USING SEQUENCE ANALYSIS TO PERFORM APPLICATION-BASED ANOMALY DETECTION WITHIN AN ARTIFICIAL IMMUNE SYSTEM FRAMEWORK

THESIS

Larissa A O'Brien, First Lieutenant, USAF
AFIT/GCS/ENG/03-15

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/03-15

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, Department of Defense or United States Government.

AFIT/GCS/ENG/03-15

USING SEQUENCE ANALYSIS TO PERFORM APPLICATION-BASED ANOMALY DETECTION WITHIN AN ARTIFICIAL IMMUNE SYSTEM FRAMEWORK

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

In Partial Fulfillment of the Requirements for the

Degree of Master of Science (Computer Science)

Larissa A. O’Brien, B.A.

First Lieutenant, USAF

March 2003

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT/GCS/ENG/03-15

USING SEQUENCE ANALYSIS TO PERFORM APPLICATION-BASED ANOMALY DETECTION WITHIN AN ARTIFICIAL IMMUNE SYSTEM FRAMEWORK

Larissa A. O'Brien, BA
First Lieutenant, USAF

Approved:

Gregg H Gunsch 7 MAR 03
Gregg H. Gunsch (Chairman) date

Gary B. Lamont 7 MAR 03
Gary B. Lamont (Member) date

Henry B. Potoczny March 7, 2003
Henry B. Potoczny (Member) date

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The Air Force and other Department of Defense (DoD) computer systems typically rely on traditional signature-based network IDSs to detect various types of attempted or successful attacks. Signature-based methods are limited to detecting known attacks or similar variants; anomaly-based systems, by contrast, alert on behaviors previously unseen. The development of an effective anomaly-detecting, application-based IDS would increase the Air Force's ability to ward off attacks that are not detected by signature-based network IDSs, thus strengthening the layered defenses necessary to acquire and maintain safe, secure communication capability.

This system follows the Artificial Immune System (AIS) framework, which relies on a sense of "self," or normal system states to determine potentially dangerous abnormalities ("non-self"). A method for anomaly detection is introduced in which "self" is defined by sequences of events that define an application's execution path. A set of antibodies that act as sequence "detectors" are developed and used to attempt to identify modified data within a synthetic test set.

USING SEQUENCE ANALYSIS TO PERFORM APPLICATION-BASED ANOMALY DETECTION WITHIN AN ARTIFICIAL IMMUNE SYSTEM FRAMEWORK

1. Introduction

The Air Force and other Department of Defense (DoD) computer systems typically rely on traditional signature-based network IDSs to detect various types of attempted or successful attacks. Information protection is a great concern of the DoD; even though no classified data is supposed to be kept on computers connected to the internet, penetration of an unclassified system could result in the compromise of sensitive data that could be used to negatively impact DoD personnel and resources. With applications such as the Air Force Portal making it possible to consolidate data from varied sources, the defensive information warrior must also consider the possibility of data aggregation leading to the presence of classified content on an inadequately protected, publicly reachable system.

The development of an effective application-based IDS would increase the Air Force's ability to ward off attacks that are not detected by signature-based network IDSs. This increased ability would strengthen the layered defenses necessary to acquire and maintain safe, secure communication capability.

1.1 Overview of Threat

An intrusion can be defined as "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource." [Head90] An attacker attempts to subvert system security to gain access to information, alter information, or deny use of the system to other users [Cros95a]. Most exploits used by attackers take advantage of a fault or vulnerability that is known to be present in a target system.

1.2 Overview of IDSs

An intrusion detection system (IDS) can be a valuable contributor to the defense of a computer system. The term "intrusion detection" is something of a misnomer; the term is not appropriately descriptive. The purpose of an IDS is to detect behavior that could result in inappropriate access, modification, or destruction of system resources. A pure IDS is a reactive form of defense; it does not attempt to stop an intrusion, merely creates an alert that can be investigated by an analyst.

1.3 Research Focus

This thesis investigation appropriates methodologies for application-based anomaly detection using event sequence pattern detection. Most intrusion detection systems (IDSs) currently in use utilize signature-based methods that are limited to detecting known attacks or similar variants. Anomaly-based systems, by contrast, alert on behaviors previously unseen. The system follows the Artificial Immune System (AIS) framework, which relies on a sense of "self," or normal system states, to determine potentially dangerous abnormalities ("non-self"). The majority of research on anomaly-based intrusion detection has involved network-based IDSs. The incidence of attacks wherein the attacker disguises an attack to subvert a network-based IDS is on the increase. As part of a defense-in-depth approach, it is important to develop models that perform anomaly-based intrusion detection at the application level. An application-based system offers the advantages of being able to be tailored to the limited set of execution paths that can occur. This research is accomplished by simulating the output of an application that has been instrumented with code that documents the order of a series of "events." Antibodies are developed to use the sensor information as inputs to characterize both the "normal" process execution and abnormal execution (as from attacks/exploits).

1.3.1 Scope

Application-based intrusion detection is necessarily limited to analyzing data from input to the application and any events that occur as a result of the input. This research focuses primarily on normalcy and anomaly of execution paths, although the model presented is extensible to other monitorable event sequences.

1.3.2 Hypothesis

The hypothesis is that event sequence descriptors can be used to detect anomalies in application paths using genetic programming techniques within an Artificial Immune System framework.

1.3.3 Objectives

The primary objective of this study is to develop an IDS for abnormal process execution that can effectively differentiate between normal and anomalous *sequences* (ordered sets) of events. The sequence may be interrupted by irrelevant events, and the logic behind the ordering is not known *a priori*.

This objective is multipart:

It is necessary to define a set of operators that is adequate to characterize the relationships among a set of events that occur during process execution.

> ***Objective I**: Identify and implement any operators necessary to describe temporal relationships among relevant events*

These relationships may be of varying complexity. In one case, the occurrence or disappearance of a single event may be enough to characterize the effects of an intrusion; in another, a set of relations may need to exist where any one of them in isolation would be within the set of normalcy. Therefore, once a set of operators is defined, a plausible technique for combining them so as to be able to characterize more complex effects must be determined.

> ***Objective II**: Determine an appropriate method of producing descriptors for event sequences of varying complexity*

Once the descriptors have been produced, a process is needed to categorize the descriptor as describing a normal sequence or an anomalous one.

> ***Objective III**: Determine an appropriate method of classifying these relationships as “normal” or “anomalous”*

Intrusion detection is often considered in isolation from the actions that must be taken subsequent to detection. Once an intrusion has occurred, there is a limited amount of data available from which a human analyst must draw conclusions. A problem inherent to anomaly-based ID is that it is difficult or impossible to differentiate between benign and malignant anomalies. This analytical step generally must be performed by human analysts post-mortem, and is both time-consuming and tedious.

***Objective IV**: Determine a method to extract and consolidate data that may assist human analysts in locating the point at which an anomaly occurs.*

1.3.4 Approach

A set of software "sensors" can be placed within the code of an open-source application. Each "canary" sensor is designed to issue an "alert" when the section of code in which it is embedded is reached, thus locating the point in the program flow. A sequential log represents the order of triggered sensors as a one-dimensional array of size *m* where log entry[m] represents the mth sensor to trigger. The assumption is that a sufficient number of sensors placed at selected locations can provide enough information for an analyst to detect inappropriate activity by performing temporal analysis on the output. This research assumes the pre-placement of sensors and the existence of an output log (an appropriate method of simulating these logs is described in Chapter 3). A set of "self" (normal) logs are used to train the system using the principle of negative selection; from these logs, a genetic programming engine can fill an antibody database consisting of non-self event sequence descriptions. These antibodies are then deployed on a test set composed of both self and non-self (abnormal) logs, resulting in a classification of each log. Analyst evaluation of the results creates a feedback loop to update the antibodies. Potential system utility can be inferred based on the accuracy of these results.

1.3.5 Assumptions

Several assumptions are:

1. It is assumed that each application contains an essential "skeleton" of events of limited number, such that a basic "template" describing all possible sequences of events can be characterized by a regular expression. The words produced when the regular expression is fully enumerated compose the regular language of normal event sequences.
2. It is assumed that exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations can be detected from abnormal patterns of usage [Denn87].
3. It is assumed that certain categories of intrusion attacks result in a change from the standard execution path(s) and/or affect the timing of progress through the path. This assumption is further explained in Chapter 2. If assumptions 1 and 2 are accepted, then it is realistic to assume that these intrusions should be detectable if a reliable way of characterizing the patterns of events within the execution paths is determined.

4. It is assumed that a basically stable environment exists regarding the instrumented application. This assumption is reasonable because a single application has a limited set of appropriate inputs that results in the execution of a limited set of paths.

1.4 Document Overview

This chapter describes the motivation and primary objectives for this research: to provide a "proof of concept" for an application-based IDS, using the AIS paradigm and incorporating genetic programming techniques, that is capable of classifying event sequences as normal or anomalous.

Chapter 2 describes essential background research and literature required to better understand the problem and prior solution methods. Chapter 3 provides a high-level overview of the methodology used in this research, while Chapter 4 discusses design and implementation details. Chapter 5 describes experiment design and results. Chapter 6 summarizes major conclusions and provides a description of proposed future work.

2. Background and Literature Review

2.1 Introduction

In this chapter, background material and relevant prior research that establish the foundation for this thesis effort are presented. Section 2.2 discusses intrusion detection systems. Section 2.3 discusses evolutionary algorithms (with emphasis on genetic programming and artificial immune systems). Section 2.4 discusses system and application vulnerabilities, while Section 2.5 discusses various types of attacks that are enabled by these vulnerabilities. Presented in Section 2.6 are several interesting analytical methods that can be used to monitor the effects of attacks. Section 2.7 describes a method for generating anomalies for system testing. A description is given of each concept and how it relates to the thesis research.

2.2 Intrusion Detection Systems

In general, IDSs are of three types, *network-based*, *host-based,* and *application-based*, and use one or both of two detection methods, *signature* and *anomaly*. Network-based detection involves watching network traffic for patterns of suspicious activity. Analysis may be performed for each host or at a centralized location that attempts to correlate activity across the network. Host-based detection involves monitoring the activities of users and/or processes on a single machine [Guns00]. Application-based detection is a subset of host-based detection in that it involves monitoring processes at the host level, but is considered a separate type because of the difference in focus. Signature (or *misuse*) detection is a problem of matching patterns of activity, network or system-level, against a database of known attacks. Any attacks with patterns that match those in the database can be identified prior to reaching the host. Most IDSs deployed currently are signature-based. Adversaries, knowing that signature-based IDSs are installed on government networks, continue to develop and deploy new attacks that evade detection; it is to detect these new attacks that anomaly-based IDSs are necessary. Anomaly detection is subtler and requires an IDS to identify events or patterns of event occurrences that are unprecedented in the system. Not all anomalies indicate intrusions, however; some form of classification system is needed to attempt to differentiate between these two sets.

Each type has certain inherent advantages and disadvantages. Both network and host-based systems face the problem of balancing the amount of data gathered and processed against desired speed. Network-based systems have the advantage of being able to screen packets before they can do harm to the system. Host-based systems have the advantage that they can detect inappropriate behaviors caused by exploits that have been altered to evade network-based detection. Host-based systems also do not have the problem that network-based systems do with encrypted packet content. If an attacker attempts to use an attack with a specific signature that involves part of the packet header, it may be more efficient to catch this attack at the network level, thus preventing it from completing. However, this method is ineffective against attacks if the attacker alters the code enough to destroy the signature, or disguises the signature through the use of encryption or mutation engines. Also, a new attack, until it is analyzed, may have no known signature, in which case the packets slip past the network-based IDS. At this point, the only hope for the system is that the attack is caught by a host-based system. Even the host-based system may fail to catch the attack if it designed only for misuse detection and is not attempting to detect anomalous behavior. These examples illustrate that a secure defense system may need to incorporate both types and methods of intrusion detection.

Research Impact

This research is limited to anomaly-based, application-based intrusion detection. It is important to have some sense of the limitations of this method. Ideally, such a system would be paired with an efficient signature-based, network-based IDS that would serve to identify the majority of attacks before they reached the host. The system would be responsible for attacks that are undetectable by that layer of defense.

2.3 Evolutionary Algorithms (EAs):

Evolutionary Algorithms are stochastic search techniques that use computational models of evolution [Heit00]. Stochastic searches are used in problem domains where the search space is too large to cover deterministically. A stochastic search is not guaranteed to find an optimal solution for a particular problem; however, a well-designed stochastic search can often find satisficing solutions in reasonable time [Mich02]. Functions inspired by biological genetics are used within an "artificial selection" framework that results in a form of guided evolution. The term "genotype" is used in biology to refer to the genetic material of an individual – its "blueprint" for construction; with EAs the genotype is the software representation of an individual solution candidate. This is the level at which the data structure can be manipulated by operators. The term “phenotype” is used to refer to the functional interpretation or evaluation of an individual [Foge95]. The Subclasses of EAs include genetic algorithms, genetic programming, evolutionary programming, evolutionary strategies, and classifier systems; despite their differences in representation, selection methods, and mutation operators, they are all related by the use of a model of biological evolution [Heit00].

Research Impact

This concept is relevant because an anomaly-based IDS requires some method for differentiating between self (appropriate use of an application) and non-self (attack). EA techniques can be used to automatically define or refine a model of normal (or abnormal) usage as well as to determine classification of unknown data.

2.3.1 GAs

Rather than explicitly creating a solution for a given task, the goal of the programmer is to design a system that rewards fitness of solutions. These systems use biologically inspired functions and artificial selection to “evolve” a solution to a predetermined problem. The system generates an initial population of “chromosomes.” In GAs, these chromosomes are fixed-length binary or character strings. With each generation, each individual in the population is evaluated and assigned a fitness value according to how well it accomplishes the task. Pairs of “parent” chromosomes are selected from the population based on their fitness, and their genetic code is manipulated, creating a new generation of chromosomes. This process continues until an adequate solution to the problem is found. Generally, the manipulation operators used are recombination (or *crossover*) and point mutation. [Heit00]

2.3.2 GP

Genetic Programming (GP) is similar to GA; however, in GP, the chromosomes are composed of operators and operands (*terminals*) that usually represent a simple meta-language tailored for a particular problem [Koza92]. GP uses recombination, and, to a lesser degree than GAs, point mutation. The result of evolution in this case is a set of programs that can be run within an interpretive system. GP solutions are not generally fixed-length, and the chromosome is often represented as a tree. As a result, they are more flexible in the range of their representations [Koza94]. Crosbie and Spafford have done exploratory work with applying GP to intrusion detection; however, their agent-based system relies on network data, not host-based data. [Cros95a, Cros95b, Cros95c]

Research Impact

The difference between GA and GP is critical to this research. IDS solutions utilizing a GA-based search engine are generally limited to examining data within a fixed window size proportional to the size of the GA chromosome. GP, with its function set and differently sized representations, is much more amenable to being used to examine data within a wide range of window sizes. This difference makes the GP search space larger, but also makes the solutions potentially more descriptive and useful. For this research, these benefits are used to both detect patterns over the course of an execution path *and* provide data about the patterns to analysts for forensic purposes.

