Searching System Call Information for Clues: The Effects of Intrusions on Processes

THESIS

Mark Gerald Reith, First Lieutenant, USAF

AFIT/GCS/ENG/03-16

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/03-16

Searching System Call Information for Clues:
The Effects of Intrusions on Processes

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)

Mark Gerald Reith, B.S. Computer Science

First Lieutenant, USAF

March, 2003

AFIT/GCS/ENG/03M-16

Searching System Call Information for Clues:

The Effects of Intrusions on Processes

Mark Gerald Reith, B.S. Computer Science

First Lieutenant, USAF

Approved:

[illegible]	3/3/03
Dr. Gregg H. Gunsch (Thesis Advisor) Assistant Professor of Computer Engineering Department of Electrical and Computer Engineering	Date
Gary B. Lamont	3/3/03
Dr. Gary B. Lamont (Committee Member) Professor of Computer Engineering Department of Electrical and Computer Engineering	Date
[illegible]	3/3/03
Dr. Eugene H. Spafford (Committee Member) Professor of Computer Sciences Director of CERIAS Purdue University	Date

Acknowledgements

I would like to take this opportunity to thank a few people that supported me during this thesis effort.

To my wife and daughter: Thank you for your patience and understanding these long 18 months. I realize that the success of each assignment relies on your strength and patience, and that this assignment here at AFIT has been the most challenging we've ever encountered. This endeavor would not have been possible without your support and encouragement.

To Professor Gregg Gunsch: It has been a pleasure working with you to explore information assurance topics. I appreciate the guidance you've given, especially with some of my "off the wall" thesis ideas. Thank you for allowing me the flexibility to pursue my many interests. I've immensely enjoyed your information assurance courses and hope that what I take from them I will be able to serve the Air Force in the best possible way.

To Professor Gary Lamont: Thank you for sharing your profound breadth of experience and insight with me. Your attitude towards learning inspires me to go the extra mile or dig a little deeper to better understand a topic.

To Professor Eugene Spafford: I wish we had more time to converse about the current state of information assurance and the direction it is heading. I think I could learn much from your experience and insight. Thank you for being a part of my thesis experience.

To the Information Assurance Research Group (Gunsch Bunch): I cannot believe that our fun is over so quickly! Perhaps we'll all meet again as Ph.D candidates for twice as much excitement. In any case, I wish the best to each of you and I hope we have the opportunity to work together again.

To the Air Force Office of Scientific Research: I hope the ideas that are evaluated in this thesis lead to better understanding of the computer intrusions and how they affect processes. Thanks to your support, I have the opportunity to explore this issue and feed my curiosity as well as possibly expand the Air Force's view on intrusion detection beyond the conventional approaches.

Mark Gerald Reith

Table of Contents

Page

Page

Page

List of Figures

List of Tables

AFIT/GCS/ENG/03-16

Abstract

The United States Air Force extensively uses information systems as a tool for managing and maintaining its information. The increased dependence on these systems in recent years has necessitated the need for protection from threats of information warfare and cyber terrorism. One type of protection utilizes intrusion detection systems to provide indications that intrusive behavior has occurred. Other types of protection may include packet filtering, cryptography and strong user authentication.

Traditional approaches toward intrusion detection rely on features that are external to computer processes. By treating processes as black-boxes, intrusion detection systems may miss a wealth of information that could be useful for detecting intrusions. This thesis effort investigates the effectiveness of anomaly-based intrusion detection using system call information from a computational process. Previous work uses sequences of system calls to identify anomalies in processes. Instead of sequences of system calls, information associated with each system call is used to build a profile of normality that may be used to detect a process deviation. Such information includes parameters passed, results returned and the instruction pointer associated with the system call. Three methods of detecting deviations are evaluated for this problem. These include direct matching, relaxed matching and artificial immune system matching techniques. The test data used includes stack-based buffer overflows, heap-based buffer overflows and file binding race conditions. Results from this effort show that although attempted exploits were difficult to detect, certain actual exploits were easily detectable from system call information. In addition, each of the matching approaches provides some indication of anomalous behavior, however each has strengths and limitations. This effort is considered a piece of the defense-in-depth model of intrusion detection.

Searching System Call Information for Clues:

The Effects of Intrusions on Processes

I. Introduction

"The information revolution is influencing far-reaching changes in the way individuals communicate one with the other, in the way commercial transactions are conducted, in the way crises are managed and even the way nations engage in warfare. Our ability to provide for the common defense is dependent on our ability to exploit the benefits of the information revolution at the same time managing the dangers inherent in rapid technological change."

"We cannot overemphasize the need for awareness of security vulnerabilities. Awareness of the threats posed by information warfare has already demonstrated the need for security products, procedures, practices and training to protect our information systems and infrastructure from both internal and external attack."

Emmett Paige Jr., Assistant Defense Secretary for Ccommand, Control, Communications and Intelligence, brief to the Personnel Security Research Center Security Conference, McLean, Va., June 25, 1996 [Paig96]

1.1 Background

The Information Age may be characterized as an era when people became increasingly dependent on information systems in many elements of modern social, political, and economic activities. These information systems provide unprecedented communication and information processing capabilities that were not readily available more than 20 years ago. Information system technology has rapidly expanded to realize both the explicit and implicit needs in many areas of domestic life, commercial ventures, education, government programs, and military operations to name a few. The relatively painless acceptance of this technology, coupled with its immense benefits, have facilitated a dependence that is not easily or willingly broken. Unfortunately, the complexity of software systems and the diversity of different technologies have led to situations where faults in the technology introduce serious liabilities to end-users. There is a common misconception that "software flaws are a consequence of rushed programming, accidents, or incompetent deployment"

[Germ02]. The information systems security company @stake reports that "70% of security vulnerabilities in software are a result of design flaws no tool could single-handedly prevent" [Germ02]. Such liabilities are not limited to only technical faults, but attributed in some cases to configuration or explicit human faults as well [Land94, Asla96]. From buffer overflows to social engineering, these faults have been exploited by different groups of people, each with varying agendas ranging from the curious to the malicious. The CERT Coordination Center states that in 2002, there were 4,129 vulnerabilities and 82,094 incidents reported [CERT02]. The magnitude and seriousness of fault exploitation often lies in the fact that the information within the systems may have been altered, disclosed, or made inaccessible to legitimate users. The 2002 CSI/FBI Computer Crime Security Survey [Powe02] reports that "90% of respondents (primarily large corporations and government agencies) detected computer security breaches within the last twelve months," and "40% detected system penetration from the outside." Of those that quantified their losses, an estimated $455 billion dollars were reported [Powe02]. Furthermore, @stake estimates "30% to 50% of the digital risks facing IT [information technology] infrastructures are due to flaws in commercial and custom software" [Germ02]. The identification and repair of faults is a common activity today, as system developers attempt to salvage consumer confidence in their products [Robe02] and potentially mitigate responsibility for these faults. It is apparent that software exploits are a serious concern to most information technology users.

More immediate to the protection of an information system is the detection of fault exploits or attempted exploits within a layered defense strategy. Intrusion detection systems are the primary type of tool for detecting fault exploits. They use a variety of precise and probabilistic rules over a set of features to identify both attempted and successful exploits, but not necessarily to prevent exploits directly. Intrusion detection tools provide awareness to administrators of systems, allowing them to take appropriate actions. This thesis effort explores a process-based approach for detecting software exploits.

This chapter introduces the foundation of this research in terms identifying the specific problem, stating the intended objectives, describing the scope of the research, and outlining the central approach. Additionally, it describes why this research is important, to whom it is important, and how it may be used to further intrusion detection system capability. The focus of this research is outlined in Section 1.2. A concise statement of the problem is presented in Section 1.3. The research objectives and approach are described in Section 1.4. The extent of this research is identified by the scope of the thesis in Section 1.5. Finally, a description of the remaining document is provided in Section 1.6.

1.2 Research Focus

The goal of this research is to validate the hypothesis that the internal information from a single computer process may be used to effectively detect computer intrusions targeted at that process. Previous work [Forr96, Almg01, Muns01] provides some positive indication that the internal behavior of an application or process may be of use. This work is considered an extension of previous work in that it experiments with alternative internal features. For example, Forrest uses sequences of system calls to identify abnormalities in a process. This thesis effort conjectures that not only the order of system calls may be perturbed by an intrusion, but also some of the data associated with those system calls.

Because all computer programs are composed of data structures and algorithms (hence referred to as process control), internal behavior or state may be described in these terms. Data structure features include, but are not limited to, values or ranges of values of data structures within the process, intermediate values within modules [Almg01], CPU register values such as stack or instruction pointer values, values passed as parameters for functions, and results from functions. Process control features include, but are not limited to, execution path, sequences of system calls [Forr96], sequences of process specific functions, counters of module calls [Muns01], and timing between modules [Almg01]. The scope of this thesis investigation is limited to specific features identified in Section 1.5.

An application-centric view of monitoring for computer intrusions is a departure from traditional computer intrusion detection in that it focuses on the behavior of a single application rather than the behavior of the entire host system or network system. This research should be regarded as one layer in the "defense-in-depth" model, and not as an all-inclusive solution to the intrusion detection problem. As such, it complements the host-based and network-based approaches by offering a focused and refined method of detection for a single process. Even then, no claim is made that this approach can detect every type of intrusion, but rather it attempts to identify those that perturb a computer process.

This research is important because it tests features that are not traditionally considered when constructing an intrusion detection system. Chapter 2 describes a few examples of intrusion detection systems and the features used. A significant number of these systems use audit trail information generated from events on a system such as user actions, changes in system objects or error notification [Bace00, Axel00]. Generally these features are external to a process and IDSs treat the process as though it were a black-box. Previous work mentioned above begins to test internal process information, but much remains to be explored. Using internal process information is appealing because it suggests an opportunity to detect the intrusion much closer to the insertion point than some other approaches. The term insertion point is defined as the point in execution or time where the intruder begins to take control of the system. For example, in buffer overflow exploits the insertion point is the first instruction executed of the shellcode. Section 2.7.1. describes buffer overflow exploits in detail. The monitoring of internal process information may provide greater fidelity [Zamb01] in detecting intrusions than previous IDSs, as well as an opportunity to prevent some intrusions from occurring [Soma00].

In a broad sense, this work is important to every information technology user because it attempts to find better methods of detecting intrusions. As mentioned in Section 1.1, intrusions are a problem that affect many people in different areas including business, national security or personal privacy. It is hoped that this work furthers intrusion detection capability in the layered defense model as a mechanism for monitoring system processes on a host system. The monitoring of processes may also be important because it provides another avenue for detecting intrusion vectors that may introduce tainted information on a computer system.

1.3 Problem Domain

Among the many challenges in handling fault exploitation, the detection of novel exploits is near the top of the list. Novel exploits are those that are not known or protected against on a system. This term is generally only applicable to intrusion detection that uses specific rules to identify exploits. Variations of well-documented exploits may also be included in this category because the mechanisms for identifying known exploits are often not effective on the variations. Conventional approaches for identifying exploitations use either precise or probabilistic rules on a set of features to distinguish fault exploitation from normal system behavior. Because of the potential proximity of normal system behavior and exploitive behavior, care is taken to choose a set of features and behavior criteria that may identify the exploitive behavior. These rules or signatures are generally highly attuned to a particular exploit and thus not normally applicable to other exploits. The novel exploit is a unique problem because there is no information on what features may be affected, or to what extent they may be affected. The established approach [Bace00, Axel00] for dealing with this problem is to make an assumption that significantly abnormal behavior from a set of features is an exploit or attempted exploit. This is performed by comparing questionable behavior from a set of features against known normal behavior. Thus the challenge is to establish a set of features to identify novel exploits, attempted or successful, and gauge their effectiveness.

Once a set of features has been selected for monitoring, one of two approaches for detection may be used. The first approach is to allow no deviation from observed normality and identify any behavior that does not match. This works well when the set of normality observations is nearly complete or when unobserved normality happens to be insignificantly different from the observed normality. The second approach is to allow deviations from observed normality up to some threshold. This is more common because the complexity of systems makes it difficult to capture the entirety of normal behavior.

1.4 Research Objectives and Approach

The first research objective is to demonstrate that system call information can provide indications of intrusive or attempted intrusive behavior. Sequences of system calls were demonstrated by Forrest [Forr96] to indicated some intrusions, however the focus of this thesis is on the information that passes across the process/operating system boundary. Such information may include parameters passed to system calls, results of those calls, or the originating address in the process where the system call was invoked.

The second research objective is to evaluate methods for utilizing this information for the purpose of determining if an intrusion has occurred or not. Three methods are considered. The first is a direct method for exactly matching observed behavior against known normal behavior. The second is a relaxed method for matching observed behavior against some range of assumed normal behavior. Finally, the third method applies immunology concepts to build rules of non-self and identify observed behavior that matches the non-self rules. These rules may be stochastically generated and expanded for identifying non-self. Details of these methods are described in Chapter 3 and Chapter 4.

This research is designed to address the detection of the novel exploit. In the context of a layered defense, signature-based detection may be used to identify known exploits and anomaly-based detection may be employed to provide some coverage in identifying all other exploits. Technical faults are the intended type of exploit, although other exploits may also be detected.

1.5 Scope

The nature of the computer intrusion detection problem is large and complex, even when the focus of the research lies exclusively on process attributes. For this reason, the scope of this research is limited to four pieces of data related to system calls. Forrest [Forr96] demonstrated that sequences of system calls were useful indicators for process intrusions. Her reasoning for initially choosing this feature relied on the conjecture that to make any significant impact on the system, system calls would be necessary. Along the same lines, the features used for this research are the system call name, the instruction address of the system call, the parameters passed, and the result returned. Other features were carefully considered and described in Chapter 3.

Experimentation is also limited to carefully reconstructed intrusions and intrusion attempts. This decision was made after considering two issues. First, there is currently a lack of a standard testing suite of intrusions. Previous work has used a select palette of UNIX/BSD/Linux applications such as ps, eject, ftp [Eski00], xlock, named [Warr99], and lpd, ftp, sendmail [Forr96]; however, the details of how the intrusions were performed are rarely published. The intrusion detection community does have network traffic data developed by Lincoln Laboratory at the Massachusetts Institute of Technology (MIT) [Lipp99], however it is difficult and unwieldy to use it for this research because it was specifically designed for testing network-based intrusion detection systems. It encompasses data for multiple TCP/IP connections ranging over several processes including HTTP, SMTP, and FTP traffic. Second, there are categories of intrusions established formally or informally by the intrusion detection and hacker communities such as buffer overflows, race conditions, and string formatting exploits. Instead of using specific exploits that may be selectively chosen to work well within a particular intrusion detection scheme, generalizations of exploits are built and tested because the fundamentals of the exploit class are the defining criteria of the category. Details of the experimentation and test module construction are described in Chapter 5.

1.6 Executive Overview

This document provides a detailed discourse on process-centric intrusion detection research. Chapter 2 presents a literature review of the intrusion detection problem, traditional and modern approaches addressing this problem, a brief taxonomy of intrusions, a discussion on problem set generation, and specific previous research on process monitoring. Chapter 3 is a high-level description of the problem and algorithm domain specifications. Chapter 4 is a low-level and detailed account of implementation. Chapter 5 presents the design of experimentation, construction of problem sets, results of experimentation, and an analysis of the results. Finally, Chapter 6 describes conclusions derived from the analysis, limitation of this approach, and suggestions for future expansions of work.

II. Background & Literature Review

This chapter provides background information in the form of a literature review on a variety of subjects related to intrusion detection. Definitions related to intrusion detection are specified in Section 2.1. A description of intrusion detection systems is provided in Section 2.2. Characteristics and challenges of intrusion detection systems are outlined in Section 2.3 and Section 2.4. Traditional and modern approaches to intrusion detection are described in Section 2.5 and Section 2.6 respectively. A brief taxonomy of security faults is described in Section 2.7. Background information on artificial immune systems and how they apply to intrusion detection is explained in Section 2.8. Finally, an overview of evolutionary computation is described in Section 2.9.

2.1 Definitions Related to Intrusion Detection

This section provides several definitions related to intrusion detection. Different viewpoints from previous work provide diverse background information on this topic. The purpose of this information is to describe the context used in following chapters.

