a broad range of known intrusions. The methodology is designed to measure the effectiveness of an IDS with respect to these objectives. It consists of strategies for selecting test cases, and a series of detailed testing procedures. To develop the methodology, we have borrowed techniques from the field of software testing and adapted them for the specific purpose of testing IDSs. We use the UNIX tool *expect* [23] as a software platform for creating user-simulation scripts for testing experiments. In addition, we have enhanced *expect* with features that allow us to simulate more-sophisticated intrusions, and a feature that greatly facilitates script creation.

Our work should be useful to IDS developers, who can use our methods and tools to supplement their own approaches to testing their respective IDSs. System administrators can use our work to check for weaknesses in the IDSs that they currently employ. An organization that plans to acquire an IDS can use our work to compare the relative strengths and weaknesses of several IDSs and choose the system that best fits their computing environment.

As background, Section 2 of this paper presents some examples of intrusions, provides motivation for intrusion detection, and discusses specific approaches to intrusion detection. Section 3 describes a software platform for testing experiments, which consists of *expect* (which, in turn, is based on *Tcl* [29]) and our enhancements. Section 4 begins the discussion of our testing methodology by first identifying a set of important IDS performance objectives, and then discussing the selection of test cases, and some limitations to our approach. Section 5 describes our specific procedures for testing experiments in detail. Section 6 presents quantitative results from our own testing experiments which we conducted on the NSM. Section 7 concludes the paper.

2 BACKGROUND

In this section, we present some information on intrusions and intrusion detection as background for our work. Readers who are already familiar with these topics may skip this section.

2.1 Intrusions

Intrusions in computer systems are occurring at an increasingly alarming rate. Some sites report that they are the targets of hundreds of intrusion attempts per month [3]. Moreover, there are numerous different intrusion techniques used by intruders [28]. The following scenarios are examples of intrusions:

- An employee browses through his/her boss' employee reviews;
- A user exploits a flaw in a file server program to gain access to and then to corrupt another user's file;
- A user exploits a flaw in a system program to obtain *super-user* status;
- An intruder uses a script to "crack" the passwords of other users on a computer;
- An intruder installs a "snooping program" on a computer to inspect network traffic [12], which often contains user passwords and other sensitive data; and
- An intruder modifies router tables in a network to prevent the delivery of messages to a particular computer.

The reader can easily infer some of the consequences of intrusions from the preceding list. Some additional consequences include loss or alteration of data, loss of money when financial records are altered by intruders, denial of service to legitimate users, loss of trust in the computer/network system, and loss of public confidence in the organization that is the victim of an intrusion [12].

2.2 Concurrent Intrusions

In addition to the variety of intrusion techniques, another complication in the task of detecting intrusions is the possibility of *concurrent intrusions*, in which one or more intruders use several terminals (or "windows" on a workstation) to carry out one or more intrusions simultaneously. Based on this observation, we classify intrusions according to the following categories[2]:

- **Single Intruder Single Terminal (SIST)**
  Intrusions in this category are launched by a single intruder from a single terminal device or its logical equivalent. The terminal device might be connected directly to the system being attacked, or it might be connected to the system remotely via a modem or network connection.
- **Single Intruder Multiple Terminal (SIMT)**
  This category consists of intrusion scenarios in which an intruder uses multiple windows on a computer to carry out one or more intrusions. The intruder might use each window to attack a different target computer. Alternatively, the intruder might use multiple windows to establish several connections to the same target, hoping to hide the intrusive activity by distributing the activity over several windows, each of which controls a separate "session" on the target computer.
- **Multiple Intruder Multiple Terminal (MIMT)**
  This category covers scenarios in which multiple intruders participate in one or more intrusions simultaneously. Intrusions in this category are similar to those in the SIMT category, in that there may be one or more target computers, and the intruders might attempt to conceal the intrusion attempt by distributing the suspicious behavior over several simultaneous sessions.

2.3 Motivation for Intrusion Detection

One approach to computer security is to attempt to create a completely-secure system. Unfortunately, in many environments, it may not be feasible to render the computer system

[2]In each of the categories, an intruder can be a *person* issuing commands manually, or a *computer* issuing commands automatically based on an intrusion script or program.

immune to intrusions, for several reasons. First, system software is becoming more complex. A major challenge programmers face in software design is the difficulty in anticipating all conditions that may occur during program execution and understanding precisely the implications of even small deviations in such conditions. Thus, system software often contains flaws that may create security problems, and software upgrades often introduce new problems. Second, the increasing demand for network connectivity makes it difficult, if not impossible, to isolate and thereby protect a system from external penetration. Finally, a central component of computer systems, the computer network itself, may not be secure. For instance, there are a number of security flaws inherent in the widely-used Transmission Control Protocol/Internet Protocol (TCP/IP) suite, regardless of its particular implementation [4].

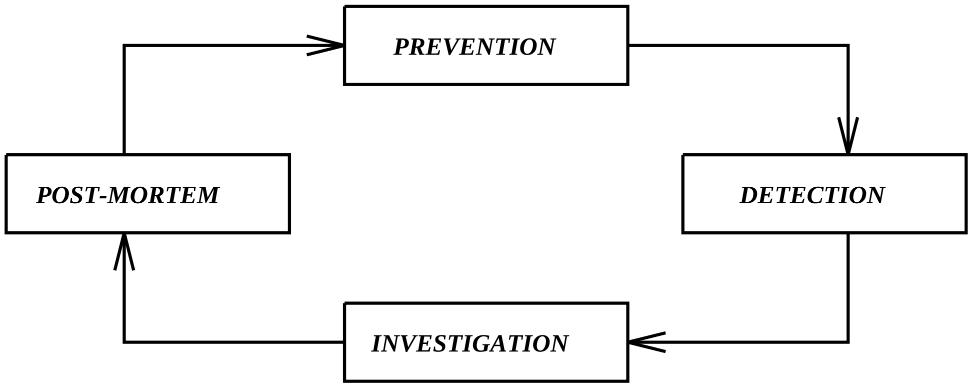

Figure 1: A Computer System Security Management Model.