2.3.3 Artificial Immune Systems

Artificial Immune Systems (AISs) include any machine learning system that applies biological immunology strategies to a problem. According to [Nune00], J. D. Farmer was the first to incorporate aspects of the immune system model into artificial intelligence techniques in his paper “The Immune System, Adaptation, and Machine Learning;” however, Stephanie Forrest first demonstrated its applicability to anomaly detection in IDSs. A biological immune system develops antibodies that are selected based on their ability to distinguish between “self” (cells, molecules of the body) and “non-self” (antigens - any foreign material). A computer intrusion detection system can be designed after this model [Dasg98]. The system must be able to form descriptions of “self” – normal events (system behavior, network traffic, etc) [Forr92]. A representation of an event pattern corresponding to an antibody is generated and compared to the self descriptions. If it matches, it is destroyed through Negative Selection; if it does not match, it is a potential identifier of non-self events. Once enough of these antibodies are generated, they are released into the system. If any antibodies find a match, it is considered a non-self antigen of some sort and is flagged as a potential intrusion [Forr97]. It can be seen that there are inherent limitations to AISs that are not present in biological immune systems, the most important of which is the inability to exploit the massive true parallelism available to autonomous antibodies within a living organism;

however, it is still a useful model for intrusion detection. One method used by Forrest and Hofmeyr involved determining "normal" for UNIX processes was accomplished by correlating system calls within a fixed-size time window [Forr96, Hofm98]. Research at AFIT has also involved AISs, most notably the Computer Defense Immune System (CDIS) [Will01], a descendant of the Computer Virus Immune System (CVIS) [Harm00]. CDIS uses antibodies to detect single packet attacks on a network, using a database of known "normal" traffic to define self. CDIS also uses a distributed AIS architecture.

Research Impact

The AIS paradigm is used in this research by training the system on a set of "self" data representing logs created as a result of normal application use. Testing is accomplished by using a data set consisting of previously unseen normal logs as well as modified logs representing non-self.

2.4 Types of Vulnerabilities

Aslam, Krsul and Spafford document a fault classification scheme for UNIX processes based on faults detected using software engineering methods [Asla96]. The scheme describes the following types of faults:

1. **Boundary condition errors**: These errors caused by inputs at the boundaries of the acceptability range can be detected with test cases using Boundary Value Analysis for functional testing of modules.

2. Input Validation Errors: These are errors caused as a result of a module failing to validate input from another module or process. They can be detected with syntax testing to validate format or path analysis to detect inappropriate execution paths

3. Access Validation Errors: These errors result from incorrectly specified condition constructs. They can be detected with path analysis or Branch and Relational Operator testing.

4. Failure to Handle Exceptional Condition Errors: These errors include unanticipated return codes and failure events. They can be detected by path analysis testing on critical sections of code.

5. Environment errors: These errors are more difficult to define and test, as they are dependent on idiosyncrasies specific to a particular machine, OS, or configuration. Mutation testing has been successfully applied to detect some of these problems.

6. Synchronization Errors: These errors result from improperly timed operation in the when a specific temporal relation is critical.

7. Configuration Errors: These are faults introduced after the software has been developed, during the maintenance phase. Static audit tools are of some use in detecting these errors.

The Fault Classification Scheme is as follows [Asla96]:

1. **Coding faults**

Definition: faults that were introduced during software development

a. **Synchronization errors**

Definition: a fault resulting from an exploitable timing window, or a fault resulting from improper serialization of operations

b. **Condition validation errors**

Definition: faults resulting from those cases where (1) a condition is missing, (2) a condition is incorrectly specified, or (3) a predicate in the condition expression is missing

2. **Emergent faults**

Definition: faults where the software performs according to specification but still causes a fault

a. **Configuration errors**

Definition: faults that result from (1) a program being installed in the wrong place, (2) a program installed with incorrect setup parameters, or (3) a secondary storage object or program installed with incorrect permissions

b. **Environment faults**

Definition: faults that result from insufficient attention being paid to the runtime environment, or faults that occur when modules interact in an unanticipated manner.

Note that while this classification system was designed to describe UNIX process faults, they are general enough to apply to non-UNIX applications as well. Most exploits take advantage of one of these faults that is known to be present in a target system.

Research Impact

While this information is not directly used in the implementation of the research system, the body of data regarding faults and fault testing would be invaluable to the researcher who is attempting to "fine-tune" a set of sensors within an application. A knowledge of common faults could allow for extra-dense sensor placement in areas that might potentially be vulnerable, although this would be time-consuming if not automated (see Zamboni's work, Section 2.6.3).

2.5 Types of Attacks

Heady et. al. define an intrusion as "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource." [HLMS90] This definition can be elaborated upon:

> *Confidentiality*, *integrity*, and *availability* are the three Critical Information Characteristics of the Information Systems Security Model. Confidentiality is the assurance that access controls are enforced. Integrity ensures the accuracy, relevance, and completeness of data. Availability ensures that information is provided to authorized users when it is needed. These characteristics represent the full spectrum of security concerns in an automated environment [NSTI94]. An attacker attempts to subvert system security to gain access to information, alter information, or deny use of the system to other users [Cros95]. The impact of an attack can be determined by examining how it impacts these three characteristics. Several examples are presented.

Scanning

A successful scan may reveal information about the topology of a network. Such information is useful to the attacker, who uses it to determine which systems present the most tempting targets, and which exploits might be effective against them. The nmap tool, for example, is used to identify live systems, perform operating system guessing, and scan for open ports on a system. [Scam01]. There are defensive steps that can be taken to eliminate or mitigate the success of such scans, but this becomes more difficult and resource-consuming as the scans become more sophisticated and complex. Scanners are important weapons in the arsenal of the attacker, but they are not in themselves sufficient to violate any of the three Critical Information Characteristics.

Denials of Service (DoS)

It is generally much easier to disrupt the operation of a network or system than to gain access or escalate privileges on a system. There are several types of DoS attacks, including bandwidth consumption, resource starvation, application crashing, and manipulating routing tables and domain name servers [Scam01]. These attacks primarily affect the *availability* of the system, although application crashing may lead to a loss of data (integrity) and manipulating routing data is a violation of integrity and, perhaps, confidentiality as well. The commonest DoS attacks are the ones that limit themselves to bandwidth consumption and resource starvation.

Buffer Overflows

A buffer is a contiguous block of memory. A buffer overflow occurs when a value is read or pushed into a buffer that is too small for it, resulting in the overwriting of data. A buffer overflow exploit involves crafting a value such that when the overflow occurs, control data or variables are overwritten with the desired data. Buffer overflows come in two flavors, stack-based and heap-based, although stack-based exploits are by far the most common. Over 60% of the advisories issued by Carnegie Mellon University's Computer Emergency Response Team in 2001 dealt with this kind of exploit [Ent01]. A successful stack-based buffer overflow exploit allows arbitrary code to be executed, usually for the purposes of escalating privileges. In the UNIX environment, exploits are run on applications running with root privileges, or on applications that are root-owned and have the setuid bit set ("set user ID on execution" – allows $user_1$ to execute a program owned by $user_2$, with $user_2$'s privileges). As root, the attacker has complete control of the system. Once the exploit is run, the process on which it is being run generally crashes. This situation represents a violation of confidentiality (the attacker has subverted access controls to achieve access to every piece of data on the system), integrity (the attacker can arbitrarily modify elements of the system), and availability (the attacker can take down any part of the system). Thus it can be seen that buffer overflow exploits represent a serious problem to computer security.

Race Conditions

A race condition can be defined as anomalous behavior caused by the unexpected dependence on the relative timing of events. Put simply, one occurs when a privileged process opens a vulnerability with the implicit assumption that it will be able to close the vulnerability before it can be exploited; an operation is treated as atomic when it is in fact not [Bish96]. In certain situations, a race condition can be exploited for the purpose of escalating privileges or to access protected data. Bishop and Dilger note that most of these attacks exploit flaws in a privileges program, or concurrent execution with a privileged and unprivileged program; few exploit operating system flaws. They were able to characterize various race conditions by describing them as “a minimal set of environmental information and a minimal sequence of actions [that result] in a breach of security.” Specifically, they identified the following conditions that indicate the possibility of a race condition:

- Two sequential system calls refer to the same object using a file path name
- A system call refers to an object by name and the second call uses a file descriptor, and the first call is *not* a call that maps a file path name to a descriptor

They tested for the presence of race conditions by using a analyzer program to scan code for such sequences, then providing the information to a human analyst who evaluated the environmental conditions [Bish96]. This method is a preventative; it does not detect attacks, but instead attempts to identify (and subsequently eliminate) the conditions which enable attacks.

Research Impact

For the purposes of this research, only intrusions that can lead directly to a breach of confidentiality or integrity regarding the information contained on a host are considered.

2.5.1 Effect of Intrusions

There are myriad indicators that an attack is occurring or has occurred, if the appropriate monitors are in place to identify the symptoms. This is not to say that it is reasonable or even possible to monitor all effects. A few examples of effects can be provided to give the reader an idea of the scope of the intrusion detection problem. The abstract term "event" is used to describe a range of actions that cause or indicate a change in the state of a system [Kunz93]. Thus, the term "events" can encompass system calls, libraries accessed, sections of code executed, change of variable value, or change in resource load. From the discussion of attacks above, it can be seen that the effects of most attacks can be generally grouped into one or more of the following categories:

- Unusual or improper sequencing of events
- Additional unusual events
- Missing events
- Unusual time delay between events

The implication attached to all categories is that the abnormalities exist *with respect to normalcy*. If an IDS cannot differentiate between normalcy and attack, then it is ineffective.

Research Impact

This concept is relevant because the research system should theoretically be extensible to detecting any of these attack effects with respect to any monitorable event types, even though only execution paths are represented for the purposes of scope.

2.6 Methods of Monitoring Event Relationships

The four categories in Section 2.5.1 can be condensed further into the concept of "relationships among events." Techniques that can be utilized to characterize or identify such relationships follow:

2.6.1 Path Profiling

Path profiling techniques are useful for characterizing execution paths of an application. A common application of path profiling is for software testing, where it is important to evaluate a program against a representative set of test data. Effective path profiling can identify unexecuted statements or control flow. Ball and Larus present a technique that uses a spanning tree to efficiently instrument a program such that path encoding is compact and minimal [Ball96]. Their implementation, PP, runs exclusively on SPARC-based machines; however, the same techniques that are used in their algorithm could be useful to characterize intrusion types that affect the execution path of an application.

Research Impact

Path profiling using a technique such as that encoded in PP would be an excellent way to perform basic application instrumentation. Using this method would ensure at a minimum that the various paths were represented, even if all the sensors required to detect anomalies were not placed.

2.6.2 Temporal Signatures

Recently, Doyle et. al. have proposed an approach based on an event-characterization language that incorporates and extends signature and anomaly methods. This method combines knowledge about activities, temporal regions, and environmental information to define a "trend." Their recognition system can use this trend template both for the purposes of identification and in explanation processes [Doyl01a, Doyl01b].

Research Impact

Timing changes can be a way to detect intrusions. For example, a buffer overflow may result in no significant difference in execution path, while creating a significant time delay while the attack code is executed. The practicality of describing event timing is discussed in Chapter 6.

2.6.3 Embedded Sensor Protection (ESP)

Zamboni presents in his thesis the Embedded Sensor Protection (ESP) system, in which he uses small pre-positioned pieces of code within the OpenBSD operating system to set flags during runtime. Based on the sensors that are set, he demonstrates that patterns can be found that allow for the identification of certain attacks on the host [Zamb01]. While the idea is intriguing, his implementation presents several problems. First, all sensors must be individually hand-coded, and their placement individually determined based on analysis of the open source code. This method is not only time-consuming but would clearly not be applicable to the more common scenario wherein one is faced with developing a defensive system for a host running an OS that does not have publicly available source code. Furthermore, all his sensors are developed and placed based on knowledge (acquired during vulnerability research and analysis of the code) of specific vulnerabilities known to be exploitable by a would-be intruder. These specifically–tailored sensors cannot be built without *a priori* knowledge of these logical flaws. It follows logically that if the exact positioning of every flaw in a selection of code was known, the vulnerabilities could easily be fixed. A relatively secure system could thereby be created with much less effort than is required to write the sensors, test the system against known attacks, and perform pattern matching to attempt to identify similar attacks in the future.

Research Impact

The ESP system demonstrated the utility of using an embedded sensor system. This research to a certain extent builds on the ESP system conceptually by attempting to extend the utility of such a system to the case where each sensor is not hand coded and is not specifically placed at fault points.

2.7 Generating Anomalies

Testing an anomaly-based system requires, obviously, a set of anomalies as test cases. Previous research has used both real attacks and artificial, generated anomalies. Both techniques have associated benefits and limitations.

Real Attacks

The most intuitive reason for testing against real attacks is that they are accurately descriptive of the real threat. Conveniently, real attacks exist in huge repositories at “computer security” sites such as *packetstorm* (www.packetstorm.org), catalogued by operating system and application. It is possible to monitor the effects of real attacks in a variety of ways and be confident that the effects are real and not an artifact of abstraction. The limitation is that they may not be representative or general enough to appropriately test a system’s classification mechanism. There are simply not enough non-DoS attacks available for any single application to create a statistically relevant amount of test data.

Artificial Anomalies

Fan et al describe the goal of generating artificial anomalies as “to coerce an arbitrary machine learning algorithm to learn hypotheses that separate all known classes from unknown classes.” In contrast to real attacks, artificial anomalies must be generated. The benefit of such a method is that a larger, more diverse set of data can be created, leading to more general models of self and non-self. Fan et al use their algorithm to design “’near misses,’ instances that are close to the known data, but are not in the training data.” [Fan01]

Research Impact

It seems that such a technique could be used at a layer of abstraction – if a plausible technique for characterizing “self” could be determined *a priori*, a wide range of “near misses” could be designed based on that characterization. These anomalies could help test the limitations of an IDS.

2.8 Summary

This chapter reviewed several background topics considered necessary for the foundation of this research. The topics discussed included intrusion detection systems, evolutionary algorithms, system and application vulnerabilities, as well as various types of attacks that are enabled by these vulnerabilities. Several analytical methods used to monitor the effects of attacks and a method for generating anomalies were also discussed.

3. High-Level Design

3.1 Introduction

This chapter outlines the methodology for the production of AIS detectors that can be used within an application-based, anomaly-detecting IDS. The goal of this system is to use a number of event sequence logs based on execution path markers and analyze their order to find patterns that describe non-self sequences. These patterns are incorporated into antibodies used by Genetic Programming agents in an attempt to do sequential event analysis of potential intrusions. This chapter discusses specific research objectives and examines solution methodologies. In addition, technical issues that surfaced along the way are presented. Solution designs for each research objective and challenge are explained in more detail in Chapter 4.

3.2 Overview of Objectives

1. It is necessary to define a set of operators that is sufficient to characterize the relationships among a set of events that occur during process execution.
2. These relationships may be of varying complexity. Therefore, once a set of operators is defined, a plausible technique for combining them so as to be able to characterize more complex effects must be determined.
3. Once the descriptors have been produced, a process is needed to categorize the descriptor as describing a normal sequence or an anomalous one.

4. A problem inherent to anomaly-based ID is that it is difficult or impossible to differentiate between benign and malignant anomalies. This analytical step generally must be performed by human analysts post-mortem, and is both time-consuming and tedious. The fourth objective concerns an attempt to extract and compile data to assist the analyst in making this determination.

The methodology is designed to address all these objectives.

3.2.1 Desired Characteristics of System

The system is application-based; it is not designed to detect scans, DoSs that overload resources or tie up bandwidth, or any other attacks in the realm of network-based intrusion detection.