2.1.1 Intrusion. The traditional and widely accepted definition of an intrusion is "any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource" [Head90]. This succinct definition reflects the idea that intrusions accomplish one or more of these three actions: to add, modify, or remove information from a system; to view or copy the information; or to prevent legitimate user from accessing information or the information system. Mukherjee [Mukh94] expands this definition to include the "unauthorized use of resources." This includes even benign activity such as utilizing excess processing power or storage space. Balasubramaniyan [Bala98] develops the definition of intrusion even further by observing that the term also includes "insider misuse" as well as the traditional exterior threat. A more recent publication attempts to refine these definitions to "a sequence of related actions by a malicious adversary that results in the occurrence of unauthorized security threats to a target computing or networking domain"

[Amor99]. This definition has a serious flaw in it because it assumes that the intent of a user is known or determinable. A more sound definition of an intrusion is any action or result of an action that contradicts the security policy on a particular computer system [Bace00]. Security policies may be in one of two forms. A formal security policy is a mathematical description of what is allowed or not allowed on a computer system, whereas a procedural security policy is a high-level description of allowed/restricted activity [Bace00]. This definition is superior to previous definitions because it removes any ambiguity in determining intrusions since they are explicitly described. It also allows people on different computer systems to define intrusive behavior with relation to the purpose of the computer, since one definition may not fit all.

2.1.2 Intrusion Detection. Often the term intrusion detection is defined without necessarily a clear idea of what an intrusion is. There are many definitions of intrusion detection in the computer security community, each with a slightly different perception of what should be monitored. SANS [SANS02] defines intrusion detection as "art of detecting inappropriate, incorrect, or anomalous activity." This is example of an ambiguous definition because there is no information regarding what 'inappropriate' means. In addition, the term 'incorrect' implies that errors in the system are the result of intrusions. A more robust definition from Mukherjee [Mukh94] defines intrusion detection as "the problem of identifying individuals who are using a computer system without authorization (i.e., 'crackers') and those who have legitimate access to the system but are abusing their privileges (i.e., the 'insider threat')." Finally, Bace [Bace00] defines intrusion detection as "the process of monitoring computer networks and systems for violations of security policy." This potentially expands previous definitions of an intrusion by explicitly listing what is authorized or denied on an information system.

2.1.3 Intrusion Detection Models. Axelsson's [Axel00] survey and taxonomy of intrusion detection systems describes three models of intrusion detection. While the purpose of all three models is to detect intrusions, the specific methodology and immediate objectives differ. The three models are anomaly detection, misuse or signature detection, and compound or signature inspired detection.

2.1.3.1 Anomaly Detection. The anomaly detection model was introduced by Denning [Denn87] but coined by Mukherjee [Mukh94]. The original basis of this model was to compare observed user behavior against that user's historical profile with the assumption that statistical differences may indicate an intrusion or attempted intrusion. It was believed that "security violations could be detected from abnormal patterns of system usage" [Denn87]. Later systems expanded anomaly detection from abnormal user behavior to abnormal system behavior. "This detection principle thus flags behavior that is unlikely to originate from the normal process, without regard to actual intrusion scenarios" [Axel00]. Unfortunately, because this model relies on a probabilistic determination of whether an intrusion occurred or not, there exists an amount statistical error. These errors manifest as false positives and false negatives and they are a key measurement of anomaly detection performance. A false positive occurs when the detection system asserts that intrusive behavior has occurred when in reality it has not [Cros95]. Conversely, a false negative occurs when the detection system fails to alert on genuine intrusive behavior [Cros95].

Anomaly detection may be further categorized as static or dynamic. Static anomaly detection is "based on the assumption that there is a portion of the system being monitored that should remain consistent" [Jone99]. It may use information about the structure of the resource to identify changes that might indicate an intrusion. Comparing checksums of files might be considered static anomaly detection. Conversely, dynamic anomaly detection "requires distinguishing between normal and anomalous activity" [Jone99]. It identifies anomalous behavior based on a comparison of current behavior against a base profile. Statistical analysis is used to determine if the difference is significant to warrant an alarm.

2.1.3.2 Signature Detection. The signature detection model uses specific rules to identify actions that exploit known vulnerabilities. This model "looks for patterns known to cause security problems" [Mukh94]. Synonymous with the term 'misuse detection,' it is largely associated with the detection of insider misuse although there is no reason why it is exclusively used within. This type of detection generally has a high detection rate and a low false positive rate because the signatures are attuned to the specific criterion that characterizes a particular intrusion. Examples of signature detection include state-modelling, expert systems, and string matching [Axel00].

2.1.3.3 Hybrid Detection. The hybrid detection model combines information from both anomaly and signature detection models. The result is a system that may "form a compound decision in view of a model of both the normal behavior of the system and the intrusive behavior of the intruder" [Axel00]. Theoretically this detection model should provide a better indication of intrusions because it has information on "the patterns of intrusive behavior and can relate them to the normal behavior of the system" [Axel00]. It provides the flexibility for catching new intrusions (anomaly detection) with the stability and reliability for catching known ones (signature detection).

2.2 Intrusion Detection Systems

Intrusion detection systems (IDS) are the tools specifically designed to perform intrusion detection. There is an assumption made in these systems that complete or partial identification of intrusions is possible. "Intrusion detection systems are based on the belief that an intrusion will be reflected by a change in the 'normal' patterns of resource usage" [Head90]. IDSs may focus on detecting the cause of an intrusion or the effect of one. These systems may attempt to detect a set or sequence of user actions that result in an intrusion, or they may attempt to detect a set or sequence of system behaviors that result from the intrusion. The former approach focuses on identifying an intrusion before or as it occurs, while the latter approach focuses on identifying it during or after it occurs. Anomaly detection or signature detection may be utilized in either case. Examples of different types of systems are listed in Section 2.5 and Section 2.6.

2.2.1 Intrusion Detection Systems Components and Architecture.

Intrusion detection systems may be as simple or complex as the resource they monitor. Most systems are composed of three basic components: "an information source that provides a stream of event records, an analysis engine that finds signs of intrusions, and a response component that generates reactions based on the outcome of the analysis engine" [Bace00]. The information source may be "network data, security logs, operating system kernel logs" [Axel00] or any other information that may be gathered. This information is processed by an analysis component that makes a determination of intrusion using either the anomaly or signature detection approaches. Finally, the response component performs some action based on the results of the analysis mechanism.

Closely related to components is the architecture of the system. Architecture refers to the design issues of data collection and data analysis [Axel00]. Centralized architectures transport audit data or analysis information to a few key locations for processing. Decentralized architectures rely on a distributed number of data storage and analysis mechanisms. Architectural decisions play a significant role in the security and integrity of the intrusion detection system. Centralized systems contend with moving information securely to the central locations whereas decentralized architectures generally do not.

2.2.2 Intrusion Detection System Scope.

Another characteristic that defines an intrusion detection system is the scope of monitoring. These include host-based, network-based, application-based, and target-based monitoring systems [Bace00].

2.2.2.1 Host-Based Monitoring.

Host-based systems utilize a type of monitoring that is concerned with the detection of intrusions that are within the computer system. Often the focus is on "operating system audit trails and system logs" [Bace00]. It may also include integrity checks of files and monitoring for unusual processes [SANS02]. Ranum [Ranu01] describes the advantages of such systems including the ability to correlate users and programs with intrusive activity, provide detailed information about the condition of the system during or after an attack, and the ability to focus on attacks directed at the host machine. Some disadvantages [Ranu01] include the susceptibility of the IDS software becoming compromised, the reliance on the host operating system not crashing, and coverage of only one computer system per IDS.

2.2.2.2 Network-Based Monitoring. Network-based systems utilize a type of monitoring that focuses on examining network packets for suspicious activity [Bace00]. The advantages [Ranu01] of this type of monitoring include a large monitoring coverage of multiple hosts, the ability to view the packets as only data and not control information such that they may be observed without triggering an intrusion, and the ability to be imperceptible to attackers. The drawbacks [Ranu01] from this approach include the lack of determining if the intrusion has any effect on a host system, an excessive workload because it monitors everything on the wire, and the lack of coverage for intrusions originating at the targeted host system.

2.2.2.3 Application-Based Monitoring. Application-based monitoring “collects data from running applications” [Bace00]. They have a narrowed perspective to detect only the events related to a particular application. The data sources used for this monitoring include “application event logs and other data stores internal to the application” [Bace00]. One example of this type of monitoring was examined by Almgren [Almg01] by integrating a logging mechanism into a web server application and analyzing these logs of internal execution. This is an interesting approach because it provides more information to the analysis component and raises the possibility of be building a “preemptive IDS” [Almg01]. One of the potential disadvantages is performance impact on the host.

2.2.2.4 Target-Based Monitoring. Target-based monitoring focuses on the state of system objects rather than the activity applied to the objects [Bace00]. This is accomplished by applying cryptographic hash functions to the objects and monitoring for changes based on given policy [Bace00].

2.3 Characteristics of Intrusion Detection Systems

There has been literature written to characterize how a good intrusion detection system should behave. Crosbie and Spafford [Cros95] describe a number of desirable characteristics of intrusion detection systems:

1. It must run continually without [or with minimum] human supervision.
2. It must be fault tolerant in the sense that it must survive a system crash and not have to have its knowledge base rebuilt at restart. It must be able to recover and continue operation.
3. It must resist subversion. The intrusion detection system can monitor itself to ensure it has not been compromised.
4. It must observe deviations from normal behavior.
5. It must impose minimum overhead and impede on the user as little as possible.
6. It must be easily tailored to the system it monitors.

These characteristics are important because they describe some of the aspects that a system should address in addition to the primary goal of intrusion detection.

2.4 Intrusion Detection Challenges

The complexity of intrusion detection systems has led to some significant challenges. Some of these challenges are the result of the large number of systems that require coverage, while others are from setting standards for the intrusion detection community. Bace [Bace00] identifies some of these current challenges as:

1. Scalability over time and space.
2. Management of intrusion detection systems (configuration, deployment, etc).
3. Sensor management and control.

4. Investigative support - IDS should tell incidents occurred, isolate the entry points of the intruder, determine the means utilized to accomplish the intrusion, and determine the effects of the intrusion on the system and data.
5. Performance loads.
6. Reliability.
7. False positives/negatives in anomaly detection.
8. Interoperability with other intrusion detection systems.
9. Audit trail standards.

This list of challenges is not all-inclusive, however it does give insight into the complexity of these systems. This may become evident from our examination of a few select examples of implemented intrusion detection systems. The focus of these examples is to outline previous work on host-based anomaly detection systems.

One of the challenges listed describes the challenge of dealing with the high number of false positives/negatives associated with anomaly detection. Maxion and Tan [Maxi00] provided research on determining the performance characteristics specifically for anomaly detection. Their work is important because they were able to solidify the types of situations where anomaly detection would be the most applicable. In particular, they concluded that the relative entropy in data has significant impact on anomaly detection performance. The relationship between the regularity of the data and the performance of anomaly detectors was clearly apparent. As regularity decreased, so did performance (increase in false alarms). For example, given a stable (high regularity) of legitimate data, the introduction of anomalous data was easily detected. However when the data was characterized as having low regularity, it became more difficult to discriminate the anomalous data from the noise in the data. Therefore, the stability of legitimate data is an important factor when applying anomaly detection.

2.5 Conventional Approaches to Intrusion Detection

An extensive amount of work in the area of intrusion detection has built up over the last 20 years. Some of the early work in intrusion detection has been instrumental in defining the successes and limitations of intrusion detection. The following systems are conventional approaches to intrusion detection.

2.5.1 Intrusion Detection Expert System. Intrusion Detection Expert System (IDES) and Next Generation IDES (NIDES) [Denn87, Ande95, Bace00, Axel00] employ user profiles to compare current behavior to past behavior in terms of different metrics. Such metrics may include file access, CPU usage, hour of use, etc. Through the use of statistical analysis, the system attempts to determine abnormal behavior. From this abnormal behavior, a rule-based expert signature analysis component would assist in determining if a security violation occurred. The expert component was useful to minimize the effect of gradual changes becoming normalized in the anomaly detection component. A third component, the resolver, uses information from these two components to filter and report alerts. This top-level system uses audit trail information from interconnected hosts utilizing C2 auditing. This is an example of a host-based, anomaly detection system in a centralized processing and management configuration. The problem with this architecture is that it relies heavily on the input from host system being transmitted to a centralized location for decision-making. Unless further steps are taken to secure the communications, this is susceptible to a 'man-in-the-middle' attack, where fictitious audit trails could be generated from a compromised (or even foreign) host to make the system believe that the host is secure. More importantly, the network is continuously flooded with audit data from all of the protected hosts [SANS02]. This presents two scalability problems related to bandwidth limitations. First, new hosts may not be added because the network cannot handle additional traffic from the host, and second, the metrics collected may not be extensible because the data load currently produced cannot be handled by the network or the server components. Extensibility is an important feature of intrusion detection systems because as new methods of intrusion are discovered, the system may need to collect further data to catch them. Distributed Intrusion Detection System (DIDS), Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD), and Architecture and Rule-based Language for Universal Audit Trail Analysis (ASAX) are similar host-based, top-level, audit trail reliant approaches.

2.5.2 Multics Intrusion Detection and Alerting System. Multics Intrusion Detection and Alerting System (MIDAS) [Bace00, Axel00, Cros03] utilizes a heuristic detection strategy. Through the use of four tiers of expert rule sets, the system makes determinations from the audit trails of both user profiles and system activity. Built for a mainframe computer, this system is an example of a host-based, anomaly detector. The advantage of this system is that it incorporates information from users as well as system activities in the decision-making process. Because it was designed for a particular architecture, it may have been able to make more deductions about the state of the machine. It may have been able to exploit the known structure of the operating system, whereas a platform independent detection system cannot make the same assumptions.

2.5.3 UNIX State Transition Analysis. UNIX State Transition Analysis (USTAT) [Bace00, Axel00, Ilgu95] uses state transition analysis to determine if a host is in a compromised state based on the audit trail supplied by the host. The idea is that by monitoring the changes in state of the host via audit trails and attempting to fit this data into state diagrams of known intrusions, this system can track the system heading into a compromised state. The strength of this approach is that "state transition diagrams identify precisely the requirements and compromise of the penetration, and list only the key actions that must occur for the successful completion of the penetration" [Ilgu95]. This is similar to creating finite state machines that recognize a pattern of events or actions. A notable limitation is that the host cannot always follow the diagram if the audit trail does not record the necessary information. USTAT uses the C2 audit mechanism BSM. This system is an example of a host-based, signature detection system. Again a top-level reliance on audit trail information exists to make decisions. A somewhat similar type of system is the Intrusion Detection In Our Time (IDIOT) that bases decisions on petri-nets to determine intrusion, yet still relies on audit data.

2.6 Modern Approaches to Intrusion Detection

In the previous section, several examples of systems that take different approaches to the challenge of intrusion detection were examined. This proposal builds upon and is influenced by certain prior works. Among these include Forrest on computer immunology; Harmer and Williams on the computer defense immune system; Maxion and Tan on performance characteristics of anomaly detection; Zamboni on internal sensors; Munson on the application of internal sensors; and finally Somayaji and Forrest on internalizing the whole intrusion detection system within the operating system kernel.

2.6.1 Host-Based Artificial Immune System. Forrest and her students have done a considerable amount of work in the area of computer immunology. Computer immunology is the application of the biological immune system model on computer systems [Forr96b]. The model focuses on the entities that identify foreign substances in the natural immune system. These entities are referred to as antibodies and they recognize specific antigens (foreign proteins) based on their physical molecular structure. Equally as important, these antibodies are crafted to physically ignore healthy native cells. This same type of idea to recognize non-self antigens while ignoring self can be applied to computer virus and computer intrusion detection systems [Forr94]. She applies this model to UNIX processes to determine if a short sequence of system calls can be analyzed for intrusion detection [Forr96]. By creating random sequences of system calls and discarding them if they match too closely to legitimate (self) processes, she emulates the negative selection process used in the biological model. The sequences (antibodies) that do graduate past this censoring process are used to detect anomalous sequences of system calls through partial matching. The idea is to create enough of the right antibodies to cover non-self and then use these antibodies to detect abnormal sequences of system calls. This approach provides an alternative means of detecting anomalous behavior to the traditional profiling methodology.

2.6.2 Network-Based Artificial Immune System. Harmer and Williams [Harm02, Will01a] apply the computer immunology model or artificial immune system (AIS) to intrusion detection. The result is the idea of a computer defense immune system (CDIS) that addresses both computer virus detection as well as network-based intrusion detection. In both cases, antibodies are generated from random strings and put through negative selection. Those that survive are used to create coverage of non-self space. Later, affinity maturation attempts to increase the coverage of this space using existing antibodies. During operation, the antibodies are used to imperfectly match questionable data and determine if the data is to be regarded as self or non-self. The matching function is not always required to be exact, but is based on how the antibodies were created. The virus detection system uses a binary string comparator with a sliding window, whereas the network-based intrusion detection system may use a straight comparator or other means because the antibody is guaranteed to be in non-self space. As far as the types of data matched against the antibodies, the virus detection system would test a particular file, whereas in the network-based intrusion detection system, the data would be the packet headers. Regardless, a process called costimulation is used to reduce the number of false positives by requiring multiple antibodies to respond to the data.