In response to these difficulties in developing secure systems (which are discussed further in [27]), a new model of system security management [2] has emerged. The model is pictured in Figure 1. In this more-realistic approach to security, developing secure systems (the *prevention* component) is just one of four parts of the security system. The *detection* component identifies security breaches. The *investigation* component determines exactly what happened based on data from the detection component. This component may also include the gathering of further data in order to identify the security violator. Finally, the *post-mortem* component analyzes how to prevent similar intrusions in the future. In the past, most of the attention of computer security researchers has been focused on the *prevention* component. With the emergence and the proven utility of the intrusion detection concept, the *detection* component is beginning to receive more attention. Unfortunately, the other two components in Figure 1 have not yet received sufficient attention. In summary, the new model divides the field of computer security into four non-trivial and challenging but more manageable sub-problems; also, it encourages security researchers and practitioners to distribute their efforts over *each* of these important components of computer security.

2.4 Approaches to Intrusion Detection

The two major approaches that are used by IDSs to detect intrusive behavior are called *anomaly detection* and *misuse detection*. The anomaly-detection approach is based on the premise that an attack on a computer system (or network) will be noticeably different from normal system (or network) activity, and an intruder (possibly masquerading as a legitimate user) will exhibit a pattern of behavior different from the normal user [8]. So, the IDS attempts to characterize each user's normal behavior, often by maintaining statistical profiles of each user's activities [25, 17]. Each profile includes information about the user's computing behavior such as normal login time, duration of login session, CPU usage, disk usage, favorite editor, and so forth. The IDS can then use the profiles to monitor current user activity and compare it with past user activity. Whenever the difference between a user's current activity and past activity falls outside some predefined "bounds" (threshold values for each item in the profile), the activity is considered to be anomalous, and hence suspicious. The interested reader is referred to [17] for a thorough discussion of both this topic and the implementation of the IDES anomaly detection component.

In the misuse-detection approach, the IDS watches for indications of "specific, precisely-representable techniques of computer system abuse" [18]. The IDS includes a collection of intrusion *signatures*, which are encapsulations of the identifying characteristics of specific intrusion techniques. The IDS detects intrusions by searching for these "tell-tale" intrusion signatures in the records of user activities.

Although there exist only the above two major approaches to intrusion detection, IDSs are nevertheless quite diverse in their designs. Different IDSs employ different algorithms, different criteria for identifying intrusive behavior, and so forth. Also, several IDSs (including several of the example IDSs mentioned in Section 1) use a combination of *both* detection approaches.

Often, the main source of information about user activity for an IDS is the set of audit records from the computer system. However, relying on audit records *alone* can be problematic [27]. First, audit records may not arrive in a timely fashion. Some IDSs use a separate computer to perform the analysis of audit records (e.g., Haystack [32]). So, it may take a significant amount of time to transfer the audit information from the monitored computer to the computer which performs the analysis. Second, the audit system itself may be vulnerable. Intruders have been known to be able to turn off the audit system or to modify the audit records to hide their intrusions. Finally, the audit records may not contain enough information to detect certain intrusions. For example, in the so-called *doorknob attack* [33], an intruder tries to guess passwords of accounts on several computers in a network. To avoid arousing suspicion, the intruder attempts only a few guesses on each individual computer. This intrusion is not likely to be detected by analysis of the audit records of any single computer in the network. We use the term *network intrusion* to refer to such an intrusion that involves more than one computer (or other component) in a network.

This example illustrates that an effective IDS should also collect and analyze information from the network itself. For example, the NSM monitors network traffic on a Local Area Network (LAN), so the NSM can detect security-related network events such as the transfer of a password file across a network. DIDS collects and analyzes information from both the group of monitored computers and the network. Thus, DIDS is capable of recognizing network intrusions such as the aforementioned doorknob attack. A number of other scenarios in which the aggregation of information from a network of computers is necessary to detect intrusions is described in [33].

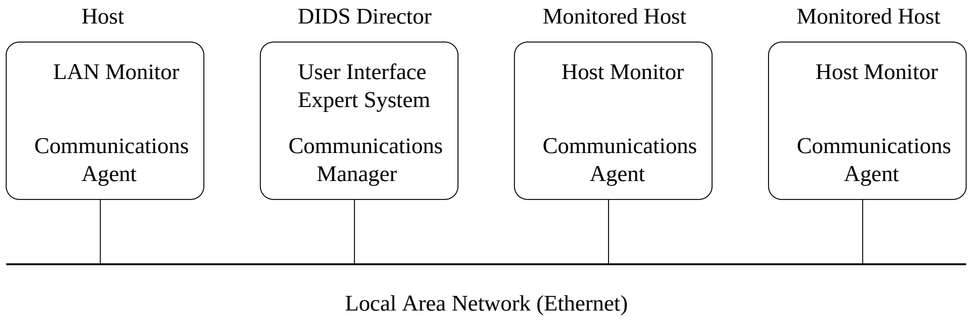

Figure 2: The Distributed Intrusion Detection System (DIDS).

To conclude this section, we describe the components of an elaborate IDS by way of an example. In the case of DIDS (see Figure 2), each monitored computer runs a Host Monitor program, which filters and analyzes the audit records associated with activity on that particular computer. The Communications Agent program on each monitored computer sends information to a central computer designated as the DIDS Director. Programs that run on the DIDS Director include: (1) an expert system to analyze the aggregated information from the monitored sources; (2) a communications manager to control the information flow for the entire system; and (3) a user-friendly interface for the SSO (System Security Officer). An additional computer runs the LAN Monitor program, which, like the NSM, monitors network traffic in the LAN. Like the monitored computers, the LAN Monitor communicates with the DIDS Director via a Communications Agent program. With this architecture, intrusions on individual computers in the LAN can be detected by the Host Monitor programs, while network intrusions such as the doorknob attack can be detected by the DIDS Director, using information from the LAN Monitor and each of the monitored computers.

3 SOFTWARE PLATFORM

Our testing methodology is based on simulating computer users—both intruders as well as normal users—while the IDS is running. We employ the UNIX package *expect* [23] to simulate users in our testing experiments. The *expect* package is based on another UNIX package called *Tcl* (Tool command language) [29]. Using the *expect* language, we can write scripts (similar to UNIX shell scripts) that include intrusive commands. For running the scripts, *expect* provides a script interpreter which issues the script commands to the computer system just as if a real user had typed in the commands.