The system is designed to use event logs. This level of abstraction provides several benefits. It allows testing of the system against a variety of simulated attack types. Logs can be transferred from any system to a single Analysis System, so "portability" is not an issue. This log transfer is also good for security: attackers tend to cover tracks on victim systems, so it is good security practice to keep records elsewhere. Lastly, by using logs, the potential of a Denial of Service being performed against the IDS is eliminated. Attackers often attempt to "flood" a system with data if they are aware (or suspect) an IDS is on it. This overloading can cripple or crash an IDS that is attempting to perform detection in real-time, in which case any attacks performed after the DoS are missed.

It is assumed the test application is instrumented with sensors using path profiling techniques, or by random dynamic instrumentation (see Chapter 6) if only the executable is available. These “canary” sensors are individually and uniquely identified; when their code is executed, they “trigger” and write their identifier to a log. A sequential log represents the order of triggered sensors as a one-dimensional array of size *m* where log entry[m] represents the mth sensor to trigger. The system is given a training data set of “self,” non-attack logs that is uses to develop a set of antibodies. These antibodies represent event sequence relationships unbounded by event size windows. Genetic programming techniques are used to develop these antibodies, as GP was determined to be an appropriate way to develop a range of differently-sized antibodies appropriate for the task. These antibodies are then “deployed” against a test set consisting of both self and non-self logs to determine the efficacy of these techniques. Analyst evaluation of the results creates a feedback loop to update the antibodies.

3.3 High-Level Design

The design incorporates certain components of the general AIS model.

3.3.1 Pseudocode for Overall Structure

```
create first generation
do
{    //TRAIN
     evaluate generation against training set
     negative selection
          select all sirens that do not match self
          perform affinity maturation for selected sirens //OPTIONAL
          place matured sirens in Antibody DB
          replace with newly-created ones
     create next generation
          use EA operations
}while not enough antibodies
```

```
//DEPLOY
for all test cases
{
    run antibodies against test case
    provide feedback
}
```

3.3.2 Design of Program

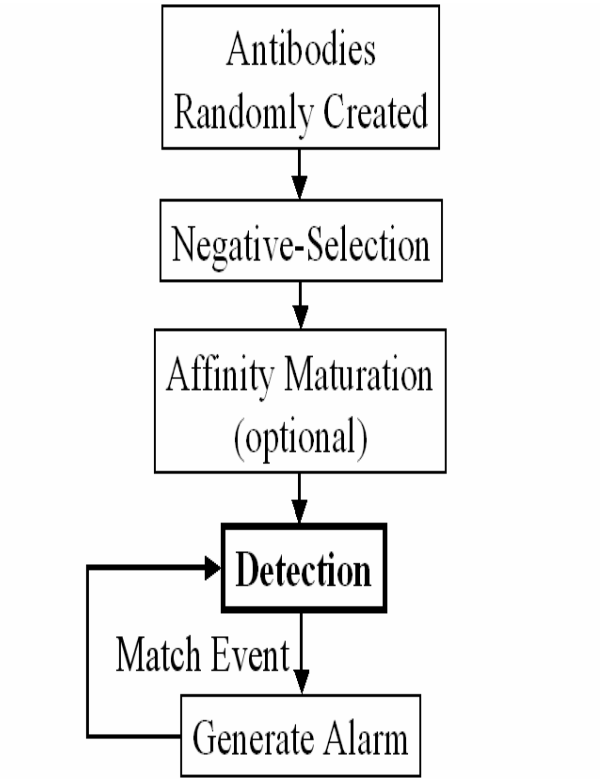


Figure 1. A Modified Subset of the AIS Cycle

The design incorporates the components of the cycle illustrated in Figure 3.1. This cycle is based on that described by Williams [Will01].

3.3.2.1 Chromosome Generation

A design is presented for the GP "sensor analyzers" that are used in the system. Because these monitor programs are supposed to sound an alarm in the event of detection of a potential intrusion, they are referred to as Sensor-based Intrusion Response Event Notifiers, or *sirens*. If a siren survives the Negative Section process, it becomes an antibody.

Producing the sirens is a matter of finding an appropriate grammar. A context-free grammar (CFG) is a language generator; in this case each chromosome generated is a string in the language. A CFG is used for several reasons. A context-free grammar is used to produce a Type 2 language, and is the minimum level grammar required to ensure that all words produced are of the correct structure to enable recombination. Specifically, recombination must always result in well-formed words. This goal cannot be achieved with a Type 1 or Type 0 language. Furthermore, a context-free language is a superset of a regular language; therefore if a regular language is sufficient to characterize either self or non-self, the regular language can be described as a subset of the context-free language.

More formally, a context-free grammar G is a quadruple (V, Σ, R, S), where [Davi94]:

V is an alphabet,

Σ (the set of terminals) is a subset of V,

R (the set of rules) is a finite subset of (V-Σ) x V*, and

S (the start symbol) is an element of V-Σ

When using a CFG, one starts with the symbol S and applies the rules defined in R to create a word in the language. In this case, a word is a siren, and the language describes potential system events. Once we know which terminals and functions we need to characterize in the system, developing a CFG to automatically generate initial generations is relatively simple.

3.3.2.2 Operators and Operands

This section discusses the set of terminals (operators and operands) necessary for this undertaking in terms of functionality and purpose. Implementation is covered in Chapter 4.

One of the most important things to consider in GP is what terminals need to be used. If these terminals are not designed properly, it may be impossible or at least improbable that the system can evolve an acceptable solution. For example, if a system were designed to perform symbolic regressions for cubic equations (e.g. $f(x)=x^3+2$) and the only terminals defined were “1” and “add,” the system would never be able to find the desired solution. One would in this case desire additional terminals representing “x” and “multiply.”

Some terminals must be able to represent a set of temporal relations among events, as represented by the log files. Terminals must be able to access and manipulate the log data to determine if the relationship represented exists in the log. For example, it might be desirable to describe the following situation:

> Within a certain number *n* of events beginning at a particular event, Sensor X is triggered, then Sensor Y is triggered twice. Within this same event range Sensor Z is triggered; it is not relevant when this occurs with respect to Sensors X and Y.

For this simple example, operators would be needed to describe the concepts of “[event] happens before [event],” “[event] happens within [number of events],” and “[situation a] AND [situation b] both occur.”

How can an appropriate set of operators be determined? Applying what is known about the intrusions against which the system is tested can help solve this problem. For example, buffer overflows have certain basic effects on the application against which they are run: they can cause the program to crash, cause arbitrary code to be executed, and/or cause other values to be overwritten. Tied to these effects may be significant additional time delay between execution of commands. Depending on which “events” are monitored, there are a variety of ways the effects can be detected.

A crash could occur after an event e_1 that is normally followed by event e_2. An operator that represented the *happens-before* relation is a likely candidate for differentiating between these two; "e_1 *happens-before* e_2" would evaluate to TRUE for the normal case, but would evaluate to FALSE for the attack.

Another possibility involves an attack that does not crash the application, but merely executes additional commands prior to returning to normal execution. If system calls are being monitored, the normal case might be represented as {... e_1, e_2, ...} in an event log, while the attack case would be represented by { ...e_1, <additional system calls>, e_2,..." Here the *happens-before* operator would not be able to distinguish between the two logs. A different operator is required to account for *proximity* of events; a sufficient choice is a *happens-within* operator. Assume the existence of a log index pointer that is set to point to the first event in a log: e_1 in this example. The function "e_2 *happens-within 1*" represents the question, "Does event e_2 happen within one event of the pointer?" This function is perfectly capable of classifying the two cases, and if the number of events is set to the size of the log, this function is equivalent to the *happens-before* version. To move down the log (so that it is possible to characterize relationships between any pair of events) requires a function that can reset the index pointer. This requirement is accomplished by adding a similar operator, *happens-within-reset*, which resets the index pointer to point to the event if it is found within the required number of events. Both of these operators evaluate to TRUE if the conditions are met and false otherwise. The *happens-within* operator is still needed to characterize cases where a set of events are present, but in no particular order (e.g. if both sequences {e_1, e_3, e_2, e_4} and {e_1, e_2, e_3, e_4} were considered "self"). This example also illustrates the need for a logical *AND* operator to link these relationships together.

Now consider the simplest form of a case involving a break in execution that occurs within a loop:

Self = {e_1, e_1}

Attack = {e_1}

None of the above operators can assist in differentiating between these two logs. We need to add an operator that can move through a sequence even if the events are duplicates of the events pointed to by the index pointer. A *next* operator that takes event *e* as a parameter will satisfy; it looks for the next occurrence of *e* and sets the index pointer to its position if found. As above, if it is successful it evaluates to TRUE, otherwise to FALSE.

Are these operators sufficient to characterize sequences? A more complicated example illustrates that they are not. Consider the following highly abstracted pseudocode:

```
$e1$
if ($variable$)
    then $e_2$
else
    $e_3$
$e_4$
```

If *variable* = 1, the event log reads: e_1 e_2 e_4; if variable = 0, the event log reads: e_1 e_3 e_4 This example demonstrates the need for a logical *OR.*

next e_1 *AND* e_2 *happens-within 1 OR* e_3 *happens-within 1 AND next* e_4

This siren describes both these cases.

A logical *NOT* is also useful; if a chromosome describes “self” cases, simply negating it may be an efficient way to describe many non-self cases.

It may also be required to identify patterns consisting of multiple distinct sequences that overlap to some extent; the addition of a *previous* operator enables the discovery of such cases. It functions similarly to *next*, but searches backward in the log.

It can be seen that these terminals can characterize truncation, succession, multiple orderings, removal, replacement, repetition, and interlacing of sequences, which appears to cover the set of possible attack effects on sequences. Thus, the required set of terminals consist of the following, where

E is the set of events,

I is the set of distances between events,

A represents the evaluation of a function or a logical combination of functions,

Event $e \in$ E, and

Integer $i \in$ I

Event Relationship Operators

(Note: Functions only have side effects if so stated)

e happens-within i:

Evaluates to: TRUE if event *e* occurs within i events of Log Pointer
FALSE otherwise

e happens-within-reset i:

Evaluates to: TRUE if event *e* occurs *within* i events of Log Pointer
FALSE otherwise

Side Effect: If TRUE, set pointer to the first occurrence of event *e* found *at* or *after* Log Pointer.

e next

Evaluates to: TRUE if event *e* occurs in the remainder of the log (*after* Log Pointer)
FALSE otherwise

Side Effect: If TRUE, set pointer to the first occurrence of event *e* found *after* Log Pointer.

e previous

Evaluates to: TRUE if event *e* occurs in the section of the log *prior to* Log Pointer
FALSE otherwise

Side Effect: If TRUE, set pointer to the closest location of event *e* found before Log Pointer

Logical Operators

A_1 AND A_2

Evaluates to: TRUE if both A_1 and A_2 evaluate to TRUE
FALSE otherwise

A_1 OR A_2

Evaluates to: TRUE if either A_1 or A_2 evaluate to TRUE
FALSE if

NOT A

Evaluates to: TRUE if A evaluates to FALSE
FALSE otherwise

Evaluation Examples

Given a producible siren

5 5 happens-within-reset 7 3 happens-within AND NOT

its value when interpreted on two different logs can be determined.

1. Log = {1 2 3 4 5 6 7 8}
    The pointer is set to 0, so that it points at event 1.
    Evaluate first relationship: *5 5 happens-within-reset*
        This relationship is TRUE; the pointer is reset to 4, pointing to event 5.
        The TRUE value (1) is pushed onto a stack
    Evaluate second relationship: *7 3 happens-within*
        This relationship is TRUE.
        The TRUE value (1) is pushed onto the stack
    Evaluate *AND*:
        Two values (1 and 1) are popped off the stack.
        1 AND 1 is 1; the one is pushed onto the stack.
    Evaluate *NOT*:
        One value (1) is popped off the stack.
        NOT 1 is 0; the zero is pushed onto the stack.
    The end of the siren has been reached; its evaluation is the value on the stack, zero. Therefore, this siren does not match the log.

2. Log = {8 7 6 5 4 3 2 1}
    The pointer is set to 0, so that it points at event 1.
    Evaluate first relationship: *5 5 happens-within-reset*
        This relationship is TRUE; the pointer is reset to 4, pointing to event 5.
        The TRUE value (1) is pushed onto a stack
    Evaluate second relationship: *7 3 happens-within*
        This relationship is FALSE.
        The FALSE value (0) is pushed onto the stack

Evaluate *AND*:

Two values (0 and 1) are popped off the stack.
0 AND 1 is 0; the zero is pushed onto the stack.

Evaluate *NOT*:

One value (0) is popped off the stack.
NOT 0 is 1; the one is pushed onto the stack.

The end of the siren has been reached; its evaluation is the value on the stack, one. Therefore, this siren matches the log.

3.3.2.3 Training

The goal of the training stage is to develop a set of sirens that represent non-self sequences. The assumption is that an intrusion results in different event patterns at the sensor level that are "matched" by one of the non-self sirens.

3.3.2.3.1 Negative Selection

Negative Selection in an AIS is intended to mimic the biological process by which antibodies produced by an organism are screened. Essentially, an antibody is exposed to a selection of normal components of the organism. If an antibody binds to one of these components, it is discarded. Only the antibodies that do not "attack self" are allowed to be deployed as part of the immune system. Similar methods are used in AISs, the goal being to create a set composed of individuals that will respond to certain anomalies, while yielding a low false positive rate. For this system, some method must be devised to ensure no self-matching antibodies are found in the final set.

3.3.2.3.2 Affinity Maturation

Affinity maturation is an optional stage in this system because there are no demonstrably appropriate ways of maximizing the "usefulness" of this type of antibody. The use of "wildcards" in the chromosome, a common technique, would be useless with regard to most of the terminals. For example, an antibody representing "Some unspecified event occurs within 1 event of the pointer" would always alert, no matter what logs were used. The operations likewise cannot be generalized; what meaning has the antibody "A some-logical-operation A?" It cannot be evaluated. The only obvious way to generalize these antibodies is by varying the "within *i*" integer values. Even this method is not as simple as it may seem. An antibody with a low *i* value may be very specific ("e happens within 1 event of pointer") or very general ("e *does not* happen within 1 event of pointer"). Multiple logic operators may change or eliminate the effect of a single relationship operator; there is no consistency. Since there is no way to measure the volume of the antibodies, the only logical compromise is to create two copies and increment one and decrement the other until they impinge on self or exceed the range of the set I. The drawbacks of generalization may actually outweigh the benefits. One of the objectives of this system is to attempt to locate the point in the log at which intrusion occurs; an antibody that states, "An anomaly occurs somewhere in the last half of the log" is less useful in this regard than one that states, "An anomaly occurs *here*." It may be more beneficial to avoid generality and aim for a large quantity of antibodies.

3.3.2.4 Quasi-GP Engine

This section briefly discusses the application of GP techniques in this research, and how they differ from traditional GP.

Traditional generational GP uses recombination and point mutation operators to create generations of program chromosomes. The individuals in each generation are evaluated using a fitness function, and a selection process is applied to determine which will reproduce. If the problem is solvable, the fitness function and selection mechanism are designed properly, the system is run for sufficient time, it is probable that a satisficing solution can be found. Generally a single solution is sufficient, and *convergence* plays a large role in the speed with which a solution is found.

In this research, by contrast, the mutation operators, fitness function, and selection mechanism all exist, but they are applied on consecutive generations not to evolve a single satisficing solution, but to evolve many solutions. Once a “solution” is found, it is added to the set of antibodies and replaced with a new chromosome. As a result, the engine faces a somewhat Sisyphean task; just as solutions are achieved, they are removed from the genetic pool. It is important to note the constant influx of new genetic material that frustrates the process of convergence. This is beneficial for our purposes as excessive convergence is not desirable.