2.6.3 Embedded Sensor Project. Zamboni [Zamb01] has done some work relevant to misuse detection using internal sensors. Through the careful examination of documented intrusions and source code, he was able to pinpoint particular pieces of data and behavior that were significant to detecting different types of intrusions. The goal was to find both specific and generic sensors that could detect various intrusions. He also addressed the fidelity problem. This occurs when intrusion detection systems assume the behavior of a system from misinterpreted or piecemeal audit trails.

The previous section examined several systems that rely heavily on the correct interpretation of native audit mechanisms. Zamboni claims that by placing the sensors and detectors within the source code of the operating system and other applications, the analyzer component has direct access to the right data in order to make an accurate decision. Another problem he addresses is the reliability problem, where an intrusion detection system (or perhaps only the client portion) may run as an application on the host. By allowing it to be a separate process, it becomes susceptible to tampering. Again, by placing the system within the source code, it becomes more difficult to subvert or tamper. Zamboni implemented 130 attack specific sensors and 23 generic sensors. His work demonstrated that an intrusion detection system could be built using internal sensors and that some new intrusions can be detected through the use of generic sensors.

2.6.4 Cylant Secure. Munson and Wimer [Muns01] apply the idea of internal sensors to instrument applications and build profiles of function and module usage. The basic premise was that the normal behavior of an application can be baselined and that intrusions would register as deviations from the base profile. Through the insertion of 3,300 internal sensors into a Linux operating system, they argued that not the sequence but the frequency of module calls within a fixed time period could distinguish abnormal behavior. In other words, they used counters to measure how often a module is called within a fixed period. A module is defined as being a system call, a function of an application, or a segment of code. Munson and Wimer use three layers of abstraction to describe why this approach should work. These three layers are operations, functions, and modules. Operations are features that allow users to control the host system. Functions are the specific components that make operations occur, while modules are the code segments that actually perform the work. Through a mapping of user operations to system functions, and system functions to modules, they were able to identify user operations based on the modules and functions that were called. In other words, by observing how often modules are executed, their intrusion detection system can determine the particular user operations performed. This is a prime example of how internal sensors have been used to implement a host-based system that is not dependent on external audit trails to determine anomalous behavior.

2.6.5 Process Homeostasis. Finally, Somayaji and Forrest [Soma00] also apply the idea of internal sensors in their "process homeostasis" (pH) approach. This approach utilizes the same system call anomaly detection as prior research, but in this case it attempts to detect an intrusion/anomaly and respond by delaying or aborting abnormal system calls. The significance of this system is that it required almost the entire system to be placed within the host operating system. Somayaji explains that audit mechanisms were generating "voluminous log files, which are expensive to create, and even more expensive to analyze" as well as not capturing all of the system calls. Furthermore, in order to exercise the responses, the system needed to be within the kernel. This is yet another example of how audit mechanisms are not adequate for modern intrusion detection systems.

2.7 Security Faults

Several researchers [Land94, Asla96, Bish99] have produced taxonomies of security faults in order to better understand and classify computer security flaws. They have identified several categories of security faults in software based on the reason that the exploit or error was allowed to occur. These security faults are important to understand because they are often the target and mechanism of exploits. A single fault or combination of faults coupled with certain conditions may lead to opportunities for intruders. The categories of security faults as described by Aslam [Asla96] are:

1. Synchronization Errors - This includes errors involving the dependence on the relative timing of two events, the synchronization of events, or the assumption of events being atomic. Race condition exploits are a prevalent type of synchronization error.

2. Condition Validation Errors - This includes errors involving boundary checking, lack of input validation, resource access rights, and resource limit checks. Buffer overflows and string formatting exploits are common condition validation errors.

3. Configuration Errors - This includes errors involving the incorrect installation and setup of software.

4. Environment Faults - This includes errors involving the usage of software within a given environment. The coupling of the software and the environmental conditions yields the fault.

All of the faults listed except configuration errors are because of programming errors. "Faults or bugs are the manifestations and results of errors during the coding of a program" [Pan92]. These faults become apparent when a set of input is provided during the execution of the software. The results of these faults can and have been researched in great detail by the hacker community for a variety of purposes including exploitation. This is evident from the detailed articles written on not only the cause of the software fault, but also methods for utilizing this information to increase system privileges [Phra02, Shel02, Inse02, Teso02]. The following sections outline several common software faults categories and corresponding examples of exploits. These include stack buffer overflow faults, heap buffer overflow faults and race condition faults. This list not all-inclusive and there exist many more types of software faults, however these represent a majority of exploited faults [CERT02, MITR02].

2.7.1 Buffer Overflow Faults. Buffer overflows are the leading type of software fault exploit [Cowa00, Laro01]. They are predominating for several reasons. One reason is that there is a lack of buffer boundary checking in many programs, often attributed to programmer oversight. Another reason is that the model of process execution allows process control data to be intermixed with user input data. Finally, the most significant reason is that the exploit allows intruders to execute arbitrary code with the exploited software's privileges.

A buffer overflow is the condition that results from writing more data to a structure than space has been allocated. The data written beyond the allocated space generally overwrites process control information, resulting in the process utilizing fraudulent control information. A buffer overflow exploit is the exploitation of the buffer overflow condition that allows arbitrary code to be executed, modifies internal variables, or denies service. This exploit generally occurs in the stack or heap sections of a process image.

2.7.1.1 Stack Buffer Overflow Faults. A stack buffer overflow exploit occurs when carefully constructed user input or environmental variable overwrites process control data with an address that points to shellcode. Shellcode is a small and lean binary program that performs some action. This action is usually to create a root level user shell (hence the name shellcode), but it also may refer to other actions as well such as shutting down the system, changing permissions on restricted files, elevating a user's privileges, etc. At some point the process follows that pointer and begins to execute the shellcode and fulfilling the exploit. In order to execute the shellcode with root privileges, the target program needs to run with root privileges. This may be accomplished by selecting a program that is set user identification (SUID) owned by root, or a daemon process that already runs as root.

To understand how this occurs, some background information [Irvi93] on how the stack works is required. A stack is a data structure that stores data in a 'last in, first out' manner by pushing data onto it and popping it off in reverse order that it was pushed. A stack pointer is a register in the CPU that maintains the location of the end of the stack. For simplicity, assume this explanation uses a 32-bit, x86 architecture. On this architecture, the stack grows down from high to low addresses. During the execution of a process, the stack is used to hold both process control and local variable data for function calls.

A function call proceeds in a systematic manner [Irvi93]. Prior to the function call, arguments of the function are pushed onto the stack in reverse order along with the return address. Within the function call, the current frame pointer (also called base pointer) is pushed onto the stack and the base pointer is assigned the address of the stack pointer. The return address is the location where execution should resume after the function has finished and the frame pointer is the address that is used as a reference to calculate addresses of arguments and variables. Next, storage is allocated for local variables by adjusting the stack pointer. When the function finishes, the stack pointer is adjusted to de-allocate the local variable storage. Then the stack pointer is assigned the value of the base pointer and the former base pointer is popped from the stack into its register. Next, the return address is popped off the stack and into the instruction pointer register. Lastly, the stack pointer is adjusted by the size of the arguments and the stack is exactly the same as before the function was called.

The key to assume execution control is to overwrite the return address with one that points to shellcode [Alep96, Cowa00, Laro01]. Note that the return address is only four bytes (size of the base pointer value) from the beginning of local variable storage. If user input is stored in those local variables, then providing more user input than storage capacity may overwrite the base pointer and the return address with the address of shellcode somewhere in memory. The shellcode may be stored in a number of places such as the local variables, the environmental variables, heap space, etc. When the function returns, the new return address is loaded into the CPU instruction pointer and execution occurs at that location. In order to write beyond the capacity of the local variables, some ill-designed library functions are used such as strcpy(), gets(), and sprintf() are used. These functions do not have any boundary checking mechanisms and as such may write beyond local variable capacity. There are many variations of this exploit, each utilizing different locations for storing the shellcode or different methods to make execution occur at the shellcode.

2.7.1.2 Heap Buffer Overflow Faults. Heap buffer overflow exploits are more complex than stack overflows. The exploit may occur when carefully constructed user input or environmental variables overwrite heap control data. The fictitious control data is used by a memory management algorithm to track allocated and free chunks of memory. The Linux kernel [Free03] uses the Doug Lea [Lea03] memory allocator (malloc). This algorithm places control data between chunks of allocated memory such as the size of the chunk, the previous size of the chunk, whether it is memory mapped, and whether the chunk is in use. When the chunk is freed, the algorithm checks to see if adjacent chunks are free and consolidates them if possible. Free chunks are accessed by a double linked list, with the two pointers stored in the area where the user had kept their data.

By crafting a special buffer [Kaem01, Anon01] that overwrites the 'in use' bit, the size field, and the previous size field, the attacker may create a fake control data structure. When the modified chunk of memory is legitimately freed, the algorithm checks adjacent chunks and notes that the fake free chunk needs to be consolidated. It uses an unlink macro to remove the fake chunk from the list so that it may do so. The unlink macro simply follows and reassigns pointers, but to do so requires specifying the address to overwrite and the address to apply. If the fake chunk data is carefully constructed, this allows an arbitrary location in memory to be overwritten. The common exploit overwrites the global offset table (GOT) for free() with the address for shellcode. This shellcode may reside in the same buffer that overflowed.

2.7.2 Race Condition Faults. A race condition is another type of software/security fault that may be exploited in certain situations. It is defined as "anomalous behavior due to unexpected critical dependence on the relative timing of events" [Howe03]. Common race condition exploits prey on the programmer's assumption that resources or the environment have not changed between events. For example, a program may check the status of a file before opening it for reading. During that brief period between the status check and the opening of the file, that file may have been deleted and replaced by mechanisms outside the program. The program would then open the replaced file instead of the original. The assumption in this case is that the status check and the opening of the file were atomic. Similar exploits involving the creation and access of temporary files commonly exist. For further information on this type of fault exploitation, refer to [Cowa01, Bish96].

2.8 Artificial Immune Systems

Artificial immune systems (AIS) are a useful tool for complex recognition and classification problems. They model the natural immune system in living organisms and have demonstrated such characteristics as "specificity, diversity, memory, adaptability, recognition, and distributed detection" [Dasg99]. AIS has been applied to several problems such as intrusion detection [Will01a, Harm02, Forr96, Hofm98, Gonz02], an immunity based management system for semiconductor production [Fuku98], autonomous mobile robot behavior [Wata98], tool breakage detection in milling operations [Dasg96], and scheduling [Hart99].

Modelled after the biological immune system, antibodies in an AIS are stochastically created to bind to foreign entities called antigens, while at the same time prevent binding or detection of legitimate host objects. Every object is considered to belong to either the self or non-self set. Antibodies cover a portion of non-self space such that any object matched in this space is identified as non-self. Figure 2.1 illustrates the disjoint nature of self and non-self as well as antibody coverage of the non-self space. A process called negative selection produces random valued antibodies and attempts to match them with self data. If a match occurs, that antibody is censored. Figure 2.2 illustrates the process of negative selection. After negative selection, the antibody is matured such that it maximizes the number of antigens it matches while still avoiding any match with self. This process is referred to as affinity maturation and often utilizes some variant of evolutionary computation to expand the matching strength. A matured antibody is compared to features or objects within the monitored system and any match indicates non-self. Figure 2.3 provides a flow chart of this process.

Artificial immune systems are a type of probabilistic detection [Dasg96]. The antibodies need only match an antigen to a given threshold before detection is affirmed [Forr94]. This is called imperfect matching, and while this produces situations where some antibodies match self data, significantly more antibodies match when non-self data is introduced. The difference between the

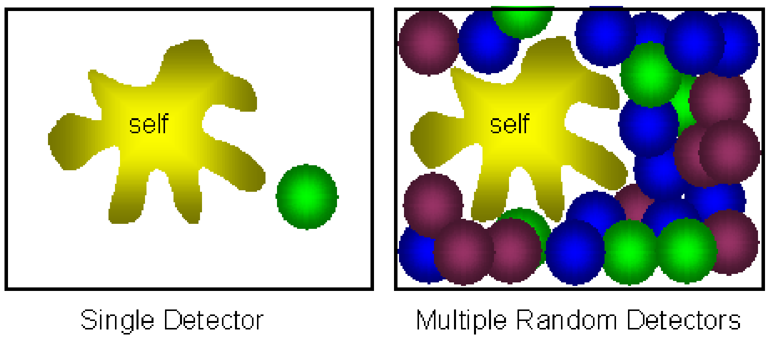

Figure 2.1 Self, Non-Self Detectors [Hof98p]

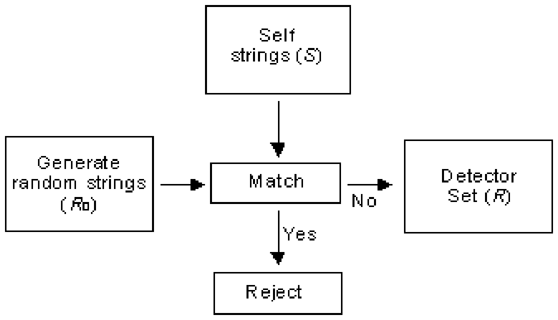

Figure 2.2 Negative Selection [Forr94]

baseline normal matching and anomalous data matching is the criteria used to distinguish non-self from self. In addition, co-stimulation is used to filter out false positives by requiring multiple antibodies to match an object before an it is identified as non-self.

2.9 Evolutionary Computation

Evolutionary computation algorithms are tools that model the natural process of biological evolution for a multitude of purposes including the search for optimality and as an adaptive system in respect to some feature(s) in a particular environment. Some of the key biological methods that are essential to the natural evolutionary process are "reproduction, random variation, competition and selection of contending individuals in a population" [Back00]. Reproduction may be sexual or asexual and result in one or more children with varying degrees of phenotypic traits from either

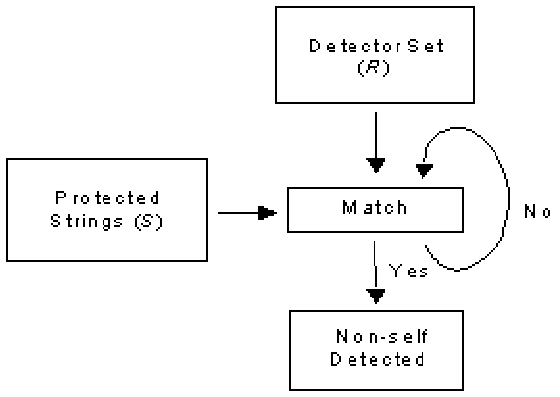

Figure 2.3 Scanning for Antigen [Forr94]

parent. Random variation introduces new genotypic characteristics that were not inherited from the parents, possibly resulting in changes to the phenotype as well. Competition ensures the individuals with the best phenotypic characteristics for a particular environment may dominate other members for a limited resource. Closely related, selection provides partiality for certain members of a population with better phenotypic characteristics to mate and reproduce.

In order to apply evolutionary computation to a problem, the problem domain must be encoded into data structures that the algorithm may manipulate [Back00]. These data structures represent the genotype or genetic composition of one individual in a population. One common type of encoding is a binary string where each bit or groups of bits may be interpreted or mapped into values in the problem domain. These values represent the phenotype and are evaluated based on user-defined criteria. A fitness is assigned to each individual's phenotype indicating how well it fits as a solution in the problem domain.

An evolutionary algorithm also includes a set of operators. These may include reproduction, mutation, selection and reinsertion [Back00]. Because there are many variants of each type of operator, only the most general instantiations of each type are described. Reproduction is the union of two or more individuals' genetic description such that the result inherits portions from each parent. Single point crossover uses two parents and a randomly chosen cross point to swap a portion of one parent with the other, resulting in two children. Mutation is a random variation introduced into an individual. Single bit mutation uses one individual and a mutation rate to probabilistically change each bit. Selection is the operator that selects individuals from a population to be reproduced and/or mutated. Tournament selection uses a tournament size to randomly chose a fixed number of individuals from the population. The best individual from this group is considered selected and the tournament selection repeats this process until the quota of individuals are selected. Finally, reinsertion places some or all of the children created by reproduction and/or mutation into the population of individuals. In doing so, some of the parents may be replaced. Elitist reinsertion replaces the worst parents with the best children.

An evolutionary algorithm repeats the process of selecting parents, performing reproduction and/or mutation, and reinserting some or all of the children into the population. Each iteration is considered a generation and many generations may occur depending on user-defined halting criteria. There is no guarantee that this algorithm produces optimum or even optimal results when halted.