The *Tcl* package provides an interpreter for a simple programming language that includes variables, procedures, control constructs such as “if” and “for” statements, arithmetic expressions, lists, strings, and other features. Recent versions of *Tcl* also provide string-matching commands for matching strings against regular expressions. The syntax of *Tcl* shares similarities with the syntaxes of the UNIX shells and Lisp. *Tcl* is implemented as a C library package.

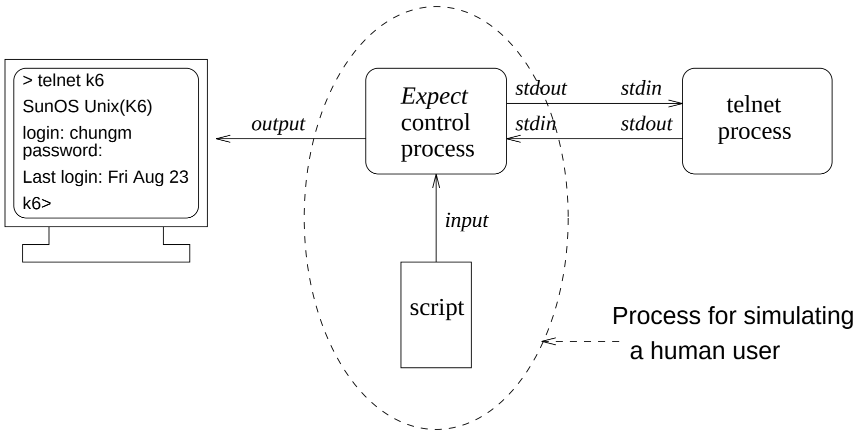

Figure 3: Simulation of a Human User’s Activity.

The *expect* package also provides a programming language interpreter. The core of the *expect* interpreter is the *Tcl* interpreter, but *expect* extends the *Tcl* command set to include several commands for controlling interactive programs. The command “spawn” creates an interactive process (such as *telnet*). The command “expect” waits to receive a specified string pattern (such as “login: ”) from the process. The command “send” sends a string to the process. Thus, a script containing several “expect/send” sequences and general programming constructs (such as variables, procedures, conditionals, and loops) can control an interactive session and thereby *simulate a human computer user*. Figure 3 illustrates the operation of *expect*. The following is a simple *expect* script that controls a brief *rlogin* session:

```
#Spawn an rlogin process.
spawn rlogin ComputerName -l UserName
#Expect the password prompt, then send the password.
expect {"Password:" send "ActualPassword \r"}
#Expect the shell prompt, then send commands.
#The shell prompt is specified in a regular expression.
expect {-re ".*%|.*>|.*#" send "whoami \r"}
expect {-re ".*%|.*>|.*#" send "ls \r" }
expect {-re ".*%|.*>|.*#" send "logout \r" }
```

The *expect* package by itself provides the capability to create a script to simulate a computer user. We have augmented *expect* with some additional commands that provide the capability to create *concurrent* scripts, complete with mechanisms for synchronization and communication among different scripts [35, 7]. These extensions to *expect* provide users with the ability to simulate concurrent intrusions, which were described in Section 2.2.

Often, in the course of testing an IDS, it may be necessary to *repeat* a particular test. For example, a test can be repeated to determine why (or why not) the IDS failed the test. Repeating the execution of a sequential test script is accomplished by simply running the script again with the same input. However, tests that involve concurrent script sets may be difficult to repeat exactly. Although each individual script is executed in a fixed order, the overall order of all of the events in all of the scripts may be non-deterministic. Unless the testers[3] employ some synchronization techniques, events are not *guaranteed* to occur in exactly the same order as in the original test.

event#		
1	User foo on host A telnets to host B	
2	foo logs in as foo on host B	
3	foo issues some commands	
4	foo logs out	
5		User foo on host C telnets to host B
6		foo logs in as foo on host B
7		foo issues some commands
8		foo logs out
	(a) Command Sequence 1	(b) Command Sequence 2

Figure 4: Two Command Sequences with no Interleaving.

As an illustration of non-deterministic execution, consider the execution of two command sequences shown in Figures 4 and 5. When two command sequences are executed concurrently,

[3]Throughout this paper we will use the term *testers* to refer to a group of people that are testing an IDS.

event#		
1	User foo on host A telnets to host B	User foo on host C telnets to host B
2	foo logs in as foo on host B	foo logs in as foo on host B
3	foo issues some commands	foo issues some commands
4	foo logs out	foo logs out
	(a) Command Sequence 1	(b) Command Sequence 2

Figure 5: Two Command Sequences with Interleaving.

different *interleavings* of the execution of commands in different sequences result in different overall execution sequences. In Figure 4, user foo from host C initiates the telnet connection to host B in command sequence 2 only after user foo logs out from host B in command sequence 1. On the other hand, in Figure 5, user foo logs in to host B from both host A and host C at the same time. Since it is unlikely that a legitimate user is logged in to a host from two different hosts at the same time, this execution sequence is more suspicious than the sequence in Figure 4. Thus, the IDS is likely to react differently to these two scenarios, even though all of the commands are the same in both scenarios.

To accommodate "reproducible testing" or "replay" [6, 22], deterministic execution of concurrent script sets is required. We have developed a synchronization mechanism to meet this requirement. The mechanism provides a means for the programmer to establish a *fixed order of execution for key events*, even if the events are associated with different scripts. Furthermore, the mechanism is flexible, so that the programmer can easily modify the order of events. This flexibility should be useful during testing experiments. The testers can run a concurrent script set several times, each time with a different synchronization specification, and they can then observe if the changes in synchronization affected the IDS's reaction to the commands in the scripts, or the IDS's ability to detect intrusions.

An additional feature that we have incorporated into our software platform is a "record-and-replay" capability. A user can type in a sequence of commands manually, and use the record feature to record that sequence. The "recording" can then be replayed at will, just like other scripts. Also, it is possible to set "synchronization points" in the recorded session, so that it can be *replayed in synchronization* with other recorded sessions. A limitation of the record-and-replay approach is that it does not directly support programmability. For instance, currently, it is not possible to insert a conditional statement into the recording at the time of recording. However, this limitation can be easily overcome by combining a programmed script with a recording. In summary, the record-and-replay feature highly facilitates the creation of simulation scripts for testing experiments. In particular, this feature should be most useful to system managers when they wish to create *site-specific* scripts to simulate *site-specific* intrusion techniques, which are described in Section 4.2.