Recombination/Crossover

To ensure that there are no type issues – for example, feeding a function a logical evaluation when it is expecting an integer – only branches where the first nodes are of equivalent types will be swappable.

Point Mutations

Point mutations include replacing an event with another from set E, replacing an integer with another from set I, and replacing operators with similar types (i.e. AND ⇔ OR).

Fitness Function

A fitness functions is required for the training phase. Specific low level design and implementatio n is discussed in Chapter 4.

Deployment

Deployment involves running the sirens on a set of logs representing a combination of known self and known attack logs and noting the results for human analyst response.

Feedback loop

The feedback subsystem should work by adding or removing antibodies as indicated by analyst response. The possible cases are as follows:

? **Both self and attack cases, log is properly classified**: No changes are necessary.

? **Self, classified as attack** (False Positive): The antibody is removed from the database and the log is added to the set of training cases.

? **Attack, classified as self** (False Negative): An antibody is generated that can differentiate between the training set and the test case. It is added to the database.

3.3.3 Design of Training and Test Sets

The antibodies need to be tested using a variety of attacks that represent common exploits. These attacks can be designed or simulated, as long as they represent a variety of sizes and complexity over a range of relevant attacks (e.g. buffer overflows, race conditions). To generate the training sets, the events of an application are represented as a regular expression. A set of words produced from this expression becomes the training set.

It is important to note that this technique is used only for research purposes to demonstrate some of the capabilities and limitations of the system. If it was known *a priori* that an application could be represented by a particular regular expression, *and* that any word not produced by the expression is an intrusion/anomaly, we could save ourselves a lot of effort by just looking at a test log and testing the "word." There are three reasons these assumptions cannot be made in the real world:

1. It cannot generally be determined (without extensive human analysis) what expression would actually represent the set of desirable paths; extensive human analysis is to be avoided.
2. Alternatively, a full representation of the source code might actually include "anomalies;" for example, a branch that is only used in case of massive error would not normally be executed, but the analyst might want to be informed in the case that event occurred.
3. Even if an appropriate expression was able to be determined, it is not necessarily exclusive of anomalies. An anomaly might not be detectable; i.e., a modification to a word might yield another word produced by the same expression. Such cases are possible sources of false negatives using this method.

For these tests, the execution paths are represented as follows:

- Each command is a unique event

- Conditionals:

The situation

```
e1;
if(condition)
        e2;
e3; //etc
```

is represented as: $(e_1 \, (\Lambda + e_2) \, e_3)$

Whereas

```
e1;
if(condition)
    e2;
else
    e3;
e4; //etc
```

is represented as: $(e_1 \, (e_2 + e_3) \, e_4)$

- Loops: *for* and *while*

A *for* loop

```
e1;
for (i = 0; i<n; i++)
    e2;
e3; //etc
```

becomes $(e_1 \, (e_2)^n \, e_3)$

while loops are represented similarly:

```
e1;
while(condition)
    e2;
e3; //etc
```

becomes $(e_1 \, (e_2)^* \, e_3)$;

however, for the purposes of actual data set generation, the Kleene star is replaced with a more practical upper limit, such as "4."

Test sets

The test cases of "unknown" logs require a set of logs that simulate self, and a set that that represents anomalous intrusions. To generate these logs, words are produced from the regular expression used to generate the test set. Some of these words are kept unmodified; they become the "self" test set. Note that the words comprising this set must be tested against the training set to determine that there is no overlap between the sets; if there is, the offending test word must be replaced or eliminated.

The "attack" subset of the test cases is created by modifying words (as described in Section 4.2.8) to simulate the effect of an intrusion.

3.4 Summary

This section outlined the methodology for the high-level design of a hybrid form of AIS that acts as an application-based, anomaly-detecting IDS. The goal of this system is to use a number of event sequence logs based on execution path markers and analyze their order to find patterns that describe non-self states. These patterns are incorporated into antibodies used by an AIS in an attempt to do sequential event analysis of potential intrusions.

4. Low-Level Design and Implementation

4.1 Introduction

This chapter covers design issues and implementation details to accomplish the research objectives covered in Chapter 3.

4.2 Implementation/Problem Solving

This section discusses specific implementation details and problem solutions. Choice of programming language, GP terminals used, chromosome generation, and different phases of the AIS and GP subsystems are discussed. Design of synthetic data sets and a data extraction method are also discussed.

4.2.1 Implementation Details

The system was implemented in C++, a commonly used object-oriented language. C++ was chosen over Java due to speed issues with the Java Virtual Machine. While Java is often preferred due to its portability, this system uses logs, rendering the problem moot.

4.2.2 GP Terminals

This section discusses the implementation of terminals (operators and operands) deemed necessary for this research.

It was established in Chapter 3 that the sequencing operators *happens-within*, *happens-within-reset*, *next*, and *previous*, combined with the logical operators *AND*, *OR*, and *NOT* appear to be sufficient to characterize truncation, succession, multiple orderings, removal, replacement, repetition, and interlacing of sequences. All operators were implemented with the exception of *previous*, which was only required to enable the characterization of interlaced sequences. A review of attacks indicated that an operator that could perform identification of interlaced sequences would be superfluous; no attacks within the scope of the research would require it to identify them.

The terminals were implemented as follows:

Operands

E (set of events): integers in the range [1,n], where n is the number of "sensors"

I (distance between events): integers in the range [0, (size of largest log in training set)-1]

Functions

All functions were implemented using post-fix notation for to facilitate stack-based evaluation.

Event Relationship Operators
e happens-within i:

Arity:	binary
Represented as:	e i happens-within
Returns:	integer: 1 if TRUE, 0 if FALSE

e happens-within-reset i:

Arity:	binary
Represented as:	e i happens-within-reset
Returns:	integer: 1 if TRUE, 0 if FALSE

e next

Arity:	unary
Represented as:	e next
Returns:	integer: 1 if TRUE, 0 if FALSE

Logical Operators

A_1 AND A_2

Arity:	binary
Represented as:	A_1 A_2 AND
Returns:	int: 1 if $A_1 = 1$ and $A_2 = 1$, 0 otherwise

A_1 OR A_2

Arity:	binary
Represented as:	A_1 A_2 OR
Returns:	int: 1 if $A_1 = 1$ or $A_2 = 1$, 0 otherwise

NOT A

Arity:	unary
Represented as:	A NOT
Returns:	int: 1 if A = 0, 0 if A = 1

4.2.3 Chromosome Generation

As noted in Chapter 3, producing the sirens is a matter of finding an appropriate CFG. Based on the discussion of necessary operators, the generation rules can be defined as follows:

Intermediate Symbols:

L: placeholder for a logical evaluation (AND, OR, NOT)

B: placeholder for a relationship evaluation

A: placeholder for a logical or relationship evaluation

Production Rules:

S ? L
S ? B
L ? A A and
L ? A A or
L ? A not
A ? L
A ? B
B ? E I occurs-within
B ? E I occurs-within-reset

A symbol has equal probability of having any of the relevant rules applied; for example, 'L' has a 33.3% probability of becoming 'A A and,' 'A A or,' or 'A not.'

All chromosomes were implemented as linear doubly-linked lists of nodes (represented as structs) to allow for easier expansion of the terminal set, should functions of different arity ever need to be added to the system.

4.2.4 Training

The goal of the training stage is to develop a set of sirens that represent non-self.

Negative Selection

Negative Selection is accomplished by only adding to the Antibody Database sirens which did not match self data and which were not already present in the Database. Matches are determined by evaluating the sirens against each element of the test data set and OR-ing the results, so that if any one of the test sets matches it, it does not become an antibody. If the siren is a non-match for all training cases, the Database is then searched for a matching chromosome; if none is found, the siren becomes an antibody. This method ensures the set is composed of unique individuals.

Affinity Maturation

Affinity maturation using the methods described in Chapter 3 was originally implemented, but pilot tests revealed an unacceptable degree of nicheing occurring as a result. Essentially, the individuals from the first several generations disproportionably contributed to the Antibody Database by flooding it with multiple slightly altered copies of themselves, lowering the diversity of the Antibody population and lowering the rate of anomaly detection. This nicheing also served to make analysis more difficult in the cases when anomalies were detected; many antibodies were subsumed by others almost identical to themselves, resulting in the extraction of more redundant information.

4.2.5 Quasi-GP Engine

A *generational* design was used. This means that during each iteration step (generation), all the individuals in the current population are evaluated and given a fitness value.

4.2.5.1 Selection

A standard *binary tournament selection* with replacement was used [Banz98]. Two individuals are selected from the population and the individual with the highest fitness is allowed to survive to the next generation. Neither is removed from the population pool. This process is repeated until the required number have been selected.

4.2.5.2 Crossover

Crossover was done by selecting a point from 0 to “size of chromosome” and swapping the branch beginning at that point with a similarly-chosen branch from another siren. To maintain operand type consistency, a *context preserving* method was used that insured crossover only occurred if the selected nodes were of exchangeable type [Banz98]. For example, a Boolean expression could not be replaced with an Integer type.

4.2.5.3 Point Mutations

Point mutation was accomplished using the following mutators, as appropriate, with probability p_m:

If chosen node is of type

E: replace with randomly chosen sensor

I: replace with randomly chosen integer

Logical, binary (AND/OR):

Replace with another, or negate by inserting NOT after it

Logical, unary (NOT):

Remove the node; equivalent to negation

Relationship, binary (happens-within(-reset)):

Replace with another, or negate

Relationship, unary (next):

Negate

4.2.5.4 Fitness Function

Training phase: A simple yet serviceable fitness function was used during the training process. The initial "score" of each siren was set to 0. The chromosome of a siren was evaluated on all n elements of the training set. For each training log, if the evaluation was equivalent to 1, the siren was considered to have "matched" the log, and the value 1 was added to the score. After the siren had been evaluated for all members of the training set, the score was interpreted as an inverse measure of fitness; i.e., the sirens with the lowest scores matched the least number of training sets and were therefore of the highest fitness. Attempts were made to further differentiate among sirens' fitness based on the size of their chromosomes; however, pilot tests indicated this idea was not appropriate, as it drove the sizes down so that there was not sufficient variation to fill the Antibody Database in a reasonable time.

4.2.6 Deployment

Deployment was simulated by running the siren antibodies on a set of logs representing a combination of self and attack logs. The accurate classifications were known to the tester a priori. Execution time and classification results resulting from deployment are noted in Chapter 5.

4.2.7 Feedback Loop

Although the feedback subsystem would be a necessity if this system were to be deployed in the real world, the full feedback system was not implemented. The reason for this decision is that for research purposes, it is desirable that all elements of the data set experience the same environment; i.e., the same set of antibodies. If the full feedback loop is in place, this cannot be accomplished, as feedback from a false classification may affect the classification of subsequent logs. However, during testing, the feedback loop was *partially* implemented so as to be able to test it in a controlled manner. (Chapter 5 presents the details.)

False Negatives: In this case, *no* antibody alerts on a known anomaly. A number (g) of further generations of antibodies are evolved. If within g generations, an antibody is generated that can differentiate between the training set and the test case, then it is added to the Antibody Database. The number of allowable generations is limited because pilot tests indicated it could take an unreasonable amount of time to evolve an antibody that satisfies these conditions. Moreover, it may be impossible to differentiate between an "anomaly" log and a "normal" log in real life – for example, if the anomaly did not create a measured change in the execution path. With this system, training and test cases are designed so that this is not the case.

False Positives: In this case, one or more antibodies alerts on a known normal log. The log is added to the training set, and the alerting antibodies are removed from the Database.

4.2.8 Design of Training and Test Sets

All logs were represented as text files containing an un-indexed, one dimensional array of event data. Synthetic training and test sets were built in the following manner. Two programs are written to assist in the development of data sets. The first takes a regular expression (in an unconventional form) and converts it to a functional representation. The standard Kleene star operation is defined as creating the largest set that can be made by concatenating zero or more strings from a set of strings. For the purposes of this research, a “limited Kleene star” operation is defined as creating the set of strings that can be made by concatenating one to n. This operation is represented by “kstar(),” disjunction is represented by or(), and conjunction is represented by “cat().” For example, the regular expression

((a1+a2+a3+a4) (a9 (a10+a11))*) is transformed to

cat(or(a1,or(a2, or(a3,a4))),kstar(cat(a9,or(a10, a11)))))

The second program fully enumerates all words producible by this expression. Each word represents a normal log created by traversing a particular execution path. For this example, with n=2, the words in the language are:

```
a1 a9 a10
a1 a9 a11
a1 a9 a10 a9 a10
a1 a9 a10 a9 a11
a1 a9 a11 a9 a10
a1 a9 a11 a9 a11
a2 a9 a10
a2 a9 a11
a2 a9 a10 a9 a10
a2 a9 a10 a9 a11
a2 a9 a11 a9 a10
a2 a9 a11 a9 a11
…etc
```

From this set, training and test sets are produced. The second largest power of two *less* than the number of words produced becomes the test set size (eight, in this case). The test set is built by selecting random strings such that there are no duplicates in the test set. Training sets of cardinality two through "test size," in multiples of two, are produced by selecting random strings such that there are no duplicates in the training set and the training and test sets do not intersect. This process would produce three training sets with cardinality of two, four, and eight. The test set is then modified so that half the strings represent anomalies, to simulate the effect of an intrusion. Modification is done by applying truncation, removal, insertion, and replacement operators, each to one quarter of the anomalous test set.

Truncation: A random number of events (between one half and one quarter of the log size) are removed from the end of the log. This modification represents attacks that cause a break in execution, such as many stack-based buffer overflows.

Removal: A random number of events (between one half and one quarter of the log size) are removed from some portion of the log. This modification represents an attack that causes the application to take an unusual path that skips normally-seen events.

Insertion: A random number of placeholder events that will be unrecognizable to the evaluation portion of the system (between one half and one quarter of the log size) are added to the log. This modification represents an attack that causes the application to take an unusual path that encounters normally-unseen events.

Replacement: A random number of events (between one half and one quarter of the log size) are replaced with a number of placeholders (not necessarily of equal cardinality to the set of events replaced) that will be unrecognizable to the evaluation portion of the system. This modification can represent an attack that causes the application to take an unusual path; in an abstract sense it can also represent a quantitative change in some other monitored event type, such as time delay between commands.

Specific training and test sets are described in Chapter 5.

4.3 Difference Analysis

This stage was added to facilitate human analysis of anomalies by creating what is referred to as "Difference Essences," or DEs. DEs represent only the parts of an antibody chromosome that specifically match a nonself log and do not match any of the self logs. DE sets are extracted from altering antibodies for each test log. Basic DEs were isolated by using DeMorgan's Laws and splitting chromosomes using the following algorithm:

Use DeMorgan's Laws to simplify chromosomes by moving *not*s as far down the tree as possible:

((A B or) not) ? ((A not) (B not) and)

((A B and) not) ? ((A not) (B not) or)

Remove all negated *not*s:

((A not) not) ? A

Where the terminal function is an *or* or *and*, both branches are individually tested and added to the DE set if they are not self-matches.