2.10 Summary

This chapter introduces some of the basic concepts associated with intrusion detection as well as examples of systems that have been implemented using various approaches. In addition, it provides background information on topics such as artificial immune systems and software faults that are relevant to the design discussion in Chapter 3 and Chapter 4. Furthermore, it provided a description of three types of intrusions that are used in Chapter 5 as test exploits.

III. High-Level Methodology

The previous chapter examined several earlier approaches to intrusion detection on host, network and hybrid systems. A select few of these specifically utilize anomalous behavior detection strategies. These strategies are based on the idea of distinguishing anomalous behavior from normality. This thesis investigates the applicability of this strategy to a process' system call information in an attempt to demonstrate the value of this concept within this context.

This chapter describes the high-level design decisions and methodology considered while defining the problem and algorithm specifications. The problem class and complexity are addressed in Section 3.1. The fitness landscape is described in Section 3.2. The definition and mapping from problem domain into the algorithmic domain is addressed in Section 3.3 and Section 3.4. Procedure description and system design are outlined in Section 3.5 and Section 3.6. Design details and implementation specific information is described in Chapter 4.

3.1 Problem Class & Complexity

The type of information utilized is collected from system call tracing on a Mandrake 7.1 distribution of Linux kernel 2.2.15, and thus specific to this operating system. This information per system call includes the originating instruction pointer address, the system call function parameters, and the function return value. Although each system call differs in the number and size of parameters, the instruction pointer address remains a fixed length and the return value varies only occasionally. The instruction pointer address ranges from 0x00000000 to 0xFFFFFFFF because system calls must originate within a process' private memory space, requiring 32 bits for representation. The return value is generally of integer word length requiring 32 bits for representation. System calls always provide return values as a mechanism for error notification, with the exception of the exit() system call [Hyde03]. Although the range for error notification does not always use all 32 bits, this size is maintained because the data structure is that size. Thus for all but one system call, the smallest system call function has a minimum of 64 bits. System calls with parameters may include several word length values. For example the system call fstat() takes two parameters: an integer file descriptor and a struct stat_t. The struct stat_t data structure includes seven 32-bit numeric values. Other parameters exist such as character strings, however they are excluded for this thesis effort. Thus fstat() takes a total of eight 32-bit numeric values as parameters. In addition to the instruction pointer address and return value bits, it takes 320 bits to represent this system call. Using these values, it would require a deterministic algorithm to search 2^{64} strings to cover the space for the smallest system calls and 2^{320} for larger system calls. If one were to search the set of 2^{64} strings, one per microsecond, it would take 5.85 x 10^5 years to complete. Similar to Williams' AIS work with packets [Will01a], and Harmer's work with virus detection strings [Harm02], the size of the search space precludes a deterministic search and likewise this problem is of NP complexity.

3.2 Search Landscape

The exact search landscape differs for each system call and is dependent on the both the structure of the process and the input applied to it. For example, the brk() system call changes the amount of memory allocated for the data segment of a process. It takes an address parameter to set the heap pointer and returns the same value if successful. Passing a value of zero returns the current data segment boundary location. Points on the search landscape for this system call for a given set of normal input may occur clustered within a region as memory is incrementally added or removed from the data segment. The conjecture here is that situations where the input influences the number or size of objects on the heap may push this beyond limits implicitly established as normal, however there is no guarantee that this clustering will occur. This clustering phenomena has been observed in the test data collected.

Conversely, memory addresses for system call functions are the same for each program that uses the same shared library because of the common stub image compiled into the binary. Instead of clusters this produces specific points in the search landscape that should not deviate unless the stub image has been overwritten or bypassed. This is apparent from the test data, however this is only one aspect of a system call. Ultimately the search space is dependant on multiple aspects of a system call, and in turn these rely on program and user input. It is conjectured that a program produces similar normal values for normal usage and input, but may produce different values if the user input is extremely out of the ordinary.

3.3 Problem Domain Requirements Specification

The problem may be viewed as a set of features that exhibit either normal or abnormal behavior. If normal behavior is defined as the space of observed samples during normal operation, the problem is then given a particular feature and normal samples (samples are also called measures in other literature) of the behavior of that feature, can detectors be constructed that effectively detect anomalous behavior? There are several assumptions made when addressing this problem. First, anomalous behavior indicates an intrusion. In reality, anomalous behavior may include unobserved measures of normalcy or a benign abnormality. Second, the set of behaviors has a level of stability that allows for distinction between normality and abnormality. In reality, a feature may be too unstable to make a distinction. These two assumptions may not completely and accurately reflect the real world in all cases but are somewhat necessary to scope the problem.

The problem domain consists of a list of tuples, where each tuple is comprised of values corresponding to each feature of a particular system call. This first list is produced from normal execution of a program and is referred to as normal behavior. A second list is produced from execution that is unknown whether it was normal or abnormal. This is the test behavior.

3.4 Algorithm Domain Requirements Specification

The algorithm domain consists of three approaches that may be useful for detection. The first is a direct matching approach where test behavior is matched against normal behavior. If the match occurs, then the test behavior is considered normal. This simple algorithm is useful for detecting any differences between the two. The second algorithm is a relaxed matching approach that matches the same as the direct matching, except that it allows matches that are within a specified amount of the normal values. The third algorithm is an artificial immune system that attempts to match non-self regions to test behavior, where matches are considered abnormal.

The biological immune system has been used as a model in other work [Will01a, Harm02, Forr96, Hofm98] as an anomaly detection tool. An artificial immune system (AIS) utilizes concepts of immunology such as the random generation of antibodies to bind to foreign antigens, negative selection to filter out antibodies that bind to self, and affinity maturation to expand the binding capabilities of mature antibodies. This model may be applied to the problem by defining what self and non-self are and applying the these operators.

In the AIS model, normality measurements are defined as self-strings. Randomly generated detector strings called antibodies represent non-self measurements. Negative selection compares the generated antibodies against the self-strings. If a match is made, then the antibody is discarded. The antibody then goes through a process of affinity maturation where the criteria for binding to antigens is increased, but not at the expense of binding to self. From the problem domain, self is defined as the list of normal behavior, whereas non-self is defined as anything not on the list of normal behavior.

One AIS model [Hofm00] defines a lifecycle process where a mature antibody that has been activated by a detection of an antigen is redefined as a memory cell and remains in the system indefinitely. Mature antibodies that are not activated are eventually removed from the system after a time period to simulate the death of the antibody. This approach has been found useful to address the changing definition of normality, however it is not implemented in this thesis effort.

During the scanning phase, antibodies are matched against further measurements. An antigen is detected when an antibody matches to a given affinity threshold. When multiple antibodies match an antigen to an affinity threshold, costimulation occurs and provides credibility that the behavior is indeed anomalous and not a false positive because several varying detectors found the behavior to be outside of self.

3.5 Procedure

The high-level methodology includes the following procedure. Various "root-level" or "set root level on execution" applications are selected for experimentation. Each application is traced under normal conditions with a tool called strace with appropriate options invoked to include originating instruction pointer address and detailed parameter and result information. This trace output is then parsed into a vector of data structures that maintain the values and valid ranges for each piece of data. A unique data structure is designed for each type of system call because different numbers and sizes of parameters exist for each. The same data structure includes space allocated for offset values to produce ranges. This allows the data structure to be used to represent either a point or a volume. These are used by antibodies to specify non-self space and also used by relaxed matching to specify normal self spaces.

Normal and test traces are compared using the three methods described in Section 3.6.2, Section 3.6.3, and Section 3.6.4. The direct matching method compares normal and test behavior and classifies any difference as an abnormality. The relaxed matching method compares and classifies test behavior as abnormal if it is outside of a tolerable range of the normal values. Finally, the artificial immune matching classifies everything as normal unless the test behavior matches any of the antibodies. The experimentation takes place by taking further traces of the same applications under abnormal and intrusive behavior. By comparing detection results with the known behavior of the application, error rates may be assessed.

3.6 System Design

Several components are necessary to perform the above methodology. These include a strace parser, a negative selection algorithm, an affinity maturation algorithm, and a scanning algorithm. Data storage is implemented as an abstract Function class and extended specific classes, one per system call. In addition, each detection approach is implemented as a subclass of an abstract AnomalyIntrusionDetectionSystem object. This section describes an overview of the implementation. Further details are included in Chapter 4.

3.6.1 Top-Level Perspective. The abstract AnomalyIntrusionDetectionSystem (AnomIDS) object defines the data storage for normal behavior data as an array of java.util.ArrayLists such that each index of the array corresponds to one system call type. This allows a trace of system calls to be categorized because only system calls of the same type are comparable. Duplicate entries are censored from the lists. A second array of java.util.ArrayLists is allocated for test behavior traces and is organized in the same manner. Finally, a parser object is defined to read in trace text produced by strace. Each line read corresponds to one system call printout. Function objects are created to store each system call's information. These objects are stored in appropriate ArrayLists.

3.6.2 Direct Matching Algorithm. The DirectMatchingSystem is extended from AnomIDS and implements an exact matching algorithm. When a questionable system call needs to be identified, the direct matching algorithm scans through the normal data of that system call type and returns a success on the first complete match, otherwise it returns a failure. The purpose of this algorithm is to provide information on the exact difference between two traces. For further information, refer to Section 5.1.2.

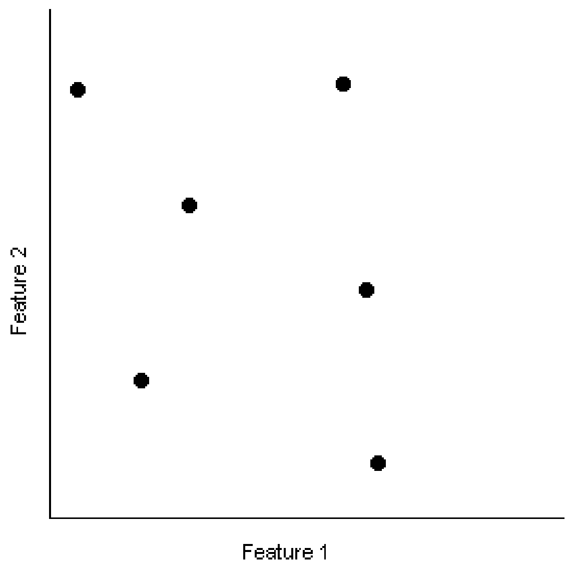

Figure 3.1 Direct Matching

System call information may be described as a point in an n-dimensional space where each feature is represented by an axis. The number of features determines the dimensionality of the space. Figure 3.1 illustrates that a direct match requires exact matching of values for each feature.

3.6.3 Relaxed Matching Algorithm. The RelaxedMatchingSystem is also extended from AnomIDS and implements a relaxed matching algorithm. After normality data has been loaded, each system call object is copied into a separate array of ArrayLists and the values of the object are defined as a range rather than a point of normalcy. Matching now occurs using the ranged normality rather than the exact match. Figure 3.2 illustrates that when a system call needs to be identified, it is considered a match if the values of all the features within that system call are in the specified ranges. The purpose of this algorithm is to test for traces that are in proximity to normal points. This is useful for comparing normal traces with other normal traces. If these two traces do not directly match, it is interesting to know if they are within a certain proximity of the normal points. For details refer to Section 5.1.2.

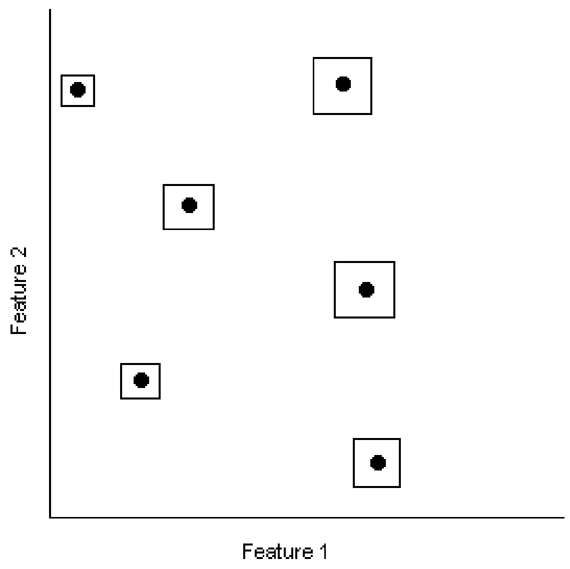

Figure 3.2 Relaxed Matching

3.6.4 Artificial Immune Algorithm. The ArtificialImmuneSystem is another extension from AnomIDS and implements an artificial immune approach similar to that described in Section 2.8. After normality data has been loaded, a specific number of antibodies are produced for each system call type and stored in a separate array of ArrayLists. These antibodies are only placed in the array if they survive the negative selection process described in Section 3.6.4.1. The purpose of the antibodies is to represent a range or a volume of non-self space. Next, the antibodies are matured using evolutionary computation as described in Section 3.6.4.2. This affinity maturation process attempts to expand the range or volume of non-self space that it can match. Finally, when a system call needs to be identified, it is compared to each antibody and considered a match if a complete match of features is made. The number of antibodies that match a system call are totaled and if no matches are made, then the system call is considered normal. Figure 3.3 illustrates the coverage of antibodies around self points.

One important aspect of artificial immunology are the concepts of self and non-self. In the context of this research, the set of self is defined as the traces recorded during normal operation. Normal operation incudes not only the legitimate usage of an application, but also any error

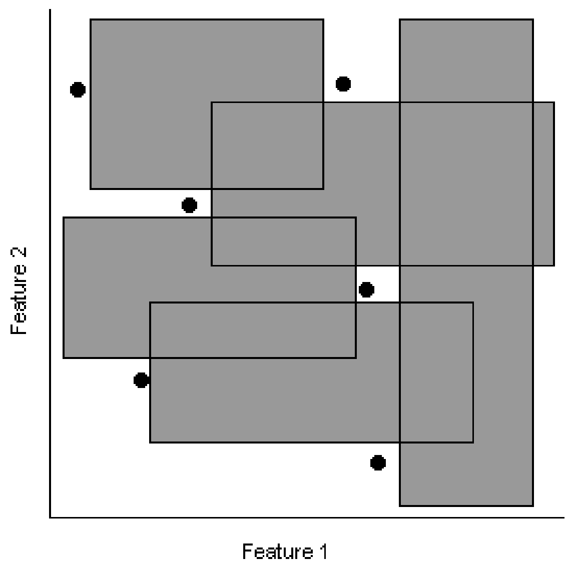

Figure 3.3 AIS Matching

handling. Training data is used to define self. Henceforth, the terms self, normal, normality data and training data are synonymous. Conversely, non-self or abnormal data is defined as everything not defined explicitly as self. A limitation of this approach is that when the self set is not completely identified, antibodies may be produced that match the unidentified portions of self, thus potentially leading to false positives. This is an important aspect of this work because legitimate usage for one user may train the system such that the legitimate usage of another user may produce false positives.

3.6.4.1 Negative Selection Algorithm. The negative selection algorithm described in Section 2.8 provides a means of stochastically generating a number of antibodies and ensuring that none of them match the self data. This algorithm continues to generate and filter antibodies until the specified amount is created. A point is initially generated and assigned a range of plus and minus one in each dimension (for each feature) as the initial volume. Next this volume is tested against all of the normal data. If a complete match occurs, the antibody is discarded. Otherwise it is added to the antibody list. Figure 3.4 illustrates how initial antibodies might be placed in non-self space.

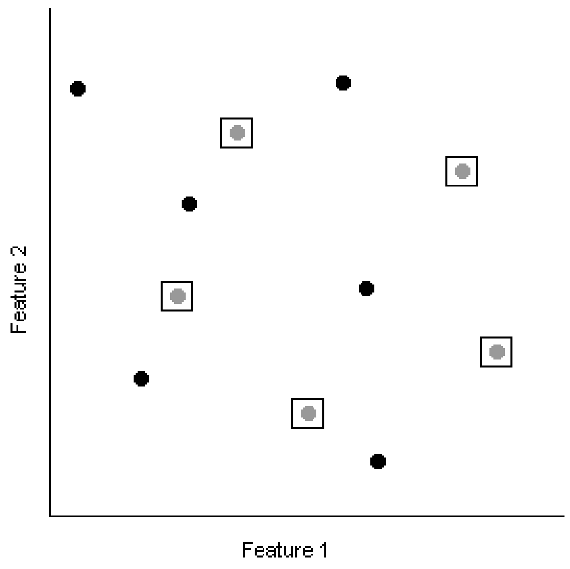

Figure 3.4 Example of Initial Antibodies Produced by Negative Selection

The reasons for selecting a point and small range are twofold. First, it allows the opportunity for antibodies to be placed in crevices close to normal data. Larger initial antibodies may not be usable because a larger area increases the probability that a normal point may be contained within it. Second, it provides an anchor for the antibody to use during affinity maturation in manner that allows it to apply different offsets to find the most fit in non-self space.