4 TESTING ISSUES

Now, we begin the discussion of testing methodology by examining some key testing issues.

4.1 Performance Objectives for an IDS

The first step in the IDS testing methodology is to identify a set of performance objectives for an IDS. We have identified the following objectives (which are similar to the key design goals for developing an IDS cited in [19]):

- **Broad Detection Range:** for each intrusion in a broad range of known intrusions, the IDS should be able to distinguish the intrusion from normal behavior;
- **Economy in Resource Usage:** the IDS should function without using too much system resources such as main memory, CPU time, and disk space; and
- **Resilience to Stress:** the IDS should still function correctly under stressful conditions in the system, such as a very high level of computing activity.

An IDS should meet the first objective or else many intrusions will escape detection. The second objective is required because, if an IDS consumes too much resources, then using the IDS may be impossible in some environments, and impractical in others. Finally, the third objective is necessary for two reasons: (1) stressful conditions may often occur in a typical computing environment; and (2) an intruder might attempt to thwart the IDS by creating stressful conditions in the computing environment before engaging in the intrusive activity. For example, an intruder might try to interfere with the IDS by creating a heavy load on the IDS host computer. Thus, we claim that it is *necessary* (though perhaps not *sufficient*) for an IDS to meet these three objectives in order to be effective in a wide range of computing environments. Our testing procedures are designed to measure the effectiveness of an IDS with respect to these objectives.

At different sites, these objectives will have different relative values. A broad detection range may not be necessary if the IDS monitors a site that is protected from many attacks by other security mechanisms. For example, the site might use a firewall to block most incoming network traffic, and strict access control and authentication techniques to prevent abuse by insiders. Economy in resource usage may not be required at a site where security is a high priority and where computing resources exceed user needs. Finally, resilience to stress may be less important if controls in the computing environment (e.g., disk quotas and limits on the number of processes per user) prevent users from monopolizing resources. Thus, the most important objectives for a particular site should be identified by system administrators before an IDS is evaluated, and the IDS tests should be developed accordingly.

4.2 Test Case Selection

In our approach to testing IDSs, a test case is a simulated user session. While some of the tests require simulated "normal" sessions, most of the test cases are simulated intrusions. A key problem is to select *which* intrusions to simulate. The testers should first collect as much intrusion data as possible. For UNIX systems, [20] and [5] report that intrusion data can be obtained from various sources, such as CERT advisories, periodicals such as PHRACK and 2600, and the USENET [10], and also by analyzing the vulnerabilities detected by security tools such as COPS [11] and TIGER [30]. Next, assuming that the number of intrusions is too large to simulate all of them, the testers must partition the set of intrusions into classes, and then create a representative subset of intrusions by selecting one or more intrusions from each class. This technique is known in the software-testing field as *equivalence partitioning* [26]. Ideally, the classes should be selected such that, within each class, either the IDS detects each intrusion, or the IDS does not detect any intrusions [34]. Then, one test case from each class can be selected to represent the class in the final set of test cases. However, in general, it is difficult to identify perfect equivalence classes [34].

Now, we consider some possible strategies for classifying intrusions. Intrusions can be classified according to the intrusion technique. A comprehensive example of this type of classification is presented in [28]. A second strategy is to classify intrusions based on a taxonomy of the system vulnerabilities that the intrusions exploit (e.g., see [5, 21]). A limitation of using either of these strategies for the purpose of selecting test cases is that, even though two intrusions share the same classification, the IDS might detect one intrusion but not the other. In other words, neither of these classification schemes is likely to produce perfect equivalence classes. However, both of the schemes would ensure that a wide range of test cases would be selected.

A third strategy is to classify intrusions based on their *signatures*, which we defined in Section 2.4 as encapsulations of the identifying characteristics of specific intrusion techniques. A classification scheme based on signatures is presented in [20]. A limitation of using this classification strategy to select test cases is that the number of classes is small. However, this technique may possibly be extended to yield a finer-grained classification. This technique could also be extended to use information about the internal representation of signatures in a particular IDS.

Given the limitations of the three classification strategies with respect to selecting test cases, the set of test cases should be constructed by using all three strategies. Also, for each strategy, several test cases should be selected from each class. A natural extension to our work would be to develop a large set of test cases for various types of computer systems, which could be used for testing a wide range of IDSs.

As the final step in selecting test cases, the testers can supplement the set of test cases with some simulated intrusions that are of particular interest to the site at which the IDS will be employed. For example, in environments with strict policies governing computer use, some activities that would be considered normal at most sites are considered to be intrusions. The testers can create test cases based on such activities. As a second example, the testers may be aware of intrusion techniques that are not well-known. Simulations of these techniques should be included in the set of test cases.

4.3 Limitations

Our approach to testing IDSs has some limitations. First, the software platform that we use to simulate users cannot completely simulate the behavior of a user working with a GUI-based program, e.g., the X Window System. However, it is not always necessary to simulate an intruder's complete behavior. The intruder's activities generate some system activity, only a subset of which is related directly to the attack. The simulation tool must only be capable of causing *that subset* of system activity to occur. For example, while the simulation tool may not be able to simulate a user who moves the mouse pointer and then presses a mouse button to select a certain option from a GUI menu, it can still issue the same command to the computer system as the GUI. Even for cases in which the simulation tool cannot re-create the key intrusive activity, our testing methodology is still valid. The intrusion can be simulated using a different tool. In the worst case, the testers can simulate the intrusion manually.

A second limitation is that our methodology is designed to test systems that primarily perform misuse detection. However, some of the testing procedures can be adapted for testing IDSs that perform anomaly detection as well.

4.4 Using the Test Results

The test results can be used by the developers, users, and potential customers of an IDS to make the IDS more effective or to make a site more secure. A developer can use the results to find and correct weaknesses in the IDS. For example, if the tests show that the IDS is unable to detect a particular attack, the developer might enhance the language for describing attack signatures, so that the IDS could recognize that attack. Or, if the tests indicate that the IDS is consuming a large amount of resources (e.g., disk space), the developer might create a more efficient implementation that uses less resources. If nothing else, the developer might advertise the weaknesses revealed by the tests, so that users of the IDS can protect their sites by supplementing the IDS with other security tools. An IDS user (e.g., a system administrator) may employ the test results to identify configuration problems, which may occur when the IDS has many configuration options or when the configuration steps are complex. If instead the user detects problems with the IDS itself, then the user can seek additional tools to protect the computer system. Finally, a potential IDS customer can use the test results to compare IDSs and thereby select the one that will perform best in the customer's computing environment.