Analyze Difference pseudocode:

```
start processing from the end of the chromosome (the rightmost operator)

if operator = = (not || happens-within || happens-within-reset)
    add branch to Difference Essence set
else if operator = = and
{
    if(!rightbranch matches self)
    {
            Analyze Difference(rightbranch);
    }

    if(!leftbranch matches self)
    {
            Analyze Difference(leftbranch);
    }
}
else if operator = = OR
{
    if((!rightbranch matches self)&&(rightbranch matches nonself))
            Analyze Difference (rightbranch);

    if((!leftbranch matches self())&&(leftbranch matches nonself))
            Analyze Difference(leftbranch);
}
```

This method breaks out all DEs, which can be combined using conjunction to produce a description of the difference between the anomaly and "self." It is important to note that the DEs themselves can have their pointers set to various starting points within the log – in other words, *3 happens-within 2 (of index position 0)* is different than *3 happens-within 2 (of index position 2).*

This process was successfully implemented. Further refinements are necessary; it is possible to have several unique DEs that describe overlapping conditions:

3 next (after pointer [0]) includes *3 next (after pointer [2])*

If the first is true, the second must also be true and vice versa. The superfluous statements need to be removed.

4.4 Summary

This chapter discussed in detail the low-level design and implementation of the system. In addition, issues that surfaced throughout the development process were explained, and appropriate solutions were presented.

5. Test Cases and Results

5.1 Introduction

This chapter reviews relevant research objectives, experiments, test cases, and evaluation results. Analysis of test results indicate that the system is successful in reaching its objectives. Further research and exploration is necessary to more thoroughly verify concepts brought about by this research. A set of future work recommendations is outlined in Chapter 6.

5.2 Design of Experiments

These experiments are needed to determine whether the implemented system can successfully classify event logs as “normal” or “anomalous.” They assist in determining to what extent the research is successful in achieving Objective III: Determine an appropriate method of classifying these relationships as “normal” or “anomalous.”

Testing occurred in several phases. Initial pilot tests were made to determine to what levels certain factors should be set. The results of these tests are discussed qualitatively in the Factors section. These levels were maintained throughout testing to provide continuity for all tests and to limit the number of experiments.

After these levels were set, experiments were run to determine the effectiveness of the system as determined by levels of false positive and false negatives. These tests were run with and without feedback to determine if feedback provided a benefit that outweighed its drawbacks. These experiments were run on a small set of different “simulated applications” represented by regular expressions.

5.2.1 System Boundaries

The System Under Test (SUT) is the IDS system. The Component Under Test (CUT) is the amalgam composed of the antibody generator and the detection module within the IDS responsible for identifying anomalies based on the logs. This CUT is being evaluated for effectiveness of detection. The system accepts an application log as input and outputs a classification – potential attack or normal behavior – based on the alerting of antibodies.

5.2.2 System Services

The service the system provides is the classification of a set of "logs" by the IDS. The possible outcomes are values representing the number of unique sirens that alert on a particular test case. These values can be *interpreted* as "Self" and "Non-self" classifications associated with a degree of sample frequency. These outcomes comprise all possibilities.

5.2.3 Performance Metrics

For the IDS, effectiveness describes the ability of the system to detect anomalous occurrences. For example, if a test set of 100 logs contained 20 attacks, and the system successfully identified 10 attacks while misidentifying 5 of the "self" test cases, the following statements could be made:

- 10/20 attacks were detected; the false negative rate is 50% for this test set
- 5/80 false positives occurred; the false positive rate is 6.25% for this test set

Efficiency is the performance, quantitative and qualitative, of the analysis program. It can be measured as raw time (seconds to completion), or calculating ratios of time as factors are manipulated. Efficiency is measured for this system to assess the effect of increased number of training cases in a training set on the time required to produce a set number of antibodies.

5.2.4 System Parameters

System parameters include the Central Processing Unit (CPU) model and speed, amount of memory, operating system. The following hardware/software specifications are used to evaluate the system: Pentium III, 1.7 GHz with 512 MB RAM running Microsoft Windows Me. All code is written predominantly in C++.

5.2.5 Workload

The workload submitted to the system consists of two parts: a training set and a test set. The training set represents the results of sequences of “common” appropriate logs produced by the application. Once the system is exposed to a set of “self” training data, the test set is introduced. The test set is similar to the training set, but is modified to represent requests that exploit vulnerabilities in the application as described in Chapter 4. The IDS processes the sets and reports the number of antibodies that alert.

5.2.6 Factors

The factors to consider for designing these experiments can be decomposed into two categories: those that characterize the data, and those that affect the operation of the IDS itself.

The number of sensors used to create the data sets has a major impact on the performance of the IDS, as each additional sensor increases the search space. Number of sensors is varied: low, medium, and high. A low number represents an application with only a few sensors placed at strategic points, the locations of which are determined via path profiling or another instrumentation technique. A high number, by contrast, represents a system with sensors placed after every line of code. For the purposes of scoping the assumption is made that the applications represented in these experiments are of relatively small size and can be adequately instrumented using 16 sensors. This assumption is reasonable; many commonly exploited applications are quite small (10-30 commands) and have relatively few possible execution paths. Because it is important to test the ability of the IDS to characterize complex programs, all regular expressions used to generate training and test cases are designed to produce 2^8-2^9 execution paths.

The IDS factors include population size during antibody production, termination condition for the antibody production stage, and the frequency of recombination and point mutation. Pilot tests were completed to determine settings for these factors based on results when run on a representative selection of training sets using 16 sensors. Levels were tested at factors of two.

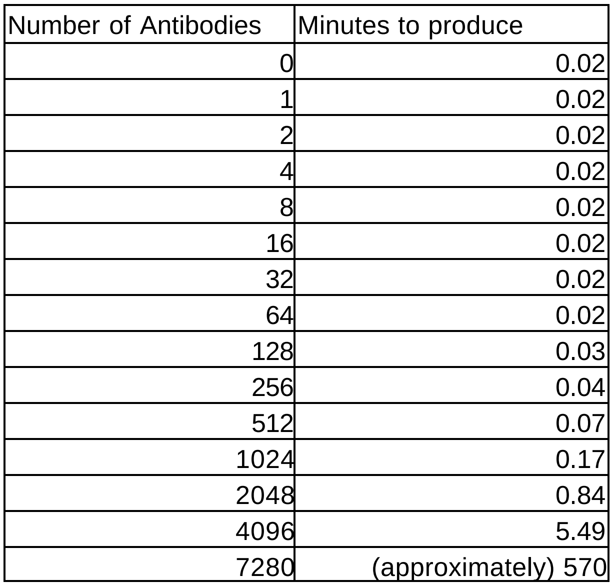
Population size was set to 512 sirens per generation. This size was found to be both consistent and stable in terms of the number of antibodies produced from each generation. With a significantly smaller population size, the number of antibodies produced tended to dwindle rather quickly, resulting in a rate of production that was lower and of no apparent better quality.

The maximum size for a chromosome was limited to 64 nodes. With significantly fewer nodes, the system again produced antibodies at a very slow rate, since there were many fewer possible sirens in the search space. With significantly more nodes, evaluation of the individuals in each generation became unmanageable after several generations, and the system often crashed from memory exhaustion (heap overflows).

A higher percentage of false negatives was correlated with a smaller Antibody Database; however, if the system was set to produce significantly more than 8192, production took an unacceptably long time, in some cases running days without terminating. Data from a simple example using a training size of 16 can be seen in Table 1.

A goal of 4096 antibodies was determined to be an acceptable termination criterion. With higher cardinality training sets, the termination criterion took disproportionally longer to reach; thus, another termination criterion was added to limit the number of generations to 1024. With mutation in standard GP, where convergence is important, the emphasis is usually on recombination; point mutations are usually kept relatively low, limiting the amount of new genetic material in the system. In this system, the ideal was to have just enough convergence to allow solutions (antibodies) to be reliably produced, but to represent as diverse a population as possible within the solution criteria. For this reason, both recombination and point mutation are performed on every individual at each generation.

Table 1. Minutes to Develop Antibodies



Number of Antibodies	Minutes to produce
0	0.02
1	0.02
2	0.02
4	0.02
8	0.02
16	0.02
32	0.02
64	0.02
128	0.03
256	0.04
512	0.07
1024	0.17
2048	0.84
4096	5.49
7280	(approximately) 570

Further experiments were run using a range of training set sizes, since training is both a major contributor to the effectiveness of an AIS *and* very time-consuming. Due to the limitation on words in the regular language, all test sets consisted of 128 logs, half normal and half modified.

For efficiency of the IDS, direct measurement was used by comparing execution times. It was found that if all other factors were kept consistent, a doubling of training set size roughly corresponded to a doubling in execution time; in other words, growth was apparently linear regarding this factor. A single training set size, 16, was chosen as a result of these tests. This value was chosen because for several “applications,” it consistently produced results where an acceptable separation between normal and anomalous data could be found (i.e with low false positive and false negative rate) in relatively little time.

5.2.7 Experimental Design

Experiments were designed to demonstrate the functionality of the program over a range of input types and to illustrate the effect of feedback. Six “application”-regular expressions were developed (Table 2).

Table 2. Regular Expressions with Number of Strings Produced

Number	Regular Expression	Strings Produced
1	((a9+a10+a11) (a1 (a2+a3+a4))* a16)	360
2	(a1 a2 (a9 (a10+a11+a12))* (a13 a15)* a16)	480
3	(a1 (a2+a3+(a4+a5+a6+a7)*) a16)	342
4	(a1 a2 a6 ((a9+a10+a11) a12)* a13 (a15 a14)* a16)	480
5	(a1 (a2+a3+a11) a4 (a5+a6)* a9 (a7)* a16)	360
6	(a2 a6 a8 (a10 (a12 a14)*)* a16)	340

For each of the six regular expressions, ten test sets of 128 logs were developed. These logs were divided in two; one set became a “tuning” test set and the other became the “experimental” test set.

Experiments were done for the following conditions:

No Feedback:

- One training set of 16 self logs was used to train the system.
- One set of 128 test logs was run through the system, and results (detection rate and FP rate) were measured.
- This experiment was run for each of the 10 test sets.

Limited Feedback:

- One training set of 16 self logs was used to train the system.
- One set of 64 test logs was run through the system while providing feedback on False Positive and False Negative hits. In the case of False Negatives, the generations allowed to attempt to evolve a matching antibody were limited to 100. Results were not measured.
- The second set of 64 test logs was run, and results were measured.
- This experiment was run for each of the 10 test sets.

5.4 Results and Analysis

As noted, these experiments were performed for six different regular expressions. Table 3 displays results in terms of mean number of alerts (with standard deviation in parenthesis) without feedback.

Table 3. Mean Number of Alerts (Standard Deviation) without Feedback

	1	2	3	4	5	6
Truncation	53 (17)	166 (87)	72 (50)	373 (151)	213 (125)	321 (146)
Removal	19 (11)	43 (49)	26 (23)	51 (42)	47 (43)	21 (31)
Insertion	15 (15)	74 (110)	90 (129)	104 (110)	128 (130)	98 (168)
Replacement	10 (11)	54 (84)	46 (68)	85 (101)	86 (84)	74 (140)
All Anomalies	24 (22)	84 (98)	58 (81)	153 (168)	118 (119)	128 (175)
All Normal	5 (6)	4 (18)	13 (33)	5 (13)	13 (35)	2 (17)

Table 4 displays results in terms of mean number of alerts (with standard deviation in parenthesis) with feedback.

Table 4. Mean Number of Alerts (Standard Deviation) with Feedback

	2F	3F	4F	5F	6F	7F
Truncation	53 (20)	171 (71)	59 (29)	308 (127)	194 (92)	306 (139)
Removal	14 (10)	29 (24)	19 (13)	44 (36)	38 (31)	19 (28)
Insertion	8 (11)	60 (105)	55 (91)	73 (103)	131 (116)	103 (182)
Replacement	6 (13)	46 (79)	34 (46)	83(96)	79 (70)	85 (156)
All Anomalies	20 (24)	77 (94)	42 (55)	127 (143)	111 (101)	128 (176)
All Normal	0 (1)	2 (16)	2 (10)	2 (9)	4 (16)	0 (2)

Since results were similar, a single representative case is discussed. All logs in this case came from Expression 2: (a1 a2 (a9 (a10+a11+a12))* (a13 a15)* a16), which produced 480 words when fully enumerated using n=4.

Figure 2 shows the distribution of the number of alerts for both normal and anomalous data. It can be seen that if the number of alerts after which a log is classified as self is set quite low, the majority of the normal and anomalous logs in the test sets can be classified correctly.

Figure 3 illustrates the distribution of alerts for the different modifications. It can be seen that there is very little overlap between the number of antibodies that alert on normal logs and the number that alert on truncated logs. This trend appears to hold for the other regular expressions as well. The other modifications are less easily separable from normal data.

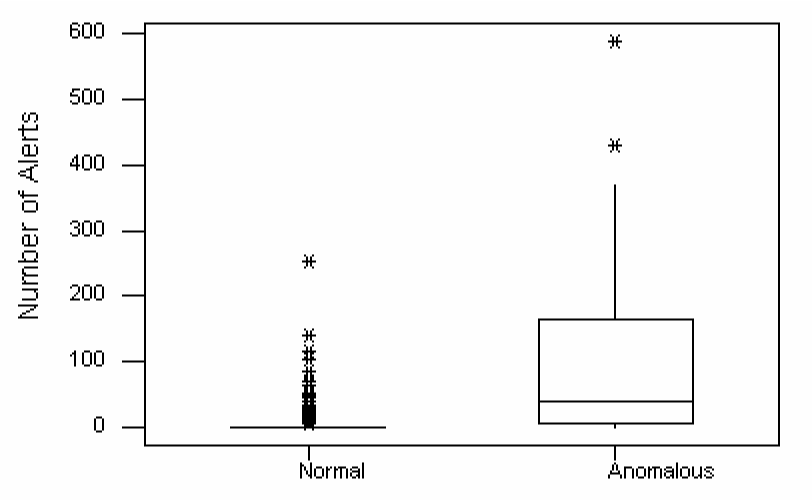


Figure 2. Number of Alerts produced for Normal and Anomalous Logs, No Feedback

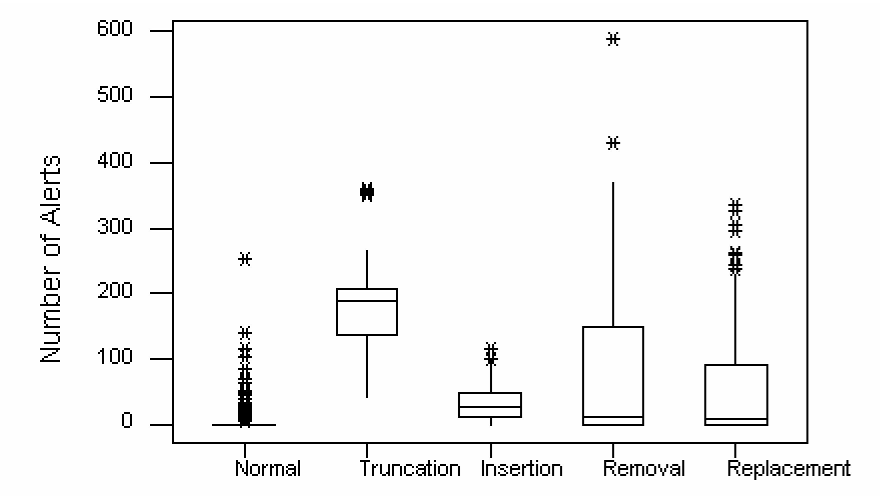


Figure 3. Number of Alerts produced for Normal and Differently Modified Logs, No Feedback

Figures 4 and 5 display the means with 90 percent confidence intervals for the different anomaly types, all types combined, and the unmodified strings. It can be seen that only the Truncation modification can be statistically distinguished from the normal data at this degree of significance in both figures. In Figure 5, the feedback appears to have enabled the Removal modification to also be distinguishable from normal; however, an examination of the data for the other regular expressions reveals that this change does not represent a consistent trend. Overall, there is no significant benefit to the use of partial feedback.