3.6.4.2 Affinity Maturation Algorithm. The affinity maturation algorithm provides an avenue for expanding existing antibodies to cover a larger non-self space. Two approaches were considered for this task. The first involved a deterministic growth of each feature of a system call, however this leads to a biased expansion of features. In order to provide a holistic method of expanding the ranges, a stochastic approach was chosen. By allowing ranges of features to expand or contract pseudo-randomly and evaluating each volume produced, the algorithm searches for the best fit with a combination of ranges. This is essentially an optimizing problem that attempts to maximize the volume of a given antibody with an anchored location and a constraint that it does not contain normal points. The static location is necessary because otherwise antibodies may gravitate towards large areas of non-self space and coverage near normal points would lack. By anchoring the location, an antibody competes with different versions of itself rather than competing against all of the other antibodies for the largest volume. Figures 3.5 and 3.6 illustrate how various combinations of ranges are evaluated before selecting the optimal range shown in Figure 3.7.

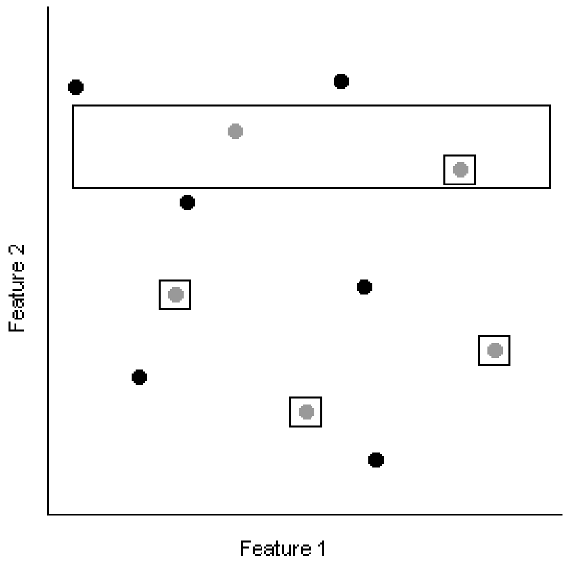

Figure 3.5 Affinity Maturation Example 1

Evolutionary computation was chosen as the mechanism for affinity maturation because of the large number of combinations of the features to expand. The details on the evolutionary algorithm are described in Chapter 4. Each feature has a positive and negative offset value from the fixed location of the antibody. This allows for independent expansion of both directions for any given feature in a system call.

3.7 Summary

This chapter provided a description of the problem domain requirements and the proposed algorithmic solution. It has provided system design and implementation decisions as well as insight

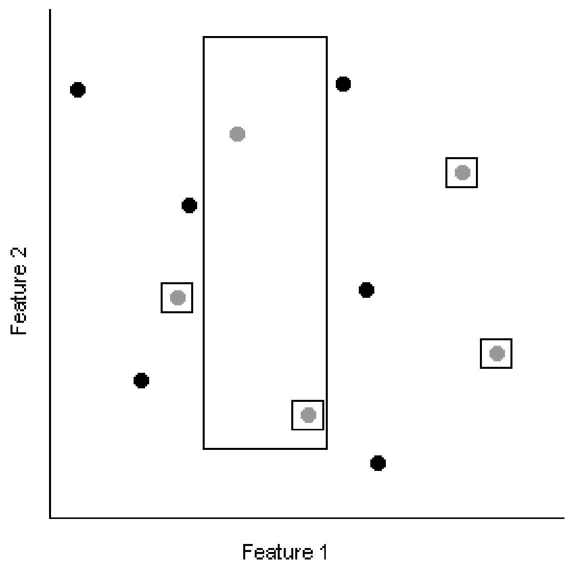

Figure 3.6 Affinity Maturation Example 2

to the types of testing expected to occur in Chapter 5. The problem class and complexity were discussed with respect to the features from the system call information. It was conjectured that the problem is of NP complexity due to the combinatorics of multiple features. This chapter also discussed and illustrated three types of matching used to compare behavior data against normal training data. The next chapter provides a detailed account of implementation and specifies the particular values used for exercising these algorithms.

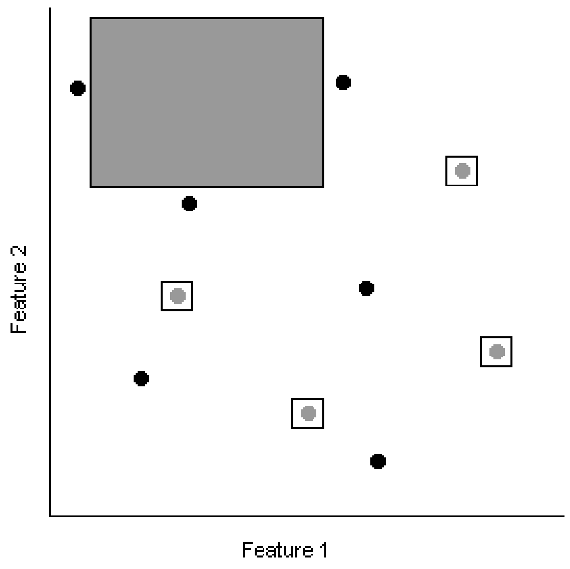

Figure 3.7 Affinity Maturation Example 3

IV. Low-Level Methodology

The previous chapter outlined the high-level aspects of the problem and algorithm domains. This chapter provides more depth as it describes the low-level design decisions and system implementation details considered while implementing a prototype system that demonstrates the artificial immune system concept applied to system call information. A short description of system components is discussed in Section 4.1. Specific information about system call tracing is explained in Section 4.2. Normality, relaxed rule, and antibody rule data structures are described in Sections 4.3 and 4.4. Finally, the negative selection and affinity maturation algorithms are described in Section 4.5 and Section 4.6.

4.1 System Design Components

The demonstrative prototype system is composed of several components. The first is a parsing mechanism for interpreting the output text file from strace. The information from strace is described in more detail in Section 4.2. The parser stores the data from each system call in a data structure unique to each type of system call. The set of each type is contained within an array list effectively categorizing every system call into exactly one group based on type. These array lists are combined into an array such that each index into the array contains the membership of a system call type. The usage of an array list within an array is necessary to effectively organize and efficiently retrieve a particular system call data structure. This is practical because system calls and associated information are compared only to those of the same type. This list is used to generate relaxed matching rules, antibody rules, and for evaluating antibody fitness during affinity maturation. Secondary or multiple normality traces may be included, and any duplicate entries are discarded such that only a set of unique values is retained.

Separate array lists for normal range rules and antibodies are constructed and maintained to be used for comparison to test behavior. Again each system call is grouped by type and indices into each type correspond to the indices of each type in the normality list. Because there exist 197 system call types in the Linux kernel 2.2.15 [Bold03], there are 197 indices for groups of system call information. Relaxed normal rules are generated by setting volume boundaries at an offset from normal points, while antibodies are generated randomly as points and filtered through the process of negative selection, as described in Section 4.5. The remaining antibodies are expanded to match additional non-self points through the process of affinity maturation, as described in Section 4.6. Both relaxed normal rules and mature antibody rules are applied to the test data.

The test data is also stored in a separate list that categorizes the different system call types. As with the normality data, a parser mechanism is required to interpret the data from a strace output file and store it into appropriate function data structures.

The entire system prototype is implemented and compiled in Java 1.3 SDK for several reasons. First, it allows the system to be run on multiple platforms. Even though strace is specific to Linux/BSD/UNIX environments, analysis may be performed offline on remote machines. Java also provides well-defined primitive data structures of fixed bit length. This is important because specific underlying kernel data structures may be unique to different systems and fixed bit length allows for a large data structure with modifiable ranges to be used. Finally, Java has a large and developed suite of support APIs for a variety of functionality including graphical user interface, string handling, arbitrarily large integers and decimals, and sorting. Future versions may be designed to utilize strace directly as a back end shared library specific to a platform while the front end is the Java analysis tool.

4.2 System Call Information

The system call information captured by strace is formatted in a specific manner commensurate with the command line options specified. For this thesis, strace version 4.2-1mdk was used on the command line in the following manner: “strace -i -f -v -p pid -o traceX.txt” or “strace -i

-f -v -o traceX.txt process". The only difference between the two is that the former specifies a process identifier number of an existing process to trace as indicated by the -p pid, whereas the latter instantiates a new process. The -i option specifies that the instruction pointer at the time of the system call is recorded. The -f option specifies that child processes spawned by the fork system call are also traced. The -v option prints the "unabbreviated versions of environment, stat, termios, etc. calls. These structures are common in calls and so the default behavior displays a reasonable subset of structure members" [OSDN03]. This effectively de-references data structure pointers and displays their values. Finally, the -o option specifies that output is written to an ASCII file. Other options for strace do exist but were not utilized. The -c option counts time, calls, and errors for each system call and prints a summary at the end of the output file. This option was not used because the scope of this research is limited to the data directly affiliated with individual system calls. Another is the -F option that attempts to follow vfork system calls. This option was not used because strace does not guarantee that every vfork will be traced, only that it attempts to do so. Such lack of reliability made this an undesirable option. Finally, the -t, -tt, -ttt, and -T options provided timing information about the system calls. This may be useful as another feature to measure normalcy, but is not used in this work because it was not considered such a solid value as those that were stored in the machine.

The output file is formatted in the following style. Refer to Fig. 4.1 for an example trace. Each system call is listed on a separate line with four pieces of information. The first is the instruction pointer enclosed in brackets in hexadecimal format. The second is the name of the system call in lower case ASCII. The third is a variable length parameter list. Finally, the result is recorded as an integer or hexadecimal value. The parameter list includes comma-delimited values corresponding to each parameter and enclosed by parenthesis. Pointers to data structures are de-referenced and displayed in terms of primitives. These data structures are enclosed in braces and are also comma delimited. A numeric value may be denoted by either integer, hexadecimal, or static name if it exists. Strings and file names are enclosed by quotation marks. Occasionally strace is not able to determine values and hence places a question mark to indicate this situation. This occurs in two situations. First, when the initial execve() system call is made, the instruction pointer is not retrieved. Second, when the exit() system call is used, a return value does not exist and strace writes the return as a question mark.

```
[4000e6d4] brk(0) = 0x8049704
[4000dd44] open("/etc/ld.so.cache", O_RDONLY) = 4
[4000e824] mprotect(0x400f7000, 32444, PROT_NONE) = 0
[400b4604] ioctl(1, TCGETS, {c_iflags=0x500, c_oflags=0x5, c_cflags=0xbf}) = 0
[4009a737] getpid() = 9404
```

Figure 4.1 Example Trace

4.3 Trace Data Structure Design

System call information data structures are used to store data interpreted from strace output files. An abstract class named Function defines the commonality between specific system call data structures such as the instruction pointer value, the string specifying the name of a system call, and the function result value. Sub-classes such as those in Figure 4.2 define the details regarding the parameters list. A sub-class exists for each system call type and is named after the system call it represents. This design is necessary for two reasons. First, the abstraction of a super class allows top-level manipulation of function data structures without regard to specific type. Second, it allows for each system call type to be customized based on the extent of the parameters list. For example, the parsing mechanism for instruction pointer, name, and result are defined in Function, whereas the parsing mechanism for the parameter list is defined in the sub-class, because it is specific to that sub-class.

Data structures referenced by multiple system calls via pointers are defined as separate objects. For example, Fstat, Lstat, and Stat use a data structure named stat_t to store information

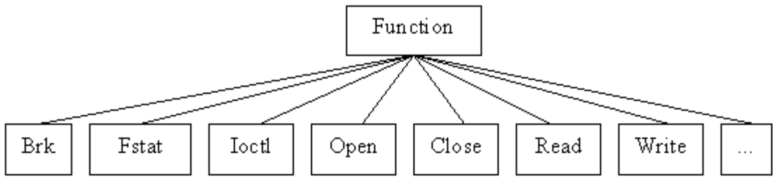

Figure 4.2 Hierarchy of Sub-Classes

regarding file status. This structure is implemented as a separate class for the purpose of code reuse.

4.4 Relaxed & Antibody Rule Data Structure Design

The relaxed and antibody rule data structures are the same data structure as the trace data structure, but utilized in a different manner. This data structure stores information regarding two sets of boundaries. The first is the minimum and maximum value that each feature may range. These values define the absolute size of the feature's native data structure. This approach also allows unsigned values to exist because Java does not support unsigned values. For example, an unsigned 32-bit integer may be defined in Java as a 64-bit long with boundaries of 0 through 4,294,967,295. Native Linux kernel data structure values are defined within the kernel documentation [Tama03]. The second set of boundaries defines the actual size of the rule in terms of lower and upper values. These values are relative offsets from the location point. In relaxed normal rules, the offset sizes are symmetrical because the intent is to include any value within the relaxed range from the normal point. In antibody rules, size is not symmetrical and either value may be independently modified. This allows the affinity maturation process to change size to conform to non-self space. Details on this process are included in Section 4.6.

4.5 Negative Selection

The purpose of the negative selection algorithm is to filter out antibodies that match self as they are randomly generated, resulting in a set of antibodies that do not match self. An explanation of the process is described in Section 3.6.4.1. This implementation creates a fixed number of antibodies per system call type.

One topic not addressed in this implementation is proximity of antibodies being randomly generated. If multiple antibodies are sufficiently close to each other, the affinity maturation process may expand them such that two or more may overlap. Overlapping antibodies do serve some purpose for costimulation, however may be detrimental on the large scale because an overlapping antibody may be more useful covering an unoccupied space of non-self. Techniques similar to niching in evolutionary computation may be useful to prevent congregation of multiple points in a local area and spread them out. Future work may address this issue.

4.6 Affinity Maturation

The purpose of the affinity maturation algorithm is to expand the values that match non-self while avoiding those that impinge on self. The result of affinity maturation is an antibody that matches a range of non-self values, thus covering an area of non-self behavior rather than only one specific point. An overview of the process is described in Section 3.6.4.2. Affinity maturation was performed with an evolutionary algorithm (EA) to grow these antibodies. It encodes each low and high value for each feature as a concatenated string of bits. The algorithm manipulates the encoding by using a single bit mutation operator with tournament selection and elitist reinsertion. Recombination was not used because each group of 32 bits in the bit string represents a positive integer. Crossing over the low order bits did not seem particularly beneficial as compared to an encoding where each bit provides phenotypic information by itself. This algorithm is limited to a finite number of 1000 generations because the evaluation function manipulates arbitrary large number and is expensive to perform. In addition, the results from parameter testing seem to suggest that this is an appropriate value. As shown in Table 5.1, fewer generations with more antibodies seemed to provide similar results, however it was decided to minimize the time required to run each experiment. Other factors were also considered in conjunction with the number of antibodies. The following sections describe these factors as well as the EA data structure and operators implemented.

4.6.1 Population & Individual Chromosomal Encoding. The population consists of a number of offset solutions encoded as individual chromosomes. Each chromosome is encoded as a concatenation of lower and upper offsets in that order for each data field in the system call information. All values are converted to a binary resulting in a binary string. Because each type of system call may have a different number of parameters, the encoding length in terms of total number of bits are different depending on type. This requires a population size and number of generations that are commensurate with the length of string. A population size of 50 chromosomes was chosen based on results from Table 5.1. These tests used most of the system calls, especially those that use longer encodings. Other values for population size were tested, but did not seem to produce better antibodies.

4.6.2 Evaluation Operator. The purpose of the evaluation function is to quantify the fitness of each individual chromosome such that better individuals are assigned higher fitness values and worse individuals are assigned lower values. The measure of fitness is defined as the size of the volume created by the ranges of information fields for each system call type. The size of each range is multiplied together and the resulting value represents the volume of the non-self space covered. A larger volume is preferred as long as it does not include any self points. Any antibody that impinges on self is assigned a fitness of zero. The assignment of zero as a fitness for violating self is used in conjunction with a fitness reinsertion operator that allows only the globally best individuals from either parents or children to be carried into the next generation. Thus those individuals that violate self are quickly removed but the parents that created them may still remain, offering the opportunity for similar and perhaps better children in the future. One option considered was to assign a penalty to individuals that impinged on self and attempt to repair that individual. This was thought too computationally expensive and more complex than necessary.

Because of the extremely large size of volumes produced by multiplying several unsigned 32-bit values, a data structure called java.math.BigInteger is used to provide storage for arbitrarily large sized values. This is necessary because otherwise the largest primitive, the 64-bit long data structure, would overflow and produce false data.