5 TESTING METHODOLOGY

We have developed a set of detailed procedures for testing an IDS. The procedures are designed for testing an IDS that monitors a network of computers, although some of the procedures can be directly applied to an IDS that only monitors a single computer. The best environment to use for these tests is an *isolated* local area network, because many of the tests require direct control over the amount of computing activity in the environment.

The installation and configuration of the IDS should be performed carefully. The testers should consult the IDS manuals to determine how to set up configuration files and how to select appropriate values for each configuration parameter[4]. The testing procedures may eventually reveal weaknesses in the IDS configuration, in which case the IDS should be reconfigured and tested again.

Most of our procedures are variations of the following basic testing procedure:

- create and/or select a set of test scripts;
- establish the desired conditions (such as the level of "background" computer activity) in the computing environment;
- start the IDS;
- run the test scripts; and
- analyze the IDS's output.

We have divided the test procedures into three categories, which correspond directly to the three performance objectives described earlier in Section 4.1. Several of the test procedures are adaptations of the "higher-order" software-testing methods described by Myers [26].

5.1 Intrusion Identification Tests

The two Intrusion Identification Tests measure the ability of the IDS to distinguish known intrusions from normal behavior. The first of these tests is the Basic Detection Test, which should be conducted as follows:

- create a set of intrusion scripts;
- as much as possible, eliminate unrelated computing activity in the test environment;
- start the IDS; and
- run the intrusion scripts.

[4]For example, to configure the NSM, the user must set up files that indicate the IP addresses of the monitored computers, and the names of the network services to be monitored. In addition, the user must specify in a file a list of strings that the NSM should use for pattern-matching against the monitored network traffic.

The testers can then analyze the IDS output. The specific analysis method depends on the *type* of information available in the output of the particular IDS. We will consider two examples. In the first example, the IDS output classifies each monitored session as "suspicious" or "normal." After conducting the test, the testers can simply calculate the percentage of intrusion scripts that were identified as suspicious.

In the second example, the IDS output consists of a numerical "warning value" for each session, such that a higher warning value indicates a more suspicious session. The testers should compare the IDS output from the test to a large sample of IDS output associated with monitoring normal users in the same computing environment. The testers can use standard statistical techniques to compare the warning values associated with the intrusion scripts to the warning values associated with the normal users. Ideally, the testers should find a statistically-significant difference between the two groups of values. If there is no sample of IDS output available for normal users, then that output can be generated by running normal-user simulation scripts while the IDS is active.

The Basic Detection Test indicates how well the IDS detects intrusions. However, there is a second component in the ability of an IDS to distinguish intrusions from normal behavior. Ideally, an IDS should rarely generate a false alarm by flagging normal behavior as "intrusive." The second Intrusion Identification Test, called the Normal User Test, measures how well an IDS meets this objective. The test is conducted in the same manner as the Basic Detection Test, except that the normal-user scripts are used instead of intruder scripts. The IDS output associated with the scripts should be examined to determine how often normal behavior is identified as suspicious. This measurement can be used to estimate how much time will be wasted in investigating false alarms if the IDS is to be used regularly. Returning to the two examples of IDS output described earlier in this section, we note that the Normal User Test is needed only for the first example. The analysis in the second example, in addition to indicating how well the IDS detects intrusions, should also indicate how often the IDS generates false alarms.

5.2 Resource Usage Tests

The Resource Usage Tests measure how much system resources are used by the IDS. The results of these tests can be used, for example, to decide if it is practical to run a particular IDS in a particular computing environment.

At this point, we have developed one Resource Usage Test: the Disk Space Test, which measures the disk space requirements of an IDS. A script that simulates a user who produces computer activity at a constant rate is required for this test. For example, the script might issue a sequence of commands repeatedly.

The procedure for the Disk Space Test is as follows:

- eliminate unrelated activity in the test environment;

- start the IDS;
- run the test script for a measured period of time (e.g., one hour); and
- calculate the total disk space used by the IDS to record the session associated with the script.

The test should be repeated several times using a range of different time intervals. Based on the group of tests, the testers can determine the relationship between disk space usage and monitoring time. For example, in the case of the NSM, disk-space usage increases in direct proportion to monitoring time. The tests should also be repeated using different numbers of simulated users, by running copies of the test script simultaneously. Then, the testers can determine the relationship between disk-space usage and the number of users monitored. The testers can use their analysis of all of these test results to predict the IDS storage space requirements when the IDS is monitoring several real users in the real computing environment. The testers can then compare several different IDSs based on such predictions.

5.3 Stress Tests

Stress Tests check if the IDS can be affected by "stressful" conditions in the computing environment. For example, an intrusion that the IDS would ordinarily detect might go undetected under such conditions. We have developed testing procedures for several different forms of stress.

5.3.1 Stress Test: Smokescreen Noise

We define *noise* to be computer activity that is not directly part of an intrusion. An intruder might attempt to disguise an intrusion by employing noise as a smokescreen. For example, an intruder on a UNIX computer might intersperse intrusive commands with normal programming commands, according to this model:

1. with some probability, do an *ls*[5] to check a file;
2. edit a file;
3. compile;
4. with some probability, do an *ls* to check a file;
5. with some probability, go back to Step 1;
6. execute the program; and

[5]The UNIX *ls* command lists the contents of a file-system directory.

7. with some probability, go back to Step 1.

The programming behavior can be used to disguise the *ls* and edit commands which the intruder may be using to examine some target files. Depending on the algorithm that the IDS is using, the IDS may not detect the intrusive behavior, because the overall behavior appears to be normal.

The first step in the Smokescreen Noise Test is to create suitable test scripts. One approach is to supplement a copy of each intruder script with a sequence of several “normal” commands between each pair of original commands. Then, the test should be conducted like the Basic Detection Test. The IDS output for each script should be compared to the corresponding IDS output from the Basic Detection Test. Ideally, this comparison will show that the IDS detects the same intrusions during each test. On the other hand, if the IDS detects a particular intrusion in the Basic Detection Test, but does not detect the same intrusion in the Smokescreen Noise Test, then that is an indicator of a weakness in the IDS. The testers should conduct further tests to determine the cause of the problem.