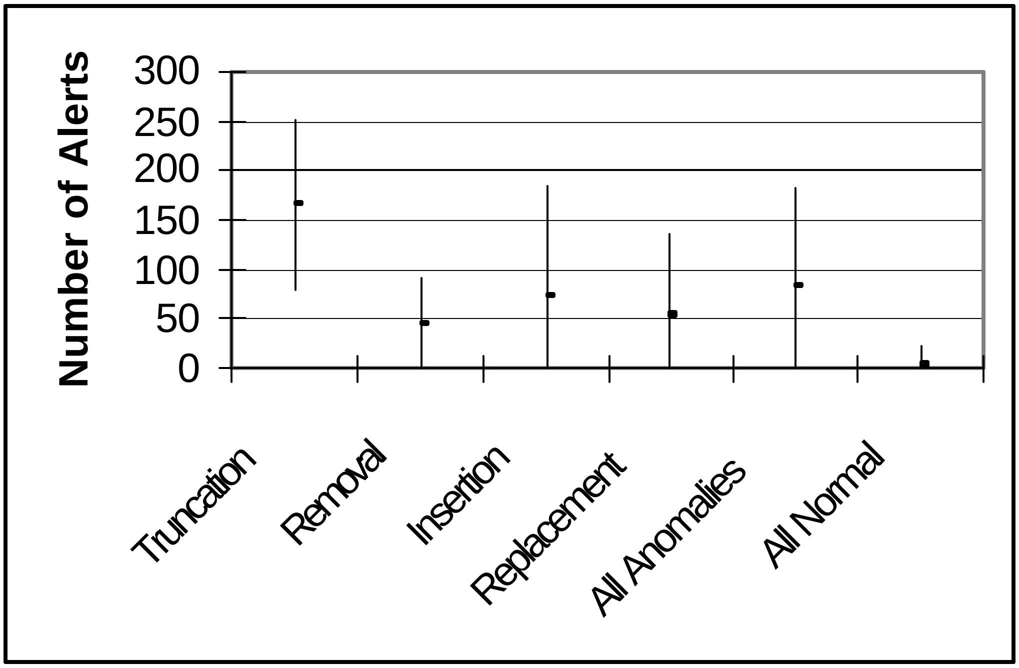


Figure 4. Means with 90% Confidence Intervals (No Feedback)

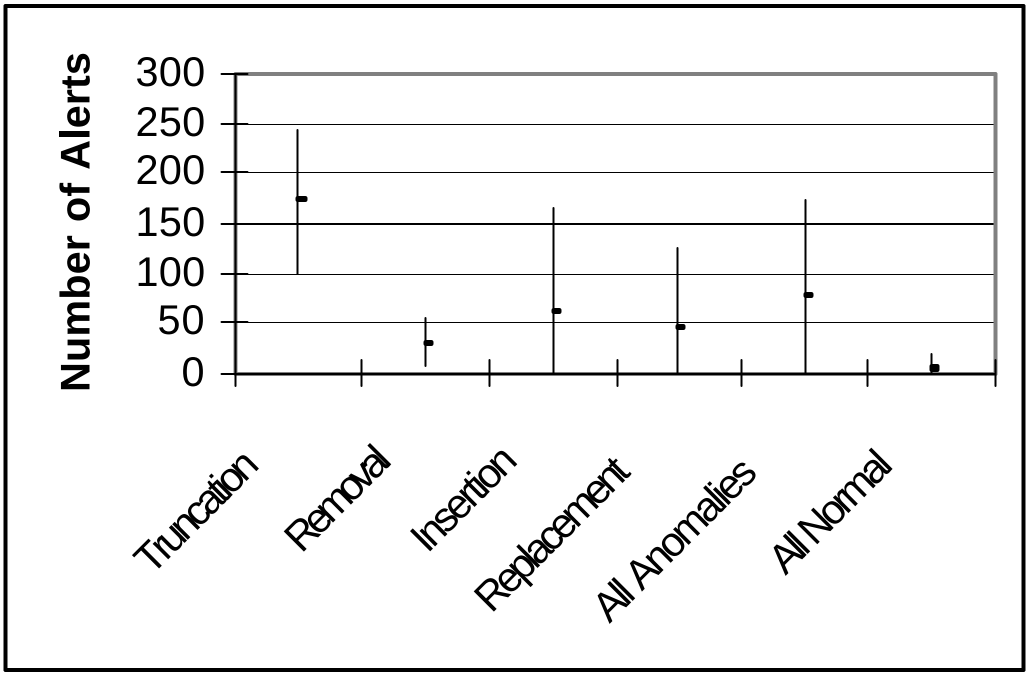


Figure 5. Means with 90% Confidence Intervals (with Feedback)

Figure 6 was produced by classifying each log according to whether the number of alerts it received was greater than the value on the x-axis, which represents a *critical decision point* (CDP). It can be seen that classification can be performed by setting the CDP to the desired value and measuring the false positive and false negative percentages achievable as a result.

If FPs and FNs are of equivalent "badness," then the best CDP for this application with these factors is approximately 3 alerts; at that point, the percentage of FPs is equivalent to the percentage of FNs – approximately 20%. This can be seen in Figure 7.

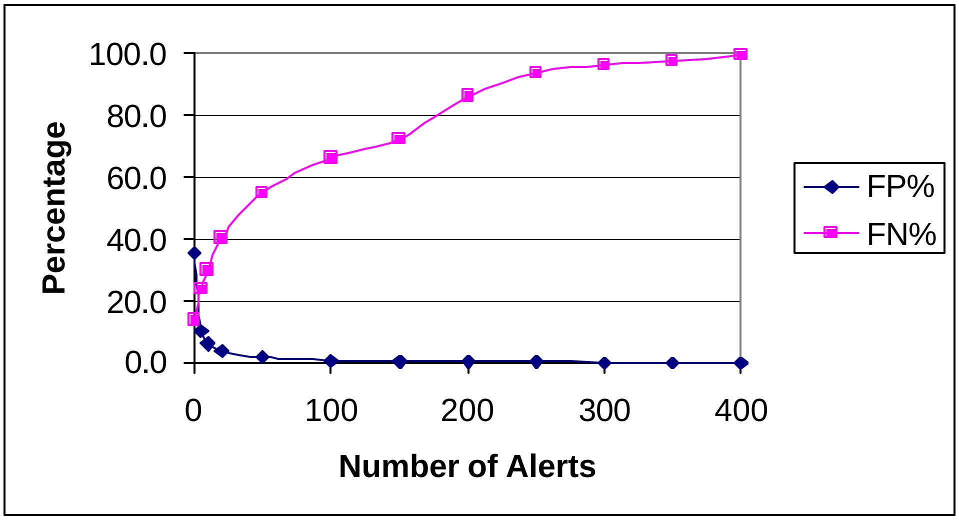


Figure 6. Control Results: Percentage vs. CDP for FP and FN

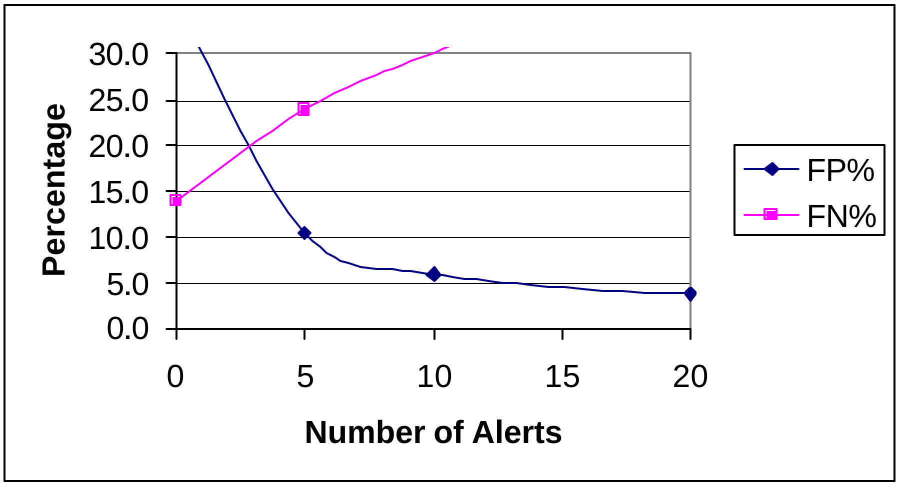


Figure 7. Control Results – Detail

Again, it cannot be shown that the addition of feedback makes the classification better in any general sense (Figure 8). The values are, for the most part, basically identical to those from the no-feedback tests, *shifted* by a few alerts to the left (Figure 9).

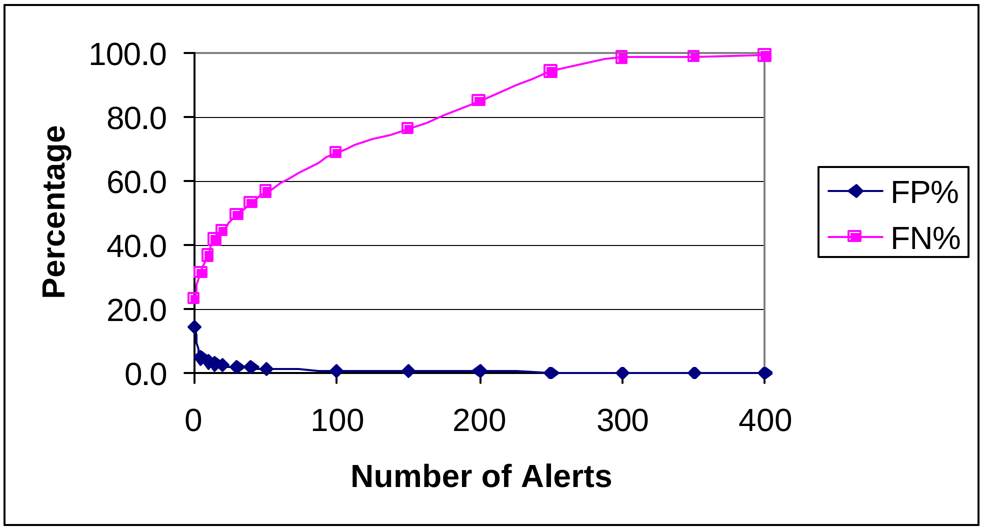


Figure 8. Limited Feedback Results: Percentage vs. CDP for FP and FN

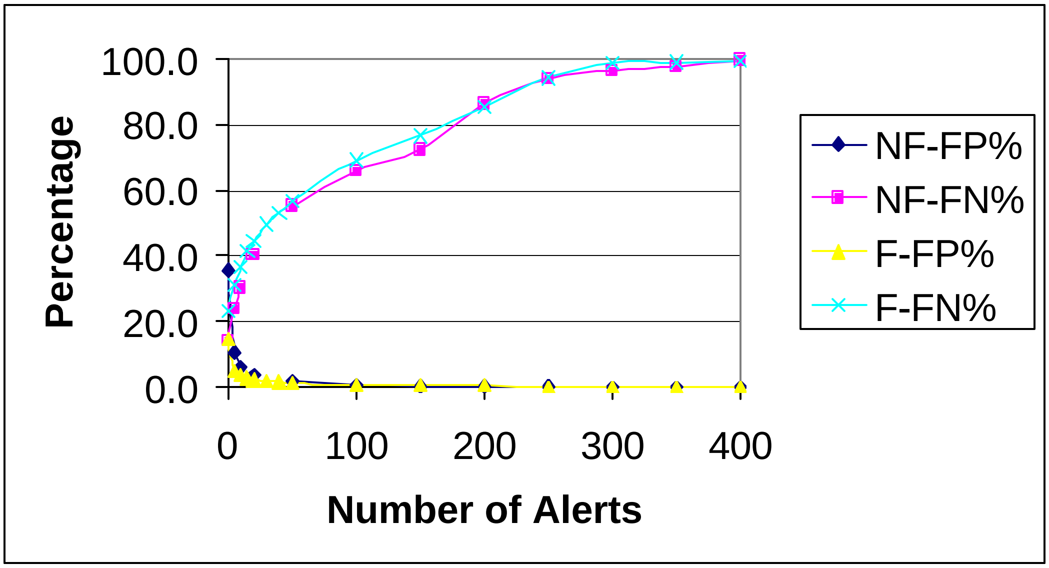


Figure 9. Percentage vs. CDP for FP and FN with (F) and without (NF) Feedback

5.3 Summary

This chapter discussed test cases as they related to the classification capability of the system. Test results are discussed as they related to the testing objectives. In these experiments, use of a partial feedback system did not appear to increase effectiveness of classification. Certain modifications are easier to detect than others - the easiest was the truncation modification, the most difficult is the replacement modification. Graphical analysis is used to determine detection and false positive rates associated with various Critical Decision Point values representing the number of alerts above which a log is classified as an anomaly.

6. Conclusions and Recommendations

6.1 Introduction

This chapter summarizes the research with respect to the objectives established in Chapter 1. The research impact and potential utility of the system are discussed. Finally, recommendations for future work are outlined.

6.2 Research Impact

Research impact can be described in terms of how the system met the objectives stated in Chapter 1.

> ***Objective I****: Identify and implement any operators necessary to describe temporal relationships among relevant events*

The relationship operators implemented were adequate to characterize and identify a significant percentage of the synthetic anomalies, especially truncations. In retrospect, a set of additional operators might have been useful to match different types of patterns. They are discussed in Section 6.3.

> ***Objective II****: Determine an appropriate method of producing descriptors for event sequences of varying complexity*

The method of using GP to produced "relationships" linked with logical operators was a success. Large quantities of descriptors were producible in reasonable time.

***Objective III**: Determine an appropriate method of classifying these relationships as "normal" or "anomalous"*

Use of the AIS paradigm was shown to be practical for isolating non-self detecting antibody descriptors. The combination of this technique with feedback and construction of the CDP curves allowed a "decision value" to be determined for a variety of different simulated intrusions.

***Objective IV**: Determine a method to extract and consolidate data that may assist human analysts in locating the point at which an anomaly occurs*

Extraction of the "Difference Essences" was a simple way of locating the point(s) at which the execution path of an application becomes anomalous. Because analysis was not limited to a fixed-size window, a plethora of information regarding the execution path as a whole could be compiled for post-mortem analysis. This information could assist not only in the identification of the attack used, but in the patching of security faults. Testing needs to be done to determine the utility of the collected information to an analyst. Further refinements, including condensing relationships and developing a method of visually representing these differences, would be of use.

6.3 Recommendations for Future Work

This research only dealt with a subset of the components that comprise the AIS model. It needs to be brought more completely within the model and further validated. Necessary alterations include running the system as truly deployed, using full feedback (not just "tuning" feedback) and testing on a representative, more realistic set of self and intrusion data. Such testing might involve setting up and instrumenting a *sendmail* server or other complex application and gathering several days worth of normal and intrusion data. An addition could easily be made to this system to allow it to behave both as a signature detector and an anomaly detector, as follows: the fully deployed system would begin with a full Antibody Database and an empty Signature Database. As antibodies detect anomalies, they could be moved or copied to the Signature Database and replaced with new antibodies.