4.6.3 Selection Operator. The purpose of the selection operator is to choose a segment of the population for mutation. Generally it is important to ensure the best individuals are bred while also providing a relatively smaller opportunity for less desirable individuals to also be bred, thus maintaining some level of diversity. The operator chosen was a binary tournament selection. This operator was chosen because it is relatively efficient and easy to implement. Roulette wheel and truncation selection were considered but each has undesirable characteristics. Roulette wheel and the similar stochastic universal sampling may have been excellent candidates had they been not so complicated to implement with extremely large fitness values, in turn requiring extremely large decimal values (java.math.BigDecimal) to store proportions. On the other hand, truncation selection may have been easy to implement, but it tends to prevent less desirable individuals from breeding. Tournament selection seemed to be a good compromise between ease of implementation, efficiency and effectiveness.

4.6.4 Mutation Operator. The purpose of the mutation operator is to provide the random variation necessary to search a large space. In genetic algorithms it is commonly used to break out of local convergence produced by recombination, but in this case it is used to pseudo-randomly flip the bits of the encoding to increase the offsets without bias. With every generation, it takes a few of the best solutions and attempts to make better ones by changing them moderately. The mutation rate for this implementation was 30%. It is set to this rate to grow antibodies quickly even with fewer generations. Ideally, this rate should taper off after awhile to allow fine tuning of good solutions. This was not done because it is not necessarily good to have the antibodies expanded too close to self points because this area may also be self and false positives may increase. This detection capability near self points was compromised in order to prevent false positives. In a layered defense, false negatives are easier to accommodate than false positives because the other layers may catch the intrusive behavior.

4.7 Summary

This chapter described the specific low-level design and implementation details of a prototype anomaly detection system using system call information. These details included information on the data structures used and the algorithms implemented. These data structures were designed to store attributes from the system call information. Particular attention was also placed on the tracing tool and justification of the features used from the trace. The next chapter describes the design of the experiments, results, and analysis.

V. Test Results & Analysis

The previous chapter described the low-level details of the problem and algorithm domains as well as system components, data structure design and operator design. This chapter provides the design of the test plan, results of experimentation, and analysis of those results. The purpose of the testing is to demonstrate that system call information may reflect attempted and successful intrusions. Furthermore, three detection methods are applied to identify differences between normality and test behavior. The design of experiments is outlined in Section 5.1. Information on the test cases is described in Section 5.2. The results from experimentation are described in Section 5.3. Finally, an overall analysis of these results is explained in Section 5.4.

5.1 Design of Experiments

Some initial testing was required to determine what values were appropriate for negative selection and affinity maturation. The negative selection algorithm needed a number of antibodies specified, while the affinity maturation algorithm needed the number of generations and size of population specified. It was assumed that there may be some tradeoff between the number of antibodies produced and the maturation strength (in terms of number of generations and size of population). The metrics used to measure the effectiveness of these parameters are the number of abnormal system calls detected out of 106 abnormal calls possible. Abnormal system calls not detected are classified as false negatives, whereas system calls misidentified as abnormal are considered false positives.

After comparing the results of 100, 500, 1000, 2000, and 5000 antibodies per system call type, the difference between them was found to be not significant considering the associated variances. One hundred antibodies were chosen because the time required to generate and mature them was relatively the shortest. Table 5.1 compares the different combinations of values. This testing of parameters was run three times to provide some indication a variance.

Number of Antibodies	**Number of Generations**	**Population Size**	**Number of Abnormal**	**False Positives**	**False Negatives**	**Duration (Hours)**
100	1000	50	39.3 (1.3)	0	66.7	0.5
500	1000	100	41.7 (1.3)	0	64.3	3.0
1000	500	50	42.3 (1.3)	0	63.7	2.5
2000	100	25	38.7 (4.3)	0	67.3	3
5000	50	10	42.3 (16.3)	0	63.7	5.5

Table 5.1 Parameter Correlation to Detection Effectiveness (Testing Localecho w/Root Shell)

The design of experiments also includes specifying an experiment's control and variable factors. The controlled factors in this experiment include the system type and implementation. The system used for performing these experiments is a 1GHz x86 compatible AMD architecture running Windows 2000 Professional with 512MB of memory. The system used for producing the traces is a 700MHz Pentium III running Linux Mandrake 7.1 with kernel version 2.2.15 and utilizing 512MB of memory. The implementation of the prototype analyzer is described in Chapter 3 and Chapter 4, whereas the descriptions of the test programs used to produce the traces are in Section 5.2. The diversity of the test programs represent the variable factor in these experiments.

5.1.1 Influential Factors. Even through a computer system is a deterministic machine, there are influencing factors that must be considered while running these experiments. One such factor is the distribution and period of pseudo-random values used within the negative selection and affinity maturation functions. This is important because a non-uniformly distributed generator may produce values within a localized area, a point that is critical for negative selection because the generator sets the location of antibodies in the feature space. A large period is also important because taking a number of samples more than the period produces repeated values. The java.Random package was used because it produces uniformly distributed values over several primitive data types (long, int, boolean) with "approximately equal probability" [Sun03] of selecting any particular value within the primitive's range. It also provides a period of 2^{32}, sufficient for this thesis effort. The top-level generator was seeded with the computer system's time in milliseconds and this produced the seeds for separate generators in each of the sub-functions. Two separate seed generators were used: one in the negative selection function and the other in the affinity maturation function. In addition, the seed generator was re-seeded with the system's time every one hundred samples to address any issues with sequence repetitiveness. The Mersenne Twister [Mats02] was also considered, however it did not seem to offer any observably better performance and required some conditioning to cast its signed integers into other primitives.

Another influential factor is the selection of programs and input tested. Each program uses different types of input that may produce a range of system call information. Some are highly consistent regardless of the input, while others may produce significant differences. These experiments use several programs that include both stable and input dependent types. Localecho, TCPecho, Rcond, and Ntbo are relatively stable programs generally regardless of the input (other than exploits). Traceroute and FTPclient are programs that are more input dependent. The input data was also chosen to reflect varying situations when applicable. This input may cause a program to execute different paths or possibly trigger errors. For example, a program susceptible to buffer overflow exploits was tested with three types of data. The first input datum was small enough to fit within the buffer, whereas the second filled the buffer exactly, and finally the third exceeded the capacity of the buffer producing an error. Error handling is considered normal. System call information is produced that reflect normality by tracing these three cases. Of course, when sampling a relatively small subset of all possible behavior, there may be instances where some normal behavior is not observed.

The last influential factor is the selection of system calls to use for tracing. The Linux kernel 2.2.15 has 197 system call entries [Bold03]. A selection of 33 system calls was made based on the common usage by the selected programs. Not every system call used by the test programs is included, but the majority are represented. When the strace parser does not recognize a system call, it is not included into the set of normal or behavior data used by the matching functions. Thus it is neither counted as normal, abnormal, nor foreign. Section 5.1.2 provides a description of matching categories. It is possible that distinguishing information is contained in those system calls that were not included. Chapter 6 describes future work to include these. The system calls used for this thesis investigation are listed in Table 5.2.

setup	exit	fork	read	write	open	close
execve	time	getpid	setuid	getuid	dup	brk
setgid	getgid	geteuid	getegid	ioctl	fcntl	setpgid
dup2	getppid	getpgrp	old_mmap	munmap	fstat	uname
mprotect	getpgid	rtsigact	rtsigproc	fstat64		

Table 5.2 System Calls Sampled

Each system call from the trace has a number of features such as the originating instruction pointer, parameters, and results. The parameters of each system call may include numeric values or strings. Only features that have numeric values are used as matching criteria, with the exception of pointer addresses. Pointer addresses are de-referenced by strace and the values of the data structure are listed and matched only if they are also numeric values. The reason strings could not be used as matching criteria is because of the difficulty in creating antibodies for them. Character strings could be converted to a binary string such that any antibodies generated do not match that string, but this does not work as naturally as numeric values.

5.1.2 Measures of Performance.

Performance is defined in terms of effectiveness of direct matching, relaxed matching, and antibody matching. The goal of this research is to demonstrate the effectiveness of the algorithms on system call information without regard to efficiency. Effectiveness is measured by the number of false positives and false negatives produced by each approach.

The direct matching approach measures the exact number of differences between two sets of system call data. When comparing normal traces with other normal traces, it provides information on how many system calls have changed over different input, or in other words the volatility of system calls over normal input. When comparing attempted or successful intrusion traces against normal traces, it again provides information on how many system calls have changed, but measures the volatility of system calls over intrusive input. This information is used as a baseline for comparing the AIS approach in terms of false positives and false negatives.

The relaxed matching approach performs exactly as the direct matching except that it identifies a system call as normal if it is within a plus or minus range of the normal value. This effectively creates a "volume" around a normal point that is considered normal as well. This is useful because it reveals situations where tested system calls are close to a normal point, but do not match it exactly. In these situations, the direct matching approach considers it abnormal even though it may be legitimate. Direct matching is always used as the basis to calculate false positives and false negatives, but the relaxed matching information provides context for explaining these values. The relaxed approach uses an offset of plus or minus 25 from a normal point. This value was chosen because it offered a reasonable leeway for the number of bytes read using the read() system call or written using the write() system call.

The artificial immune system matching approach measures the number of system calls that are considered abnormal. A system call is considered abnormal if one or more antibodies match it completely. The total number of antibodies that bind are provided as additional information since costimulation is not used in this thesis effort. As with the relaxed matching approach, the results are compared to direct matching to determine false positives and false negatives.

A false positive occurs when the matching approach incorrectly identifies a system call and associated information as abnormal. The number of false positives is determined by comparing the number of actual deviations against the number produced by the approach in question. During matching, system calls are designated as either normal or abnormal. The resulting count of each type is used to gauge overall approach performance. The results from direct matching are considered the baseline for comparison. These results may be validated by comparing the trace files manually. Conversely, a false negative occurs when the detection approach incorrectly identifies behavior as normal. Similarly, the number of false negatives is determined by comparing the number of actual normal system calls and associated information against the normality count from the tested approach.

Foreign system calls are also measured during scanning. A foreign system call is one that was sampled, but not found in the normality information. They are identified prior to any matching method and counted separately because no comparisons of system call information can be made to something that does not exist from the training data. This information is added to the normality/abnormality count based on the approach. The direct matching and relaxed matching approaches consider foreign system calls to be abnormal events, while the artificial immune approach considers them to be normal because they can not be bound to any antibody.

5.2 Experimental Data

The data used for these experiments are generated from traces of several programs during normal, attempted intrusion, and exploited intrusion conditions. These programs were carefully selected or constructed to provide meaningful data about the exploits and the effect they have on processes. To this extent, the programs were simple in the sense that each program only performs one specific task without command line options or auxiliary functionality. The idea is to first test with pedagogical programs and then test with non-trivial programs. The goal of each exploit is to perform privileged activity that otherwise would not be authorized. The source code for the constructed programs is provided in Appendix A.

5.2.1 Stack Overflow Exploit. A buffer overflow occurs when more data is written than capacity has been allocated. A detailed description of overflows is in Section 2.7.1. A stack overflow occurs when more data is written to the stack than space allocated. Three pedagogical programs were designed to demonstrate these exploits.

The first program is called Localecho. It takes a string argument, copies it to a 100-byte buffer, and prints it back. No boundary checking is performed as it utilizes the strcpy() function, such that the program overwrites the stack if the input is sufficiently long. The purpose of this program is to demonstrate a buffer overflow on a pedagogical example. Various shellcodes were used including the standard root shell, changing permissions on the shadow password file, and an XOR encrypted root shell. The encrypted root shell was chosen because it demonstrates that these exploits may be identified even where signature-based intrusion detection may fail. The various shellcodes were used because the intrusion detection approach needs to be tested under different exploit conditions.

The second program is called TCPechod. It is a daemon service that listens and accepts connections to client TCPecho programs. Similar to Localecho, it receives data, copies it to a 100-byte buffer, and echoes it back. The purpose of this program is to demonstrate buffer overflows on a network service. This example is more complex than the previous and may exhibit a larger set of normal/abnormal behavior.

The third program is called Ntbo, and represents a non-terminated adjacent storage buffer overflow. It is a simple program that may represent a routine commonly found within a larger program. It takes two input strings, possibly representing such things as username and password, or hostname and port number. These two strings are copied into a buffer using a safe copy function called strncpy(). This function limits the copying to a specified amount and is used to prevent buffer overflows. Later in the program, the sprintf() function is used to copy data from one of the input buffers to another without explicitly limiting the amount copied. The programmer assumes that because the input data was limited to a given length, an overflow cannot occur. Unfortunately, when a user supplies an input that exceeds the size strncpy() allows, this function does not terminate the buffer with a null character and sprintf() copies not only the source buffer, but also adjacent memory. Essentially the shellcode is broken into two pieces and provided as input that eventually is placed adjacently together in memory. The destination buffer may overflow allowing the exploit code to be executed. The purpose of this program is to demonstrate a variant of the classic buffer overflow exploit.

5.2.2 Heap Overflow Exploit. Another type of buffer overflow involves overwriting control data on the heap rather than the stack. The heap is memory that may be dynamically requested or returned during a program. To manage this memory, an algorithm is used to allocate and deallocate blocks of memory. The Linux kernel uses the Doug Lea malloc algorithm [Lea03, Free03]. This algorithm places control data between blocks of allocated heap memory. The control data may be overwritten when a program writes beyond the allocated space. There are variations of this overflow, depending on the particular target program. Traceroute 1.4a5 was chosen as a test case because it is included in the Mandrake distribution, making it widely available, and because of the ease of exploiting a heap overflow using the command line user arguments. This vulnerability [MITR00] exploits a software fault that performs improper management of dynamic memory. The fault involves the allocation of only one block of memory for IP addresses when using multiple -g options. The -g option allows a user to specify particular gateways to use by IP address. During process execution, the program attempts to free a block of memory for each -g option specified, even though only one was allocated for all of them. This produces a segmentation fault. The exploit involves writing fictitious control data that simulates a second block. By writing past the original buffer, the attacker may modify the size data of the allocated block such that free() uses the fake control data instead of the legitimate control data. This fictitious data indicates that it is already free and needs to be consolidated with adjacent free memory. The consolidation process involves the use of a macro called unlink, and it is used on the fictitious control data to overwrite an entry into the global offset table (GOT) with an address for shellcode. Section 2.7.1.2 provides further explanation on this exploit. The purpose of including this exploit is to expand the types of exploits beyond traditional stack overflows and provide test data on heap overflow exploits.

5.2.3 File Binding Race Condition Exploit. A file binding race condition occurs when the status checking and file binding routines are not atomically associated. The file in question may have been moved, recreated, deleted, or modified in the short amount of time between the status checking and file binding. A pedagogical example called Rcond was designed to demonstrate this vulnerability. This program takes a file name and message and writes the message to the file after it has checked that the file exists, that the file is regular (not a symbolic link), and that the user has appropriate permissions to modify the file. An artificial pause was added to allow an easy opportunity to switch the requested file with a symbolic link to a privileged file. This pause does not create the vulnerability, but rather exposes it for easier exploitation.

5.3 Experimental Results & Analysis

The results of the experiments are tabulated as raw counts of normal, abnormal and foreign system calls. These categories are interpreted depending on the approach used. Direct and relaxed matching approaches identify normal as any system call that matches the normal data or normal range of data. Abnormal system calls are those that do not match the normal data, including foreign ones. Conversely, AIS matching approaches identify abnormal behavior by the system calls that match the antibodies. Any system call not matched by any of the antibodies are categorized as normal, including foreign system calls. False positives for a particular approach are determined by the type of data used. If the trace data is normal, then any abnormal system call is considered a false positive. If the trace data is attempted or exploited, then false positives are calculated by subtracting the direct matching abnormal count from that approach's abnormal count. False negatives are calculated by subtracting the direct matching normal count from a particular approach's normal count.

There are three categories tested: normal, attempted intrusion, and exploited (successful) intrusion. The normal category includes various amounts or types of input to generate a diverse sampling. Attempted intrusions use various input that is intended to exploit, but does not. Exploited intrusions use input that does exploit the vulnerable program. In addition, each program and input combination is repeated three times to provide some level of confidence in the results. In simple programs the difference between runs with the same input is generally nonexistent, however in complex programs there may be unanticipated diversions.

5.3.1 Localecho Stack Overflow with Root Shell. The first experiment exploits a stack overflow vulnerability in the Localecho program using root shell shellcode. This shellcode was obtained from [Alep96]. The first set of test cases establishes normal traces on such input as the first case with 11 characters (bytes) in length, the second with 89 characters in length, and the third with 199 characters in length. The input of the third case exceeds the 100 character capacity in the program and produces a segmentation fault (a buffer overflow occurs, but not an exploit).