5.3.2 Stress Test: Background Noise

We define *background noise* to be noise caused by legitimate user activity. For example, an intrusion may occur during working hours, when there are several legitimate users logged in to the computer system. To prepare for the Background Noise Test, a set of noise scripts that generate continuous normal activity should be created. The scripts for the Normal User Test can be adapted for this test.

The testers should start running the noise scripts first. Then, the test proceeds just like the Basic Detection Test, in which each attack script is run one at a time. Again, the IDS output for each script should be compared to the corresponding IDS output from the Basic Detection Test, and differences may indicate a vulnerability in the IDS.

The testers should repeat this procedure several times, each time with a different amount of noise. Different numbers of copies of the same noise script (or script set) can be run at the same time to create different levels of noise. If possible, a *maximal* noise level should be used in at least one of the tests. For example, there might be a limit on the number of sessions that the IDS can monitor at the same time. In that case, the IDS should be forced to monitor as many noise-script sessions as possible.

5.3.3 Stress Test: High-Volume Sessions

The Volume Test checks how the IDS is affected by high-volume sessions. The definition of “high volume” depends on the IDS. For example, if the IDS monitors each command in user sessions, then a high-volume session would be a session in which a large number of commands are issued. A "volume script" that simulates a high-volume session should be created for this test.

The purpose of this test is to check if the IDS monitors the high-volume session correctly, and to check if the IDS can still correctly monitor other sessions at the same time. This test might detect, for example, a case in which the IDS runs out of main memory, and is physically unable to monitor all of the user sessions at once.

The volume script should be started first. Then, each intrusion script should be run one at a time. After each script has stopped running, the IDS output associated with the volume script should be analyzed carefully. Also, the IDS output associated with each intruder script should be compared to the IDS output from the Basic Detection Test. Differences may indicate that the IDS was affected by the volume script.

This test should be repeated, using different numbers of volume scripts running concurrently each time. As in the Background Noise Tests, a maximal level of volume should be used in at least one of the tests, in which the IDS should be forced to monitor as many volume-script sessions as possible.

5.3.4 Stress Test: Intensity

The Intensity Test checks if the IDS is affected by sessions in which a lot of activity is generated very quickly, and therefore the IDS's information source logs a lot of activity in a short time. First, a "stress script" that simulates such a session should be created. The script should simulate several consecutive user sessions, in each of which the simulated intruder logs in, carries out some intrusive activity, and then logs out. Such a script could be constructed by combining several of the scripts from the Basic Detection Test. *expect* includes a mechanism that allows the user to specify how quickly consecutive script commands will be issued [23]. Such a mechanism should be used for this stress script so that the commands are issued at a high rate.

The script should be run once. Then, a modified version of the script should be created, which generates the same commands, but at a much slower rate. After the slower script is run, the IDS output associated with the two scripts should be compared. Differences may indicate a weakness in the IDS. For example, due to the high rate of activity caused by the stress script, the IDS might "miss" some of the intrusive activity.

It should be possible to run several stress scripts concurrently. Then, the stress test can be repeated several times, each time with a different number of stress scripts running. This is important because the IDS may be able to cope with one high-intensity session, but perhaps it will make errors if it is forced to simultaneously monitor several high-intensity sessions. In each case, the IDS output associated with the stress scripts should be compared to the IDS output associated with the slower test script.

5.3.5 Stress Test: Load

The Load Stress Test investigates the effect of the load on the IDS host CPU[6].

This test is conducted in the same way as the Basic Detection Test, except that a high load should be established on the IDS host during the test. A high load can be created by running additional programs on the IDS host, so that the IDS program must share CPU time with the other programs. For a UNIX system, this effect can be enhanced by using the UNIX "nice" command to lower the "scheduling priority" of the IDS program while other CPU-intensive programs are running on the same host. This tends to decrease the percentage of CPU time allocated to the IDS program by the operating system of the IDS host.

The output from this test should be compared to the output from the Basic Detection Test. Differences may be evidence that the IDS is missing some intrusive activity because it is not running for a high-enough percentage of time on the CPU. This test should be repeated several times, each time with a different load on the IDS host.

6 EXPERIMENTAL RESULTS

We conducted testing experiments on the NSM. The NSM monitors all of the packets that travel on the LAN to which the NSM host[7] is connected. The NSM can associate each such packet with the corresponding computer-to-computer connection. It assigns numerical warning values to connections based on the contents of the packets, and on the likelihood of the connection occurring, based on a profile of recent connections. The warning values range from 0 to 10, such that a higher warning value indicates a more suspicious connection. We ran the NSM on a Sun SPARCstation 2 workstation connected to the Computer Science (CS) LAN segment at UC Davis (UCD).

6.1 Basic Detection Test

For this test, we used several different *expect* scripts, each designed to simulate a specific intrusive command sequence. Specifically, the scripts simulate these behaviors:

- browsing through a directory, using the *ls* command to list files, and an editor to view files;
- password-cracking;
- password-guessing using a dictionary file;

[6]In our testing experiments, we measure the load by using the UNIX *uptime* command, which reports the average number of jobs in the CPU run queue.

[7]We use the term *NSM host* to refer to the computer on which the NSM programs are running.

- door-knob rattling (password-guessing using common passwords);
- attempting to modify system files (e.g., */etc/passwd*);
- excessive network mobility (moving from computer to computer via *telnet* connections); and
- exploiting a vulnerability in a system program to obtain *super-user* status.

Each script establishes a *telnet* connection to another computer, sends a sequence of intrusive commands to the remote computer, and then closes the connection. The NSM monitored the execution of each of these scripts, and assigned a warning value to each connection. For comparison, we also set up the NSM to monitor regular traffic to and from a busy computer on the UCD CS LAN segment for several hours. Although some of this traffic could have been caused by intrusive behavior, we expect that most of it was caused by the legitimate activities of legitimate users.

Ideally, of course, the warning values for connections associated with known intrusive behavior would be high, and warning values for connections associated with normal, benign behavior would be low. Assuming that most of the connections to and from the busy computer were normal, the NSM succeeded in this case in assigning a relatively low warning value on average to these connections.