There are many possible more dramatic extensions to this research that could improve its utility. Several examples follow:

<u>Real-time Analysis</u>

Real-time analysis has the benefit of use as part of an Intrusion *Response* System – a system that does not merely passively detect intrusions but actively attempts to prevent them from occurring. In this case, once a number of antibodies (greater than the CDP) alert, the process could be slowed or halted while a human responder is summoned to examine the situation. Such a method would be similar to the *process homeostasis* used by Somayji and Forrest [Soma00]. Real-time analysis would allow certain intrusions to be thwarted, increasing system security. At the same time, real-time analysis still carries with it the problems discussed in Section 3.2.1; i.e., possibility of the attacker becoming aware of the IDS, susceptibility to DoS, etc.

Dynamic Instrumentation

Dynamic instrumentation addresses the more common scenario wherein one is faced with developing a defensive system for a host running an application that does not have publicly available source code. Dynamic instrumentation techniques allow code to be inserted directly into the compiled executable. These methods are relatively easy to use if the developer has a solid grasp of assembly languages. One drawback to this method is that such insertion may actually create unexpected race conditions. Furthermore, this method is by nature OS specific, which means that there is no single portable system that can be widely deployed if the network is heterogeneous.

Extension of Function Set, Features Monitored

There is still a wealth of application-related data unexamined by this system; it would be sensible to determine whether monitoring other features (resource load, time delay between command execution, system calls) might be effective. For example, temporal functions, such as a "happens-within [*t* time steps]" might assist in identification of attacks that do not affect the execution path, but do affect the rate of progression through the path, such as some heap-based buffer overflows.

Functions that evaluate relationship between two events may also serve some purpose in this system. They could be used to describe sequences in a different way – instead of basing the evaluation of a relationship on the distance from a pointer, it could be based on distance from, for example, any event of a particular identifier. It may be more relevant when an event occurs relative to a set of events of the same type than that it occurs within a certain number of events of a single event. For example, the regular expression (1 2* 3) can produce the log {1 2 3}. In this case, it may be more useful to describe this log using the relationship like *there-exists <event 3> after-any <events 2>*, or something similar, instead of *next <event 3>* or *<event 3> happens-within <6> of event[0]*.

6.4 Summary

The techniques used in developing this system are shown to have utility in meeting the stated objectives. These techniques show potential utility for application to real world data. This work establishes a solid foundation for continued research in this area.

Appendix: Sample Regular Expression, Fully Enumerated

Expression 2: (a1 a2 (a9 (a10+a11+a12))* (a13 a15)* a16
Number of Strings = 480

```
1 2 9 10 13 15 16
1 2 9 10 13 15 13 15 16
1 2 9 10 13 15 13 15 13 15 16
1 2 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 13 15 16
1 2 9 11 13 15 13 15 16
1 2 9 11 13 15 13 15 13 15 16
1 2 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 13 15 16
1 2 9 12 13 15 13 15 16
1 2 9 12 13 15 13 15 13 15 16
1 2 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 13 15 16
1 2 9 10 9 10 13 15 13 15 16
1 2 9 10 9 10 13 15 13 15 13 15 16
1 2 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 13 15 16
1 2 9 10 9 11 13 15 13 15 16
1 2 9 10 9 11 13 15 13 15 13 15 16
1 2 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 13 15 16
1 2 9 10 9 12 13 15 13 15 16
1 2 9 10 9 12 13 15 13 15 13 15 16
1 2 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 13 15 16
1 2 9 11 9 10 13 15 13 15 16
1 2 9 11 9 10 13 15 13 15 13 15 16
1 2 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 13 15 16
1 2 9 11 9 11 13 15 13 15 16
1 2 9 11 9 11 13 15 13 15 13 15 16
1 2 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 13 15 16
1 2 9 11 9 12 13 15 13 15 16
1 2 9 11 9 12 13 15 13 15 13 15 16
1 2 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 13 15 16
1 2 9 12 9 10 13 15 13 15 16
1 2 9 12 9 10 13 15 13 15 13 15 16
1 2 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 13 15 16
1 2 9 12 9 11 13 15 13 15 16
1 2 9 12 9 11 13 15 13 15 13 15 16
1 2 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 13 15 16
1 2 9 12 9 12 13 15 13 15 16
1 2 9 12 9 12 13 15 13 15 13 15 16
1 2 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 10 13 15 16
1 2 9 10 9 10 9 10 13 15 13 15 16
1 2 9 10 9 10 9 10 13 15 13 15 13 15 16
```

```
1 2 9 10 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 13 15 16
1 2 9 10 9 10 9 11 13 15 13 15 16
1 2 9 10 9 10 9 11 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 13 15 16
1 2 9 10 9 10 9 12 13 15 13 15 16
1 2 9 10 9 10 9 12 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 13 15 16
1 2 9 10 9 11 9 10 13 15 13 15 16
1 2 9 10 9 11 9 10 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 13 15 16
1 2 9 10 9 11 9 11 13 15 13 15 16
1 2 9 10 9 11 9 11 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 12 13 15 16
1 2 9 10 9 11 9 12 13 15 13 15 16
1 2 9 10 9 11 9 12 13 15 13 15 13 15 16
1 2 9 10 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 13 15 16
1 2 9 10 9 12 9 10 13 15 13 15 16
1 2 9 10 9 12 9 10 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 13 15 16
1 2 9 10 9 12 9 11 13 15 13 15 16
1 2 9 10 9 12 9 11 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 13 15 16
1 2 9 10 9 12 9 12 13 15 13 15 16
1 2 9 10 9 12 9 12 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 13 15 16
1 2 9 11 9 10 9 10 13 15 13 15 16
1 2 9 11 9 10 9 10 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 13 15 16
1 2 9 11 9 10 9 11 13 15 13 15 16
1 2 9 11 9 10 9 11 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 13 15 16
1 2 9 11 9 10 9 12 13 15 13 15 16
1 2 9 11 9 10 9 12 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 13 15 16
1 2 9 11 9 11 9 10 13 15 13 15 16
1 2 9 11 9 11 9 10 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 13 15 16
1 2 9 11 9 11 9 11 13 15 13 15 16
```

```
1 2 9 11 9 11 9 11 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 13 15 16
1 2 9 11 9 11 9 12 13 15 13 15 16
1 2 9 11 9 11 9 12 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 13 15 16
1 2 9 11 9 12 9 10 13 15 13 15 16
1 2 9 11 9 12 9 10 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 13 15 16
1 2 9 11 9 12 9 11 13 15 13 15 16
1 2 9 11 9 12 9 11 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 13 15 16
1 2 9 11 9 12 9 12 13 15 13 15 16
1 2 9 11 9 12 9 12 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 13 15 16
1 2 9 12 9 10 9 10 13 15 13 15 16
1 2 9 12 9 10 9 10 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 13 15 16
1 2 9 12 9 10 9 11 13 15 13 15 16
1 2 9 12 9 10 9 11 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 13 15 16
1 2 9 12 9 10 9 12 13 15 13 15 16
1 2 9 12 9 10 9 12 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 13 15 16
1 2 9 12 9 11 9 10 13 15 13 15 16
1 2 9 12 9 11 9 10 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 13 15 16
1 2 9 12 9 11 9 11 13 15 13 15 16
1 2 9 12 9 11 9 11 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 13 15 16
1 2 9 12 9 11 9 12 13 15 13 15 16
1 2 9 12 9 11 9 12 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 13 15 16
1 2 9 12 9 12 9 10 13 15 13 15 16
1 2 9 12 9 12 9 10 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 11 13 15 16
1 2 9 12 9 12 9 11 13 15 13 15 16
1 2 9 12 9 12 9 11 13 15 13 15 13 15 16
1 2 9 12 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 13 15 16
1 2 9 12 9 12 9 12 13 15 13 15 16
1 2 9 12 9 12 9 12 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 10 9 10 13 15 16
1 2 9 10 9 10 9 10 9 10 13 15 13 15 16
1 2 9 10 9 10 9 10 9 10 13 15 13 15 13 15 16
1 2 9 10 9 10 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 10 9 11 13 15 16
1 2 9 10 9 10 9 10 9 11 13 15 13 15 16
1 2 9 10 9 10 9 10 9 11 13 15 13 15 13 15 16
1 2 9 10 9 10 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 10 9 12 13 15 16
1 2 9 10 9 10 9 10 9 12 13 15 13 15 16
1 2 9 10 9 10 9 10 9 12 13 15 13 15 13 15 16
1 2 9 10 9 10 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 9 10 13 15 16
1 2 9 10 9 10 9 11 9 10 13 15 13 15 16
1 2 9 10 9 10 9 11 9 10 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 9 11 13 15 16
1 2 9 10 9 10 9 11 9 11 13 15 13 15 16
1 2 9 10 9 10 9 11 9 11 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 9 12 13 15 16
1 2 9 10 9 10 9 11 9 12 13 15 13 15 16
1 2 9 10 9 10 9 11 9 12 13 15 13 15 13 15 16
1 2 9 10 9 10 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 9 10 13 15 16
1 2 9 10 9 10 9 12 9 10 13 15 13 15 16
1 2 9 10 9 10 9 12 9 10 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 9 11 13 15 16
1 2 9 10 9 10 9 12 9 11 13 15 13 15 16
1 2 9 10 9 10 9 12 9 11 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 9 12 13 15 16
1 2 9 10 9 10 9 12 9 12 13 15 13 15 16
1 2 9 10 9 10 9 12 9 12 13 15 13 15 13 15 16
1 2 9 10 9 10 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 9 10 13 15 16
1 2 9 10 9 11 9 10 9 10 13 15 13 15 16
1 2 9 10 9 11 9 10 9 10 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 9 11 13 15 16
1 2 9 10 9 11 9 10 9 11 13 15 13 15 16
1 2 9 10 9 11 9 10 9 11 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 9 12 13 15 16
1 2 9 10 9 11 9 10 9 12 13 15 13 15 16
1 2 9 10 9 11 9 10 9 12 13 15 13 15 13 15 16
1 2 9 10 9 11 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 9 10 13 15 16
1 2 9 10 9 11 9 11 9 10 13 15 13 15 16
1 2 9 10 9 11 9 11 9 10 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 9 11 13 15 16
1 2 9 10 9 11 9 11 9 11 13 15 13 15 16
1 2 9 10 9 11 9 11 9 11 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 9 12 13 15 16
1 2 9 10 9 11 9 11 9 12 13 15 13 15 16
1 2 9 10 9 11 9 11 9 12 13 15 13 15 13 15 16
1 2 9 10 9 11 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 12 9 10 13 15 16
1 2 9 10 9 11 9 12 9 10 13 15 13 15 16
1 2 9 10 9 11 9 12 9 10 13 15 13 15 13 15 16
1 2 9 10 9 11 9 12 9 10 13 15 13 15 13 15 13 15 16
```

```
1 2 9 10 9 11 9 12 9 11 13 15 16
1 2 9 10 9 11 9 12 9 11 13 15 13 15 16
1 2 9 10 9 11 9 12 9 11 13 15 13 15 13 15 16
1 2 9 10 9 11 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 11 9 12 9 12 13 15 16
1 2 9 10 9 11 9 12 9 12 13 15 13 15 16
1 2 9 10 9 11 9 12 9 12 13 15 13 15 13 15 16
1 2 9 10 9 11 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 9 10 13 15 16
1 2 9 10 9 12 9 10 9 10 13 15 13 15 16
1 2 9 10 9 12 9 10 9 10 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 9 11 13 15 16
1 2 9 10 9 12 9 10 9 11 13 15 13 15 16
1 2 9 10 9 12 9 10 9 11 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 9 12 13 15 16
1 2 9 10 9 12 9 10 9 12 13 15 13 15 16
1 2 9 10 9 12 9 10 9 12 13 15 13 15 13 15 16
1 2 9 10 9 12 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 9 10 13 15 16
1 2 9 10 9 12 9 11 9 10 13 15 13 15 16
1 2 9 10 9 12 9 11 9 10 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 9 11 13 15 16
1 2 9 10 9 12 9 11 9 11 13 15 13 15 16
1 2 9 10 9 12 9 11 9 11 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 9 12 13 15 16
1 2 9 10 9 12 9 11 9 12 13 15 13 15 16
1 2 9 10 9 12 9 11 9 12 13 15 13 15 13 15 16
1 2 9 10 9 12 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 9 10 13 15 16
1 2 9 10 9 12 9 12 9 10 13 15 13 15 16
1 2 9 10 9 12 9 12 9 10 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 9 11 13 15 16
1 2 9 10 9 12 9 12 9 11 13 15 13 15 16
1 2 9 10 9 12 9 12 9 11 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 9 12 13 15 16
1 2 9 10 9 12 9 12 9 12 13 15 13 15 16
1 2 9 10 9 12 9 12 9 12 13 15 13 15 13 15 16
1 2 9 10 9 12 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 9 10 13 15 16
1 2 9 11 9 10 9 10 9 10 13 15 13 15 16
1 2 9 11 9 10 9 10 9 10 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 9 11 13 15 16
1 2 9 11 9 10 9 10 9 11 13 15 13 15 16
1 2 9 11 9 10 9 10 9 11 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 9 12 13 15 16
1 2 9 11 9 10 9 10 9 12 13 15 13 15 16
1 2 9 11 9 10 9 10 9 12 13 15 13 15 13 15 16
1 2 9 11 9 10 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 9 10 13 15 16
1 2 9 11 9 10 9 11 9 10 13 15 13 15 16
1 2 9 11 9 10 9 11 9 10 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 9 11 13 15 16
1 2 9 11 9 10 9 11 9 11 13 15 13 15 16
1 2 9 11 9 10 9 11 9 11 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 9 12 13 15 16
1 2 9 11 9 10 9 11 9 12 13 15 13 15 16
1 2 9 11 9 10 9 11 9 12 13 15 13 15 13 15 16
1 2 9 11 9 10 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 9 10 13 15 16
1 2 9 11 9 10 9 12 9 10 13 15 13 15 16
1 2 9 11 9 10 9 12 9 10 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 9 11 13 15 16
1 2 9 11 9 10 9 12 9 11 13 15 13 15 16
1 2 9 11 9 10 9 12 9 11 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 9 12 13 15 16
1 2 9 11 9 10 9 12 9 12 13 15 13 15 16
1 2 9 11 9 10 9 12 9 12 13 15 13 15 13 15 16
1 2 9 11 9 10 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 9 10 13 15 16
1 2 9 11 9 11 9 10 9 10 13 15 13 15 16
1 2 9 11 9 11 9 10 9 10 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 9 11 13 15 16
1 2 9 11 9 11 9 10 9 11 13 15 13 15 16
1 2 9 11 9 11 9 10 9 11 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 9 12 13 15 16
1 2 9 11 9 11 9 10 9 12 13 15 13 15 16
1 2 9 11 9 11 9 10 9 12 13 15 13 15 13 15 16
1 2 9 11 9 11 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 9 10 13 15 16
1 2 9 11 9 11 9 11 9 10 13 15 13 15 16
1 2 9 11 9 11 9 11 9 10 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 9 11 13 15 16
1 2 9 11 9 11 9 11 9 11 13 15 13 15 16
1 2 9 11 9 11 9 11 9 11 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 9 12 13 15 16
1 2 9 11 9 11 9 11 9 12 13 15 13 15 16
1 2 9 11 9 11 9 11 9 12 13 15 13 15 13 15 16
1 2 9 11 9 11 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 9 10 13 15 16
1 2 9 11 9 11 9 12 9 10 13 15 13 15 16
1 2 9 11 9 11 9 12 9 10 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 9 11 13 15 16
1 2 9 11 9 11 9 12 9 11 13 15 13 15 16
1 2 9 11 9 11 9 12 9 11 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 9 12 13 15 16
1 2 9 11 9 11 9 12 9 12 13 15 13 15 16
1 2 9 11 9 11 9 12 9 12 13 15 13 15 13 15 16
1 2 9 11 9 11 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 9 10 13 15 16
1 2 9 11 9 12 9 10 9 10 13 15 13 15 16
```

```
1 2 9 11 9 12 9 10 9 10 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 9 11 13 15 16
1 2 9 11 9 12 9 10 9 11 13 15 13 15 16
1 2 9 11 9 12 9 10 9 11 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 9 12 13 15 16
1 2 9 11 9 12 9 10 9 12 13 15 13 15 16
1 2 9 11 9 12 9 10 9 12 13 15 13 15 13 15 16
1 2 9 11 9 12 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 9 10 13 15 16
1 2 9 11 9 12 9 11 9 10 13 15 13 15 16
1 2 9 11 9 12 9 11 9 10 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 9 11 13 15 16
1 2 9 11 9 12 9 11 9 11 13 15 13 15 16
1 2 9 11 9 12 9 11 9 11 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 9 12 13 15 16
1 2 9 11 9 12 9 11 9 12 13 15 13 15 16
1 2 9 11 9 12 9 11 9 12 13 15 13 15 13 15 16
1 2 9 11 9 12 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 9 10 13 15 16
1 2 9 11 9 12 9 12 9 10 13 15 13 15 16
1 2 9 11 9 12 9 12 9 10 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 9 11 13 15 16
1 2 9 11 9 12 9 12 9 11 13 15 13 15 16
1 2 9 11 9 12 9 12 9 11 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 9 12 13 15 16
1 2 9 11 9 12 9 12 9 12 13 15 13 15 16
1 2 9 11 9 12 9 12 9 12 13 15 13 15 13 15 16
1 2 9 11 9 12 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 9 10 13 15 16
1 2 9 12 9 10 9 10 9 10 13 15 13 15 16
1 2 9 12 9 10 9 10 9 10 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 9 11 13 15 16
1 2 9 12 9 10 9 10 9 11 13 15 13 15 16
1 2 9 12 9 10 9 10 9 11 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 9 12 13 15 16
1 2 9 12 9 10 9 10 9 12 13 15 13 15 16
1 2 9 12 9 10 9 10 9 12 13 15 13 15 13 15 16
1 2 9 12 9 10 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 9 10 13 15 16
1 2 9 12 9 10 9 11 9 10 13 15 13 15 16
1 2 9 12 9 10 9 11 9 10 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 9 11 13 15 16
1 2 9 12 9 10 9 11 9 11 13 15 13 15 16
1 2 9 12 9 10 9 11 9 11 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 9 12 13 15 16
1 2 9 12 9 10 9 11 9 12 13 15 13 15 16
1 2 9 12 9 10 9 11 9 12 13 15 13 15 13 15 16
1 2 9 12 9 10 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 9 10 13 15 16
1 2 9 12 9 10 9 12 9 10 13 15 13 15 16
1 2 9 12 9 10 9 12 9 10 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 9 11 13 15 16
1 2 9 12 9 10 9 12 9 11 13 15 13 15 16
1 2 9 12 9 10 9 12 9 11 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 9 12 13 15 16
1 2 9 12 9 10 9 12 9 12 13 15 13 15 16
1 2 9 12 9 10 9 12 9 12 13 15 13 15 13 15 16
1 2 9 12 9 10 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 9 10 13 15 16
1 2 9 12 9 11 9 10 9 10 13 15 13 15 16
1 2 9 12 9 11 9 10 9 10 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 9 11 13 15 16
1 2 9 12 9 11 9 10 9 11 13 15 13 15 16
1 2 9 12 9 11 9 10 9 11 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 9 12 13 15 16
1 2 9 12 9 11 9 10 9 12 13 15 13 15 16
1 2 9 12 9 11 9 10 9 12 13 15 13 15 13 15 16
1 2 9 12 9 11 9 10 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 9 10 13 15 16
1 2 9 12 9 11 9 11 9 10 13 15 13 15 16
1 2 9 12 9 11 9 11 9 10 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 9 11 13 15 16
1 2 9 12 9 11 9 11 9 11 13 15 13 15 16
1 2 9 12 9 11 9 11 9 11 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 9 12 13 15 16
1 2 9 12 9 11 9 11 9 12 13 15 13 15 16
1 2 9 12 9 11 9 11 9 12 13 15 13 15 13 15 16
1 2 9 12 9 11 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 9 10 13 15 16
1 2 9 12 9 11 9 12 9 10 13 15 13 15 16
1 2 9 12 9 11 9 12 9 10 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 9 11 13 15 16
1 2 9 12 9 11 9 12 9 11 13 15 13 15 16
1 2 9 12 9 11 9 12 9 11 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 9 12 13 15 16
1 2 9 12 9 11 9 12 9 12 13 15 13 15 16
1 2 9 12 9 11 9 12 9 12 13 15 13 15 13 15 16
1 2 9 12 9 11 9 12 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 9 10 13 15 16
1 2 9 12 9 12 9 10 9 10 13 15 13 15 16
1 2 9 12 9 12 9 10 9 10 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 9 11 13 15 16
1 2 9 12 9 12 9 10 9 11 13 15 13 15 16
1 2 9 12 9 12 9 10 9 11 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 9 12 13 15 16
1 2 9 12 9 12 9 10 9 12 13 15 13 15 16
1 2 9 12 9 12 9 10 9 12 13 15 13 15 13 15 16
1 2 9 12 9 12 9 10 9 12 13 15 13 15 13 15 13 15 16
```

```
1 2 9 12 9 12 9 11 9 10 13 15 16
1 2 9 12 9 12 9 11 9 10 13 15 13 15 16
1 2 9 12 9 12 9 11 9 10 13 15 13 15 13 15 16
1 2 9 12 9 12 9 11 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 11 9 11 13 15 16
1 2 9 12 9 12 9 11 9 11 13 15 13 15 16
1 2 9 12 9 12 9 11 9 11 13 15 13 15 13 15 16
1 2 9 12 9 12 9 11 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 11 9 12 13 15 16
1 2 9 12 9 12 9 11 9 12 13 15 13 15 16
1 2 9 12 9 12 9 11 9 12 13 15 13 15 13 15 16
1 2 9 12 9 12 9 11 9 12 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 9 10 13 15 16
1 2 9 12 9 12 9 12 9 10 13 15 13 15 16
1 2 9 12 9 12 9 12 9 10 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 9 10 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 9 11 13 15 16
1 2 9 12 9 12 9 12 9 11 13 15 13 15 16
1 2 9 12 9 12 9 12 9 11 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 9 11 13 15 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 9 12 13 15 16
1 2 9 12 9 12 9 12 9 12 13 15 13 15 16
1 2 9 12 9 12 9 12 9 12 13 15 13 15 13 15 16
1 2 9 12 9 12 9 12 9 12 13 15 13 15 13 15 13 15 16
```