	Direct	**Relaxed**	**AIS**
Abnormal	1 (0)	0.3 (0.3)	0 (0)
% Abnormal	4.0 (7.1x10^{-6})	1.2 (4.6x10^{-4})	0 (0)
False Positive	1 (0)	0.3 (0.3)	0 (0)
False Negative	0 (0)	0 (0)	0 (0)

Table 5.3 Localecho Normal Results Across Different Input (Variance over Three Samples, One for each Input Group)

The baseline is established by comparing the normal traces with each other using direct matching. Nine traces were recorded such that the first group three traces recorded system call information using 11 bytes of input, the second group of three traces used 89 bytes, and the third group used 199 bytes. The results comparing the same input size showed no abnormalities. Next comparisons between groups were analyzed. A trace from group one and group two were compared, then a trace from group one and three were compared, and finally a trace from group two and three were compared. Even across the varying sizes of input, the normal count remained consistently close to the total number of system calls. Only one deviation was detected from the write() system call because it specified the number of bytes to write to the terminal. Interestingly, the third case provided only 24 system calls as compared to 27 from the other two. This was due to a segmentation fault that occurred because of the large input size. The AIS matching approach identified zero abnormalities and no false positives between any of the normal traces.

An attempted intrusion for this program is defined as an overflow of the buffer without exploit success. The intrusion may have failed because of incorrectly guessed return address. In this category, the direct and relaxed matching approaches detect one abnormal system call out of 25, again because of the number of bytes written by the write() system call. The AIS matching detected zero abnormalities. It is interesting to note that attempted intrusion for this program created similar results as the error overflow in normal traces while using direct matching.

	Direct	**Relaxed**	**AIS**
Abnormal	106 (0)	106 (0)	39 (4)
% Abnormal	82 (0)	82 (0)	30 (2.4x10^{-4})
False Positive	n/a	0 (0)	0 (0)
False Negative	n/a	0 (0)	67 (4)

Table 5.4 Localecho w/Root Shell, Exploit Results (Variance over Six Samples)

Finally, a successful intrusion is defined as one that produces a root privileged shell. In this category, the direct and relaxed matching approaches detect 106 abnormal system calls out of 130 total, with 14 of those being foreign. The remaining 92 are system calls that have information different from the normality data. The AIS approach detected an average of 39 abnormal system calls using an average 3032.7 antibodies. This resulted in 67 false negatives and zero false positives.

The relatively large number of total system calls from test behavior compared to the total number from normal traces may also be a useful feature as a flag for intrusive behavior. By manually reviewing the exploited program traces, it is obvious that most of the abnormal system calls are from the shellcode being executed. Unfortunately this feature may not always exist for two reasons. First, not all shellcode may produce a large number of system calls. In this case it did because the shellcode invoked "/bin/sh" (another program) to create the root shell. Second, server type applications do not necessarily have a consistent number of system calls because they may be dependent on the number of connections received or how long the service was operating.

For these reasons a percentage of abnormal out of total system calls is used for comparing normal to test behavior.

It is obvious when comparing Table 5.3 and Table 5.4 that there is a significant difference between normal behavior and exploited behavior. The AIS approach does a good job at keeping the number of false positives minimized while retaining the ability to identify some abnormalities in the exploited behavior. The number of false negatives is unimpressive, and the reasoning for these high numbers is conjectured that there are either not enough antibodies or insufficient maturation. The relaxed method and the direct method performed about the same except that the relaxed approach had a slightly lower number of false positives.

5.3.2 Localecho Stack Overflow with Xor Root Shell. The same Localecho program was used with a different shellcode to determine if the type of shellcode used affects the system call information as compared to the previous case. Here the program is applied to a root shell shellcode that has been partially encrypted with xor to prevent signature-based intrusion detection systems from detecting the string "/bin/sh". This shellcode was acquired from [Shel02]. Attempted and exploited traces were compared to the baseline normal data from Section 5.3.1 because the program and normal input have not changed.

	Direct	**Relaxed**	**AIS**
Abnormal	111.7 (2.3)	111.7 (2.3)	37.3 (24.3)
% Abnormal	82.9 ($3.8\text{x}10^{-6}$)	82.9 ($3.8\text{x}10^{-6}$)	27.7 ($1.2\text{x}10^{-3}$)
False Positive	n/a	0 (0)	0 (0)
False Negative	n/a	0 (0)	74.3 (14.3)

Table 5.5 Localecho w/XOR Root Shell, Exploit Results (Variance over Three Samples)

In the attempted intrusion category, all three approaches detected zero abnormal system calls. However in the exploited intrusion category, the direct and relaxed matching approaches detected 23 normal and an average of 111.7 abnormal system calls, where 15 were foreign. The AIS approach detected an average of 37.3 abnormal system calls using an average of 3052 antibodies. This resulted in an average of 74 false negatives and zero false positives.

The comparison between Table 5.3 and Table 5.5 illustrates that there is again a significant difference between normal behavior and exploited behavior. The three approaches produce similar results as the previous section.

5.3.3 Localecho Stack Overflow with Chmod Shadow File. The last experiment with the Localecho program uses a command of chmod to change the permissions of the shadow password file to allow all users to read and write to it. This differs from previous overflows because there is no "/bin/sh" shell used. Instead, this exploit calls "/bin/chmod" and may exhibit a shorter trace than the previous shellcodes. This shellcode was acquired from [Shel02]. Attempted and exploited traces were compared to the baseline normal data from Section 5.3.1 because the program and normal input have not changed.

	Direct	**Relaxed**	**AIS**
Abnormal	1.3 (0.3)	1.3 (0.3)	1.3 (0.3)
% Abnormal	5.3 ($5.3\text{x}10^{-4}$)	5.3 ($5.3\text{x}10^{-4}$)	5.3 ($5.3\text{x}10^{-4}$)
False Positive	n/a	0 (0)	0 (0)
False Negative	n/a	0 (0)	0 (0)

Table 5.6 Localecho w/Chmod, Exploit Results (Variance over Three Samples)

In the attempted intrusion category, all three approaches detected zero abnormal system calls. However in the exploited intrusion category, the direct and relaxed matching approaches detected 23 normal and an average of 1.3 abnormal system calls. No foreign system calls were detected. The AIS approach detected an average of 1.3 abnormal system calls using an average 124.7 antibodies. This resulted in zero false negatives and zero false positives.

Note that the chmod() system call was invoked but not detected because it was not one of the sampled system calls. If it had been, it should be classified as foreign because it was not any part of the normality data.

The comparison between Table 5.3 and Table 5.6 suggests that the system call information was not effective in this case. Although the AIS approach demonstrates a change from zero to 1.3 abnormalities, this change is probably not significant enough to alert on by itself. The other two methods also demonstrated a slight change, but not significant enough. Perhaps if more system calls were sampled, then the intrusion may have been more obvious.

5.3.4 TCPechod Stack Overflow with Root Shell. The purpose of the TCPechod program was to represent a more complex version of program behavior because it accepts network connections and spawns child processes to handle the echo task. Traces from the child processes are included in the experiment. The shellcode used was obtained from [Alep96]. The normal set of test cases establishes normality on such input as the first case with 11 characters (bytes) in length, the second with 82 characters in length, and the third with 199 characters in length. As with Localecho, the input of the third case exceeds the 100 character capacity in the program and produces a segmentation fault (a buffer overflow occurs, but not an exploit).

	Direct	**Relaxed**	**AIS**
Abnormal	2 (0)	1.3 (0.3)	0 (0)
% Abnormal	4.4 (2.8x10^{-6})	2.9 (1.3x10^{-4})	0 (0)
False Positive	2 (0)	1.3 (0.3)	0 (0)
False Negative	0 (0)	0 (0)	0 (0)

Table 5.7 TCPechod Normal Results Across Different Input (Variance over Three Samples, One for each Input Group)

The comparison between the normal traces seems to indicate that there is not much difference between them because only a low number of abnormalities were found. When comparing normal traces with the same input, no abnormalities were found, however when comparing normal traces with different input, two abnormalities were found using direct matching and 1.3 abnormalities using relaxed matching. The two abnormities were associated with the number of bytes used for the read() and write() system calls. The AIS approach detected zero abnormal system calls.

As with Localecho, an attempted intrusion for this program is defined as an overflow of the buffer without exploit success. In this category, the direct and relaxed matching approaches detect one abnormal system call out 1 of 43, again because of the number of bytes read by the read() system call. Because the program faulted, the write() system call did not occur. The AIS matching detected zero abnormal system calls. Again, no significant difference between normal and attempted intrusion behavior.

	Direct	**Relaxed**	**AIS**
Abnormal	105 (0)	105 (0)	39.1 (1.5)
% Abnormal	70.9 (0)	70.9 (0)	26.4 (6.6x10^{-5})
False Positive	n/a	0 (0)	0 (0)
False Negative	n/a	0 (0)	65.9 (1.5)

Table 5.8 TCPechod w/Root Shell, Exploit Results (Variance over Three Samples)

Finally in the exploited category, the direct and relaxed matching approaches detected 105 abnormal system calls out of 148 total, with only 12 of those being foreign. The AIS approach detected an average of 39.1 abnormal system calls using 712.1 antibodies. This resulted in an average of 65.9 false negatives and zero false positives.

The comparison between Table 5.7 and Table 5.8 suggests that the system call information was effective in this test case. The AIS approach demonstrates a change from zero to 39.1 abnormalities. The other two methods also demonstrated a significant change. As with previous cases, the relaxed method provided slightly lower false positives than the direct method, while retaining the same abnormality matching results.

5.3.5 Non-Terminated Buffer Overflow. The experiment again exploits a stack overflow vulnerability, but instead of an openly vulnerable program like Localecho, this program has protection mechanisms in place thought to prevent a buffer from overflowing. It uses strncpy() to copy no more than the exact buffer size of user input. The shellcode was obtained from [Alep96]. The first set of test cases establishes normal traces on such input as the first case with 5 characters (bytes) in length, the second with 89 characters in length, and the third with more than 260 characters in length. The input of the third case exceeds the 256 buffer but it is checked before copying it into the buffer, preventing a buffer overflow. However because of the non-terminated buffer, a copy by sprintf() and return from the function may or may not produce a segmentation fault.

	Direct	**Relaxed**	**AIS**
Abnormal	2 (0)	1.0 (0)	0 (0)
% Abnormal	7.8 (1.2x10^{-5})	3.9 (2.9x10^{-6})	0 (0)
False Positive	2 (0)	1.0 (0)	0 (0)
False Negative	0 (0)	0 (0)	0 (0)

Table 5.9 Ntbo Normal Results Across Different Input (Variance over Three Samples, One for each Input Group)

The baseline is established by comparing the normal traces with each other using direct matching. The results comparing the same input size showed no abnormalities. Across varying sizes of input, the abnormality count for direct matching is two and relaxed matching is one. These are attributed to the number of write() bytes. The AIS matching approach identified zero abnormalities.

In the attempted intrusion category, the direct and relaxed matching approaches detect one abnormal system call out of 25, again because of the number of bytes written by the write() system call. The AIS matching detected zero abnormal system calls.

	Direct	**Relaxed**	**AIS**
Abnormal	113.3 (6.3)	113.3 (6.3)	38.8 (19.0)
% Abnormal	82.5 (9.8x10^{-6})	82.5 (9.8x10^{-6})	28.3 (1.1x10^{-3})
False Positive	n/a	0 (0)	0 (0)
False Negative	n/a	0 (0)	74.5 (27.5)

Table 5.10 Ntbo w/Root Shell, Exploit Results (Variance)

In the exploited intrusion category, the direct and relaxed matching approaches detected an average of 113.3 abnormal system calls out of an average of 137.3 total system calls, with 15 of those classified as foreign. The AIS approach detected an average of 38.8 abnormal system calls using 3066.7 antibodies. This resulted in an average of 74.5 false negatives and zero false positives.

By comparing Table 5.9 and Table 5.10, it is once again easy to see the change between the level of anomalous system calls in normal behavior and exploited behavior. The number of false negatives for the AIS approach could be improved in order to attain a more significant difference.

5.3.6 Rcond File Binding Race Condition.

This experiment exploits a file binding race condition vulnerability in the Rcond program. The baseline for normalcy is established using three cases. The length of the strings and locations/names of the files were chosen at random. The first uses a legitimate file name and a character string of 11 bytes. The second uses a different file name (different location) and a character string of 79 bytes. Finally, the third cases uses a file name different from the previous two, however same location as the second, and a character string of 37 bytes. No abnormalities between normal traces with the same input, but 1.7 abnormalities using different input. These are attributed to different inode numbers in the fstat system call.

	Direct	**Relaxed**	**AIS**
Abnormal	1.7 (0.3)	1.3 (0.3)	0 (0)
% Abnormal	5.1 (3.1x10^{-4})	4.0 (3.1x10^{-4})	0 (0)
False Positive	1.7 (0.3)	1.3 (0.3)	0 (0)
False Negative	0 (0)	0 (0)	0 (0)

Table 5.11 Race Condition Normal Results Across Different Input (Variance over Three Samples, One for each Input Group)

An attempted intrusion for this program is defined as the swapping of targeted file with a symbolic link to another file (that is owned and protected by another user) without the success of writing to that file. The intrusion may have failed because of missing the window of opportunity where the vulnerability accessible. In this category, the direct matching approaches detected between 2 abnormalities, while the relaxed matching detected none. These were attributed to the write() system call and the exit() system call because the program failed to find the target file and the exception was handled by gracefully ending. Both of these abnormalities could have been legitimate. The AIS matching detected zero abnormal system calls.

	Direct	**Relaxed**	**AIS**
Abnormal	2.0 (0)	1.0 (0)	0 (0)
% Abnormal	6.1 (0)	3.1 (0)	0 (0)
False Positive	n/a	0 (0)	0 (0)
False Negative	n/a	1.0 (0)	2.0 (0)

Table 5.12 Race Condition, Exploit Results (Variance over Three Samples)

Finally, a successful intrusion is defined as one that writes to the protected file. In this category, the direct and relaxed matching approaches detect two and one abnormal system calls respectively. The AIS approach detected zero abnormal system calls.

The comparison between Table 5.11 and Table 5.12 suggests that the system call information was ineffective in discerning the exploited behavior from the normal behavior. This is probably because of the fact that the process was never perturbed in this case, it was simply directed to operate on different data than previous executions. In these cases, the system call information may offer only a token of assistance.

5.3.7 Traceroute Heap Overflow with Root Shell. This experiment exploits a heap overflow vulnerability in the Traceroute program using root shell shellcode. The set of normal test cases is produced on such input as the first case with a remote Internet protocol (IP) address, the second with a remote IP address using the -g option to specify a gateway to use, and the third with a local IP address.

	Direct	**Relaxed**	**AIS**
Abnormal	21.5 (263)	10.5 (133.7)	0.3 (0.3)
% Abnormal	22.3 (2.9x10^{-2})	10.9 (1.5x10^{-2})	0.3 (2.7x10^{-5})
False Positive	21.5 (263)	10.5 (133.7)	0.3 (0.3)
False Negative	0 (0)	0 (0)	0 (0)

Table 5.13 Traceroute Normal Results Across Both the Same and Different Input (Variance over Three Samples, One for each Input Group)

In the normal category, the direct matching produces 7 abnormalities on the same input, however relaxed matching produces none. The abnormalities are again because of the write() system call. When comparing different normal input, an average of 26.3 abnormalities are produced from direct matching and an average of 13.7 for relaxed. The AIS matching identifies zero abnormalities. The relatively high number of abnormalities among different traces of normal behavior indicates that more examples of normal behavior need to be added.

In the attempted intrusion category, the direct and relaxed matching approaches detect 10.3 abnormal system calls out of 24.3 total system calls. The AIS matching detected an average of 1 abnormal system call using an average of 100 antibodies, and thus producing an average of 9.3 false positives.

	Direct	**Relaxed**	**AIS**
Abnormal	124.7 (0.3)	112.7 (0.3)	49.0 (1.0)
% Abnormal	89.9 (1.8x10^{-7})	81.2 (6.1x10^{-7})	35.3 (5.4x10^{-5})
False Positive	n/a	0 (0)	0 (0)
False Negative	n/a	12.0 (0)	75.7 (1.3)

Table 5.14 Traceroute w/Root Shell, Exploit Results (Variance over Three Samples)

Finally, in the exploited intrusion category, the direct matching approach detected 124.7 abnormal system calls out of 138.7 total, with only 11 of those being foreign. The relaxed method detected 112.7 abnormal system calls, and the AIS approach detected 49 abnormal system calls using an average of 906 antibodies. This resulted in approximately 75.7 false negatives and zero false positives.

The data between Table 5.13 and Table 5.14 suggests that the system call information was effective in discerning the exploited behavior from the normal behavior, even in this non-trivial program. While all three methods worked well, the AIS approach possibly performed the best because it maintained a low false positive count during normal behavior while demonstrating a healthy percentage of abnormal system calls. The relaxed method also did well by correctly categorizing almost half of the direct method's false positives as self. It is also interesting to note that the variance of abnormality counts in the normal traces was high. This is attributed to the sizable impact that various input may have on this program.