However, the NSM also assigned low warning values to some of the connections associated with the intrusive scripts. We determined, though, that this was caused by *our configuration* of the NSM. Like many IDSs, the NSM can be "tuned" so that it is sensitive to particular intrusive sequences of commands. Our experience illustrates how testing procedures can be used to uncover weaknesses in both the IDS itself *and* the IDS configuration.

6.2 Stress Tests

We hypothesized that stress in the form of a high load on the NSM host might affect the NSM's ability to monitor network connections. So, we performed a Load Stress Test on the NSM, based on the procedure described in Section 5.3.5.

We configured the NSM so that it would monitor the TCP (Transmission Control Protocol) packets associated with *telnet* connections to and from a specific computer ("Computer A") on the LAN. To create various levels of load on the NSM host, we used a "load script," which simply creates a *telnet* connection to the NSM host, and then issues a continuous sequence of UNIX shell commands. We measured the load using the UNIX *uptime* command, which reports the average number of jobs in the CPU run queue. We created higher levels of stress in successive tests by running several copies of the load script simultaneously. In addition, we added a second form of stress on the NSM. As described in Section 5.3.5, we used the UNIX *nice* command to lower the "scheduling priority" of the NSM program, so that the operating system would tend to allocate the NSM program a lower percentage of CPU time than it normally would otherwise. The scheduling priority of the NSM program was the same for each test.

For each test, we generated the desired load on the NSM host, and then we ran an "intrusion script" on Computer A. The script would establish a *telnet* connection from Computer A to another computer on the LAN, issue a sequence of several intrusive commands to the remote computer, and then close the connection. The intrusion script is a combination of several of the intrusion simulation scripts described in Section 6.1.

For each test, the NSM produced a report describing the connection established by the intrusion script. Ideally, the report would be identical for each test, because the same script was run for each test. However, the connection reports *were* affected by the increased loads on the NSM host. Apparently, the lower scheduling priority together with high loads on the NSM host caused the NSM program to *miss* some network packets. The NSM connection reports include the number of TCP data bytes missed for each connection. The NSM can calculate this number by monitoring the sequence numbers in the TCP headers. As indicated in Figure 6, the percentage of data bytes missed by the NSM tended to increase as the load on the NSM host increased.

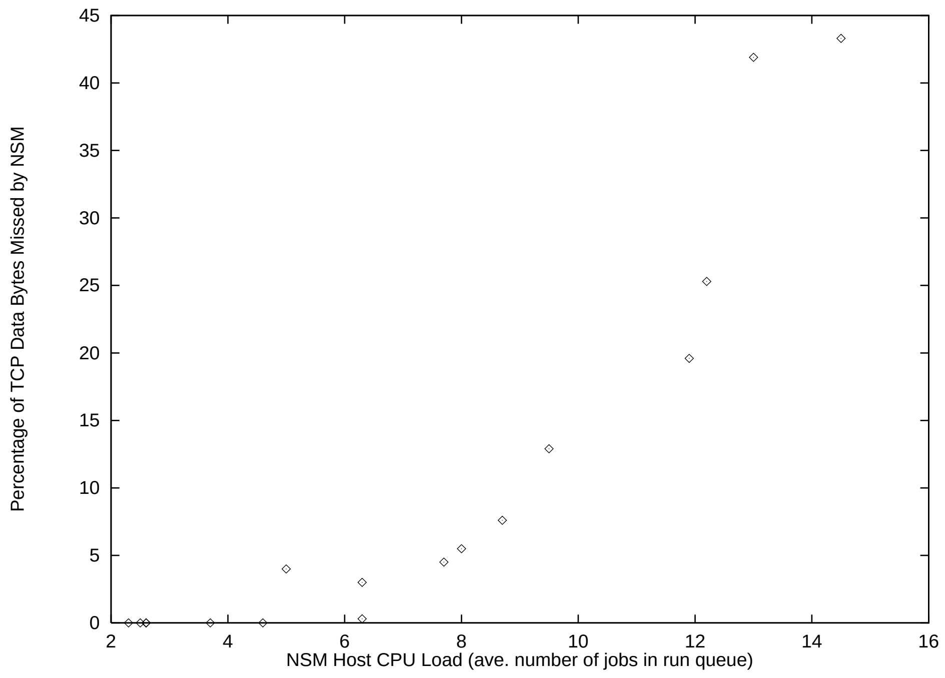

Figure 6: Bytes Missed by NSM vs. NSM Host CPU Load.

One possible explanation for the missed bytes is that the NSM program, when it is waiting to run due to the scheduling decisions of the operating system, can miss the transmission of some of the TCP data bytes. However, the monitoring operation involves several components of network hardware and software, and thus the actual explanation for the missed bytes may be more complex. The test results indicate, though, that an IDS can be affected by stressful conditions, and it may be potentially vulnerable to an attack by an intruder who knows how to exploit this weakness.

7 CONCLUSION AND FUTURE WORK

Our experimental results demonstrate that our testing methodology can reveal useful information about an IDS and its capabilities. As the growth in the use and development of IDSs continues, such testing techniques are growing in importance. Future work includes the careful development of a suite of intrusion test cases for the Basic Detection Test. We plan to develop additional performance objectives and tests for IDSs based on the related work of other groups. For example, tests that measure the *processing speed* of USTAT are described in [16]. Another task is to fine-tune the testing procedures and develop suitable metrics to create a "benchmark suite" for IDSs, similar in spirit to the well-established benchmarks such as SPECmarks, Livermore Loops, and Dhrystone, which are used to test the performance of various computer architectures. In the meantime, though, our tools and approaches can be used to systematically evaluate and measure the effectiveness and performance of IDSs. We expect that the development of such methods for assessing IDSs will also have a positive impact on the field of intrusion detection, since developers can use assessment results as feedback in the design process. We also expect that some of our testing techniques can be adapted to testing other software systems. In particular, the stress-testing techniques can be applied to testing computer operating systems and real-time control systems, and in general any software system which is required to cope with stressful conditions.

We have encapsulated our work into a package that includes sample test scripts, our enhancements to *expect* for concurrent scripts and for the record-and-replay feature, and complete documentation and installation instructions. Readers are encouraged to contact the authors if they are interested in acquiring this package.

Acknowledgements

We thank Becky Bace, our NSA project monitor, for her guidance and enthusiastic support of this project. We also appreciate the helpful comments from the anonymous reviewers and from our colleagues in the Computer Security Research Laboratory at UC Davis.