BIBLIOGRAPHY

[Asla96] Aslam, T., I. Krsul and E. Spafford. “Use of a Taxonomy of Security Faults.” *Proceedings of the 19th NIST-NCSC National Information Systems Security Conference*. 1996. http://citeseer.nj.nec.com/aslam96use.html.

[Bish96] Bishop, M. and M. Dilger. Checking for Race Conditions in File Accesses. Technical Report CSE-95-10, University of California at Davis. 1996.

[Cros95a] Crosbie and Spafford. "Applying Genetic Programming to Intrusion Detection." *AAAI Symposium on Genetic Programming*. 1995.

[Cros95b] Crosbie, Mark and Gene Spafford. “Defending a Computer System using Autonomous Agents,” 1995. http://citeseer.nj.nec.com/crosbie96defending.html.

[Cros95c] Crosbie and Spafford. “Active Defense of a Computer System using Autonomous Agents,” 1995. http://citeseer.nj.nec.com/138521.html.

[Dasg98] Dasgupta, D., editor. *Artificial Immune Systems and their Applications*. Berlin, Springer-Verlag, 1998.

[Davi94] Davis, M.D., R. Sigal and E.J. Weyuker. *Computability, Complexity, and Languages: Fundamentals of Computer Science*. Morgan Kaufmann Publishers, San Francisco, CA. 1994.

[Denn87] Denning, D. “An intrusion-detection model.” *IEEE Transactions on Software Engineering*. 1987.

[Doyl01a] Doyle, Jon, Isaac Kohane, William Long, Howard Shrobe, and Peter Szolovits. “Agile Monitoring for Cyber Defense,” 2001. Available at http://citeseer.nj.nec.com/doyle01agile.html.

[Doyl01b] Doyle, Jon, Isaac Kohane, William Long, Howard Shrobe, and Peter Szolovits. “Event Recognition Beyond Signature and Anomaly,” 2001. Available at http://citeseer.nj.nec.com/doyle01event.html.

[Ent01] Entercept Security Technologies. Press Release: “Entercept Awarded Patent for Advanced Buffer Overflow Protection.” Press Release Available at http://www.entercept.com/news/uspr/12-10-01.asp.

[Fan01] Fan, W., W. Lee, M. Miller, S.J. Stolfo, P.K. Chan. “Using Artificial Anomalies to Detect Known and Unknown Network Intrusions.” To appear in *Knowledge and Information Systems*, Springer. Available at http://www.cs.fit.edu/~pkc/papers/icdm01.pdf.

[Foge95] Fogel, D.B. "Phenotypes, Genotypes, and Operators in Evolutionary Computation." *Proceedings of the 1995 IEEE Inernational Conference on Evolutionary Computation (ICEC'95)*. IEEE Press, New York, NY. 1995.

[Forr94] Forrest, S., et al. "Self-Nonself Discrimination in a Computer." *Proceedings of 1994 Symposium on Research in Security and Privacy*. 1994.

[Forr96] Forrest, S., et al. "A Sense of Self for UNIX Processes." *Proceedings of 1996 Symposium on Research in Security and Privacy*. 1996.

[Forr97] Forrest, S., S. Hofmeyr, and A. Somayaji. "Computer Immunology." *Communications of the ACM*, 40 (10):88-96. 1997. Available at http://citeseer.nj.nec.com/forrest96computer.html.

[Guns00] Gunsch, Gregg H. "What is Intrusion Detection?" CSCE 525: Introduction to Information Warfare Winter 2002 class CD, 2002.

[Harm00] Harmer, P. A Distributed Agent Architecture for a Computer Virus Immune System. MS Thesis, AFIT/GCE/ENG/00M-02, Graduate School of Engineering and Management, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH. 2000.

[Heit00] Heitkotter, J. and D. Beasley. The Hitch-Hiker's Guide to Evolutionary Computation: A list of Frquently Asked Questions (FAQ), Issue 8.2. 2000. Available from ftp://rtfm.mit.edu/pub/usenet/news.

[Head90] Heady R., G. Luger, A. Maccabe, and M. Servilla. "The Architecture of a Network-level Intrusion Detection System," Technical Report, CS90-20. Dept. of Computer Science, University of New Mexico. 1990.

[Hofm98] Hofmeyr, S., et al. "Intrusion Detection Using a Sequence of System Calls." *Journal of Computer Security*, 6. 1998.

[Koza92] Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press. 1992.

[Koza94] Koza, J.R. *Genetic Programming II: Automatic Discovery of Reusable Programs*. The MIT Press. 1994.

[Kunz93] Kunz, Z. and D. Fogel. "Event Abstraction", Springer-Verlag, Berlin, Germany. 2002.

[Mich02] Michaelwizc, Z. and D. Fogel. *How to Solve It: Modern Heuristics*, Springer-Verlag, Berlin, Germany. 2002.

[NSTI94] National Security Telecommunications and Information Systems Security. "NSTISSI No. 4011: National Training Standard for Information Systems Security (INFOSEC) Professionals." 1994.

[Nune00] Nunes de Castro, Leandro and Fernando Jose Von Zuben. “Artificial Immune Systems: Part II – A Survey of Applications,” Technical Report DCA-RT 02/00, 2000. ftp://ftp.dca.fee.unicamp.br/pub/docs/vonzuben/lnunes/rtdca0200.pdf

[Scam01] Scambray, J., S. McClure, G. Kurtz. *Hacking Exposed: Network Security Secrets & Solutions, Second Edition*. Osbourne/McGraw-Hill, Berkeley, CA. 2001.

[Soma00] Somayaji, A., S. Forrest. “Automated Response Using System-Call Delays.” *Proceedings of the USENIX Security Symposium (2000)*. Available at www.usenix.org/publication/library/proceedings/sec2000/somayagi.html. 2000.

[Will01] Williams, P.D., K.P. Anchor, J.L. Bebo, G.H. Gunsch, and G.L. Lamont. CDIS: Towards a Computer Immune System for Detecting Network Intrusions. *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection (RAID) 2001*. Springer-Verlag. Oct 2001.

[Zamb01] Zamboni, Diego. Using Internal Sensors for Intrusion Detection. Center for Education and Research in Information Assurance and Security (CERIAS) Technical Report 2001-42, Purdue University. August 2001.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of the collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to an penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)* 25-03-2003	**2. REPORT TYPE** **Master's Thesis**	**3. DATES COVERED** *(From – To)* Aug 2001 – Mar 2003

4. TITLE AND SUBTITLE
USING SEQUENCE ANALYSIS TO PERFORM APPLICATION-BASED ANOMALY DETECTION WITHIN AN ARTIFICIAL IMMUNE SYSTEM FRAMEWORK

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

O'Brien, Larissa A., 1 Lt, USAF

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES(S) AND ADDRESS(S)
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way, Building 640
WPAFB OH 45433-7765

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/03-15

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research (AFOSR)
Attn: Dr. Robert Herklotz
4015 Wilson Boulevard, Room 713
Arlington, VA 22203-1954
DSN: 426-6565
e-mail: robert.herklotz@afosr.af.mil

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The Air Force and other Department of Defense (DoD) computer systems typically rely on traditional signature-based network IDSs to detect various types of attempted or successful attacks. Signature-based methods are limited to detecting known attacks or similar variants; anomaly-based systems, by contrast, alert on behaviors previously unseen. The development of an effective anomaly-detecting, application-based IDS would increase the Air Force's ability to ward off attacks that are not detected by signature-based network IDSs, thus strengthening the layered defenses necessary to acquire and maintain safe, secure communication capability.

This system follows the Artificial Immune System (AIS) framework, which relies on a sense of "self," or normal system states to determine potentially dangerous abnormalities ("non-self"). A novel method for anomaly detection is introduced in which "self" is defined by sequences of events that define an application's execution path. A set of antibodies that act as sequence "detectors" are developed and used to attempt to identify modified data within a synthetic test set.

15. SUBJECT TERMS
Computer security, Intrusion detection, Immunology, Instrumentation, Data mining, Antibodies, Stochastic processes

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Gregg H. Gunsch (ENG)
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			**19b. TELEPHONE NUMBER** *(Include area code)*
U	U	U	UU	96	(937) 255-6565, ext 4281; e-mail: Gregg.Gunsch@afit.edu