References

[1] D. Anderson et al., "Next Generation Intrusion Detection Expert System (NIDES)," Software Design, Product Specification, and Version Description Document, Project 3131, SRI International, July 11, 1994.

[2] R. G. Bace, Division of Infosec Computer Science, Research and Technology, National Security Agency, private communication, May 1995.

[3] S. M. Bellovin, "There Be Dragons," *Proc., Third USENIX UNIX Security Symposium*, Baltimore, MD, pp. 1-16, September 1992.

[4] S. M. Bellovin, "Security Problems in the TCP/IP Protocol Suite," *ACM Computer Communication Review*, vol. 19, no. 2, pp. 32-48, April 1989.

[5] M. Bishop, "A Taxonomy of UNIX System and Network Vulnerabilities," Technical Report CSE-95-10, University of California at Davis, September 1995.

[6] P. Brinch Hansen, "Reproducible Testing of Monitors," *Software–Practice and Experience*, vol. 8, pp. 721-729, 1978.

[7] M. Chung, N. Puketza, R. A. Olsson, and B. Mukherjee, "Simulating Concurrent Intrusions for Testing Intrusion Detection Systems: Parallelizing Intrusions," *Proc., 18th National Information Systems Security Conference*, Baltimore, MD, pp. 173-183, October 1995.

[8] D. E. Denning, "An Intrusion-Detection Model," *IEEE Transactions on Software Engineering*, vol. SE-13, no. 2, pp. 222-232, February 1987.

[9] C. Dowell and P. Ramstedt, "The COMPUTERWATCH Data Reduction Tool," *Proc., 13th National Computer Security Conference*, Washington, D.C., pp. 99-108, October 1990.

[10] D. Farmer and W. Venema, "Improving the Security of Your Site by Breaking Into It," USENET posting, December 1993.

[11] D. Farmer and E. H. Spafford, "The COPS Security Checker System," *Proc., Summer USENIX Conference*, pp. 165-170, June 1990.

[12] L. D. Gary, talk presented in "Crime on the Internet" session, 17th National Computer Security Conference, Baltimore, MD, October 12, 1994.

[13] L. T. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, and D. Wolber, "A Network Security Monitor," *Proc., 1990 IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 296-304, May 1990.

[14] J. Hochberg et al., "NADIR: An Automated System for Detecting Network Intrusion and Misuse," *Computers and Security*, vol. 12, no. 3, pp. 235-248, May 1993.

[15] K. Ilgun, "USTAT: A Real-Time Intrusion Detection System for UNIX," *Proc., IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 16-28, May 1993.

[16] K. Ilgun, R. A. Kemmerer, and P. A. Porras, "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 181-199, March 1995.

[17] H. S. Javitz and A. Valdes, "The SRI IDES Statistical Anomaly Detector," *Proc., IEEE Symposium on Research in Security and Privacy*, Oakland, CA, pp. 316-376, May 1991.

[18] S. Kumar and E. H. Spafford, "A Software Architecture to Support Misuse Intrusion Detection," Technical Report CSD-TR-95-009, Purdue University, March 17, 1995.

[19] S. Kumar and E. H. Spafford, "An Application of Pattern Matching in Intrusion Detection," Technical Report CSD-TR-94-013, Purdue University, June 17, 1994.

[20] S. Kumar and E. H. Spafford, "A Pattern Matching Model for Misuse Intrusion Detection," *Proc., 17th National Computer Security Conference*, Baltimore, MD, pp. 11-21, October 1994.

[21] C. E. Landwehr et al., "A Taxonomy of Computer Program Security Flaws," *ACM Computing Surveys*, vol. 26, no. 3, pp. 211-254, September 1994.

[22] T. J. LeBlanc and J. M. Mellor-Crummey, "Debugging Parallel Programs With Instant Replay," *IEEE Transactions on Computers*, vol. C-36, no. 4. pp. 471-482, April 1987.

[23] D. Libes, *Exploring Expect: A Tcl-based Toolkit for Automating Interactive Programs*, O'Reilly & Associates, Inc., 1994.

[24] T. F. Lunt et al., "IDES: A Progress Report," *Proc., Sixth Annual Computer Security Applications Conference*, Tucson, AZ, December 1990.

[25] T. F. Lunt et al., "A Real-Time Intrusion Detection Expert System(IDES)," Interim Progress Report, Project 6784, SRI International, May 1990.

[26] G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., 1979.

[27] B. Mukherjee, L. T. Heberlein, and K. N. Levitt, "Network Intrusion Detection," *IEEE Network*, vol. 8, no. 3, pp. 26-41, May/June 1994.

[28] P. G. Neumann and D. B. Parker, "A Summary of Computer Misuse Techniques," *Proc., 12th National Computer Security Conference*, Baltimore, MD, pp. 396-407, October 1989.

[29] J. K. Ousterhout, *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.

[30] D. R. Safford, D. L. Schales, and D. K. Hess, "The TAMU Security Package: An Ongoing Response to Internet Intruders in an Academic Environment," *Proc., Fourth USENIX UNIX Security Symposium*, Santa Clara, CA, pp. 91-118, October, 1993.

[31] M. M. Sebring, E. Shellhouse, M. E. Hanna, and R. A. Whitehurst, "Expert Systems in Intrusion Detection: A Case Study," *Proc., 11th National Computer Security Conference*, Baltimore, MD, pp. 74-81, October 1988.

[32] S. E. Smaha, "Haystack: An Intrusion Detection System," *Proc., IEEE Fourth Aerospace Computer Security Applications Conference*, Orlando, FL, December 1988.

[33] S. Snapp, J. Brentano, G. Dias, T. Goan, L. Heberlein, C. Ho, K. Levitt, B. Mukherjee, S. Smaha, T. Grance, D. Teal, and D. Mansur, "DIDS (Distributed Intrusion Detection System) – Motivation, Architecture, and An Early Prototype ," *Proc., 14th National Computer Security Conference*, Washington, D.C., pp. 167-176, October 1991.

[34] E. J. Weyuker and B. Jeng, "Analyzing Partition Testing Strategies," *IEEE Transactions on Software Engineering*, vol. 17, no. 7, pp. 703-711, July 1991.

[35] K. Zhang, *A Methodology for Testing Intrusion Detection Systems*, M.S. Thesis, University of California at Davis, May 1993.