5.3.8 FTP Client Normal Behavior. The final trace and matching does not test for intrusions, but is used to illustrate the volatility of some programs such that it may be difficult to gauge normality or abnormality from the system call information. The ftp client program from the Mandrake 7.1 distribution was traced and various actions were performed using it. These include transferring files, changing directories, listing files, toggling the mode of transfer between ASCII and binary, and toggling verbose mode on and off. These actions were performed in a random manner for each trace. A direct matching comparison between the first and second trace reveals 20 abnormal system calls, while the relaxed matching reduces this to only 5. All of the abnormalities are because of the number of bytes read or wrote. As expected, the AIS matching approach did not detect any abnormalities.

A completely different story occurred when comparing the first and third traces, yielding 61 abnormal system calls via direct matching and 39 via relaxed. While a significant portion of these were read/write operations, several other types of system call were also identified with abnormal values. A manual review of the trace file revealed that many of the differences had to do with the files being checked using fstat(), memory modification with the brk() system call, or differences in the number of bytes written to the terminal with write(). Even AIS matching detected 10 abnormalities. This demonstrates that even within system information, feature selection is an important consideration for accurate anomaly detection and is specific to the process being traced.

	Direct	**Relaxed**	**AIS**
Abnormal	27.0	14.0	0
% Abnormal	15.2	7.9	0
False Positive	27.0	14.0	0

Table 5.15 FTP Client, Normal Results, Trained w/5 Normal Traces

Finally, when using five traces of normality and testing a sixth normal trace, the direct matching produces 27 abnormal system calls and zero foreign ones. Table 5.15 provides a comparison of the three approaches using normal data. The relaxed approach produces only 14 abnormal while the AIS approach does not produce any. This demonstrates that some programs may require significant amounts of normality tracing before the false positives subside. This is often commensurate with the complexity of the program in question, where the more complex a system is, the more normality information is required to reach a comfortable false positive count. Also note that no variance is supplied with this table because this is one particular comparison rather than the average of several comparisons.

5.4 Overall Analysis

The results of these experiments provide information about how processes are affected by attempted and successful intrusions. With regard to attempted intrusions, the results indicate that there is generally no significant difference between normal and attempted intrusion traces. While some attempts were detectable by signals of segmentation faults and illegal instructions in the traces, this analysis prototype was limited to only system call information. System calls prior to the signals did not provide indications of intrusions in any of the test cases, except for abnormal numbers of bytes read by the read() system call. The attempted race condition intrusion did not produce significant abnormal system calls or fault signals.

With regards to successful intrusions, the results indicate that there are some cases where significant differences between normal and intrusion traces were found. In the cases of buffer overflows with root shell shellcode, there were a number of abnormal matches using direct matching. Because of an increased total number of system calls, a percentage of abnormality out of total system calls may be a more appropriate measure. In general, the direct matching with normality had a range of 4% to 26% of abnormal calls whereas the successful intrusion that were detected produced a range of 70% to 90% of abnormal system calls. Table 5.16 provides provides a comparison of abnormality in normal, attempted and exploited data using direct matching.

The relaxed matching approach also did well by retaining similar matching results as the direct method, but improved on the number of false positives. Table 5.17 provides a comparison of abnormality in normal, attempted and exploited data using relaxed matching. This method did detect a few less abnormalities than direct matching in the exploited Traceroute case. The strength

Program	% Abnormal in Normal Data	% Abnormal in Attempted Data	% Abnormal in Exploited Data
Localecho w/Root Shell	4.0 ($7.1\text{x}10^{-6}$)	4.0 (0)	81.5 (0)
Localecho w/Xor Shell	4.0 ($7.1\text{x}10^{-6}$)	0.0 (0)	82.9 ($3.8\text{x}10^{-6}$)
Localecho w/Chmod Shell	4.0 ($7.1\text{x}10^{-6}$)	0.0 (0)	5.3 ($5.3\text{x}10^{-4}$)
TCPecho w/Root Shell	4.4 ($2.8\text{x}10^{-6}$)	2.3 (0)	70.9 (0)
Ntbo w/Root Shell	7.8 ($1.2\text{x}10^{-5}$)	4.0 (0)	82.5 ($9.8\text{x}10^{-6}$)
RCond	5.1 ($3.1\text{x}10^{-4}$)	5.2 ($1.4\text{x}10^{-3}$)	6.1 (0)
Traceroute	22.2 ($2.9\text{x}10^{-2}$)	42.4 ($7.9\text{x}10^{-4}$)	89.9 ($1.8\text{x}10^{-7}$)
FTP Client	26.0 ($2.2\text{x}10^{-2}$)	n/a	n/a

Table 5.16 Average Percent (Variance over Three Samples) of Abnormal System Calls out of Total System Calls using Direct Matching

of relaxing the normality definition to reduce false positives is also a weakness because any non-self near a normal point is considered normal, thus increasing false negatives.

Program	% Abnormal in Normal Data	% Abnormal in Attempted Data	% Abnormal in Exploited Data
Localecho w/Root Shell	1.2 ($4.6\text{x}10^{-4}$)	0.0 (0)	81.5 (0)
Localecho w/Xor Shell	1.2 ($4.6\text{x}10^{-4}$)	0.0 (0)	82.9 ($3.8\text{x}10^{-6}$)
Localecho w/Chmod Shell	1.2 ($4.6\text{x}10^{-4}$)	0.0 (0)	5.3 ($5.3\text{x}10^{-4}$)
TCPecho w/Root Shell	2.9 ($1.3\text{x}10^{-4}$)	2.3 (0)	70.9 (0)
Ntbo w/Root Shell	3.9 ($2.9\text{x}10^{-6}$)	4.0 (0)	82.5 ($9.8\text{x}10^{-6}$)
RCond	4.0 ($3.1\text{x}10^{-4}$)	0 (0)	3.0 (0)
Traceroute	10.9 ($1.5\text{x}10^{-2}$)	4.1 ($4.0\text{x}10^{-6}$)	81.2 ($6.1\text{x}10^{-7}$)
FTP Client	13.5 ($1.5\text{x}10^{-2}$)	n/a	n/a

Table 5.17 Average Percent (Variance over Three Samples) of Abnormal System Calls out of Total System Calls using Relaxed Matching

The AIS matching approach performed well by detecting zero false positives in almost all of the cases. Table 5.18 provides a comparison of abnormality in normal, attempted and exploited data using AIS. While it did not detect the high number of abnormalities that the other two approaches did, the difference between normal behavior and intrusive behavior was significantly large enough make a distinction. Further maturation or the inclusion of more antibodies better dispersed may increase that distinction.

Although the number of bytes used in read() and write() system calls were observed in many of the experiments, others were pre-tested for stability between runs and found to change at almost every execution.

Program	**% Abnormal in Normal Data**	**% Abnormal in Attempted Data**	**% Abnormal in Exploited Data**
Localecho w/Root Shell	0 (0)	0 (0)	30.0 (2.4x10^{-4})
Localecho w/Xor Shell	0 (0)	0 (0)	27.7 (1.2x10^{-3})
Localecho w/Chmod Shell	0 (0)	0 (0)	5.3 (5.3x10^{-4})
TCPecho w/Root Shell	0 (0)	0 (0)	26.4 (6.6x10^{-5})
Ntbo w/Root Shell	0 (0)	1.3 (5.3x10^{-4})	28.3 (1.1x10^{-3})
RCond	0 (0)	1.6 (7.6x10^{-4})	0 (0)
Traceroute	0.3 (3.6x10^{-5})	4.1 (4.0x10^{-6})	35.3 (5.4x10^{-5})
FTP Client	2.3 (1.6x10^{-3})	n/a	n/a

Table 5.18 Average Percent (Variance over Three Samples) of Abnormal System Calls out of Total System Calls using AIS Matching

The selection of features used from the system call information was found important for minimizing false positives in normal traces with various input. For example, system calls that have time, process identification numbers, or user/group identification numbers as a parameter or result were often identified as abnormal because over several executions they are different. These particular features, but not the whole system call, were suppressed during the experiments. This was necessary because regardless of the amount of training, these values are identified as false positives because there is no stability between traces. The determining factor of whether a feature should be suppressed is if it changes during training. Two particular features, the number of bytes read using the read() system call as well as the number of bytes wrote with the write() system call, were not suppressed because it was believed that they may provide some indication of extreme sizes of input/output, even though there is the possibility that these feature will change over several executions because of different input.

Finally, comparisons of normal traces using Traceroute and FTP client provided indication that different input may drastically affect the system call information. This was apparent from the high variance of Traceroute features and the necessity of many normal traces of FTP client before the false positives subsided. This suggests that it is important to have a variety of normality traces in order to capture normal behavior in non-pedagogical programs.

5.5 Summary

This chapter described the design of experiments used to test the hypothesis made in Chapter 1. Some indication was established that system call information is affected by certain intrusions. This is evident from the change in the number of abnormal system calls in the training data compared to the tested behavior data. Three matching methods were evaluated and results from each were described. The next and final chapter states the conclusions drawn from these experiments, the limitations of this research and proposed future work.

VI. Conclusions and Recommendations

The previous chapters discuss the design and implementation of a prototype anomaly detection analyzer that uses three approaches for identifying abnormal behavior. Chapter 5 applied these approaches to several program traces of system call information to determine what effect intrusions had on these processes, as well as compare and analyze methods of detection. This chapter consists of conclusions, limitations and recommendations for future work. Section 6.1 presents some general conclusions regarding the entire research effort. Section 6.2 provides limitations of this approach, while Section 6.3 gives some recommendations for future research.

6.1 Conclusions

The purpose of this research was to validate the hypothesis that the internal information from a single computer process may be used to effectively detect computer intrusions targeted at that process. An assumption was sustained that anomalous behavior indicates possible intrusions or attempted intrusions. Analysis of the experimental results indicate that in certain cases the difference between the normal trace data and the exploited trace data are different enough to distinguish the occurrence of anomalous behavior. For example, using direct matching on the non-trivial program Traceroute had an average of 22.2 percent abnormal system calls when comparing different traces of normal activity, but the traces of exploited activity reveal an average of 89.9 percent abnormal system calls. Neither values had a variance greater than $3.0\text{x}10^{-2}$. Even though the normal trace data had a significant number of false positives, the exploited trace data showed a higher percentage of abnormality. The large difference between the two values provides satisfactory indication of abnormality. Other program traces also provided similar results regarding the difference between the average normality and the average exploit traces, but this difference was not always large. When the differences were large (using an arbitrary difference of 20%), it was obvious that the process was perturbed. When it is not large enough, some ambiguity may remain. Localecho with chmod shellcode and Rcond were two examples where the difference between the averages were less than two percent using direct matching. Because of the small differences, some doubt remains whether the process was seriously perturbed.

It was also demonstrated that system call information may not be as useful for identifying attempted intrusions because they reflect normality too closely. For example, using direct matching on the trivial programs Localecho with xor root shell had an average of four percent abnormal system calls when comparing several different traces of normal activity, but the traces of exploited activity reveal an average of zero percent abnormal system calls because of the lack of difference between the normal and attempted traces. The other pedagogical programs performed in a similar manner such that they produced the same percentage or lower of abnormal system calls. However, while using direct matching with Traceroute, the average percent of abnormal calls was 42.4 percent for attempted trace data as compared to 22.2 percent for normal trace data (variance less than $3.0\text{x}10^{-2}$). It would seem that this is a considerable indication of an attempted intrusion, but when comparing the same data using relaxed matching, the abnormal percentage of the normal traces is 10.9 and the abnormal percentage of the attempted traces is only 4.1 (variance not greater than $1.5\text{x}10^{-2}$). This situation was created by attempted behavior being very close but not an exact match to the training data. When using AIS matching, the false positives were low enough to discriminate between percent abnormal in normal data (0.3) and in attempted data (4.1) with a variance less than $3.6\text{x}10^{-5}$. Because there were only a few instances where the results provided a large difference of more abnormal calls in the attempted traces than the normal traces, this suggests that system call information is useful for identifying the after-effects of intrusions but not necessarily the causal situations. Detecting both causal and effect indicators are important in a layered defense strategy.

The secondary research objective was to explore methods of discerning these system call clues. Three methods were evaluated: direct matching, relaxed matching, and artificial immune system matching. Direct matching was used as the baseline for comparing raw differences in trace data, but this approach misclassifies trace data that is not an exact match yet within a short range of self. This was evident from the previous example concerning Traceroute. It also produced a significant number of false positives over the other two approaches in all cases. This approach may require that an intrusion is identified only if the percentage of abnormality meets a threshold, however the threshold may require tuning per program.

The relaxed matching approach identified about half of the false positives compared to direct matching while retaining almost the same percentages of abnormality in exploited traces. As with direct matching, a threshold may still be necessary, but since may of the false positives were filtered out, the difference between normal and exploited traces becomes more apparent. This may provide an increase in confidence that an abnormality has occurred. The limitation of this approach is again the determination of a threshold value, but also a range value (possible for each feature) that determines how far normality reaches beyond the training data. Identifying these values may be tedious and probably program dependent.

Finally, the AIS matching approach provided the lowest number of false positives while retaining the ability to correctly identify the abnormal behavior in the attempted and exploited traces. The percentage of abnormal system calls in normal trace data was zero for all of the pedagogical programs and less than three percent for non-trivial programs. A threshold again may be necessary, but it would probably be a low value. It is interesting to note that the low false positive rate provides a stronger sensitivity than the other two approaches. For example, Localecho with chmod shellcode was identified with more confidence because the abnormal percentage of the normal traces is zero while the abnormal percentage of the exploited traces is 5.3 (variance in both values not greater than $5.3\text{x}10^{-4}$). The difference between these two values is larger than the difference between normal (4.0) and exploited (5.3) percentages of abnormality using direct matching. It is also important to note that the number of false negatives increased. The percentage of abnormal system calls in exploited traces is in some cases less than half compared to direct and relaxed matching approaches. This is probably because areas of non-self space were not covered by antibodies. This can be addressed by several methods such as increasing the number of antibodies, performing more affinity maturation generations, or possibly by explicitly forcing coverage.

6.2 Limitations

One limitation associated with this research is based on the use of previous traces to define normality for a particular process. The usage of a process in terms of input and environment may differ depending on various users. Certain programs may allow such a large input domain that capturing enough normality for this approach to work is infeasible.

Another limitation of this work is associated with the stated assumption that anomalies indicate intrusions or attempted intrusions. This research does not provide a means of detecting intrusions. It only provides a means of detecting anomalies that may indicate intrusions and thus supports the intrusion detection effort in that regard. It is important to note that this work does not support any information on the intent of the user, only limited information on how the process behaved compared to previous runs.

Finally, this research only tested a limited number of intrusion methods. These intrusions were limited to a few examples of condition validation and synchronization fault exploits. No information suggests that other faults may be detected using system call information.

6.3 Recommendations for Future Research

One avenue for future research involves expanding the number of system calls sampled. Ideally these tests should have been performed while sampling all 197 system calls. This would provide a better measurement of detecting abnormal as well as foreign system calls. In addition, signaling information may also be useful for detecting some attempted intrusions because they may reveal segmentation faults or illegal instruction faults. From this group of 197 system calls, features should be tested for detection effectiveness. Those features not contributing to the effectiveness of or inhibiting detection should be removed from sampling. Feature reduction may reveal that only a select group of features are necessary to perform intrusion detection.

Another possible area for future research might be to hone the AIS approach to produce more distributed antibodies. The use of niching techniques may assist in providing better non-self coverage. Additionally, comparison of other methods of affinity maturation may be useful. One approach may use a greedy strategy that always expands the feature with the most room to grow first.

A final recommendation for future research involves dynamically adjusting the list of system calls that may be considered abnormal. For instance, if a feature changes in multiple normality traces, then it should not be counted in the future as abnormal, but rather volatile. Volatile feature suppression may assist direct matching by removing those features that change too often to be a measure of normality. This may reduce the number of false positives because the system no longer treats the feature as a reliable indicator of anomalies.

Appendix A. Constructed Programs

A.1 Localecho Source Code

```
/* Localecho.c */

#include <stdio.h>

#define SMALLBUFSIZE 100
#define LARGEBUFSIZE 256

int vulnbufcopy(char* buf)
{
  // Create a smaller buffer to overflow
  char smallBuffer[SMALLBUFSIZE];
  int i;

  // Copy the large into the small
  strcpy( smallBuffer, buf );

  return 0;
};

int main(int argc, char* argv[])
{
  // Create our large buffer to feed input
  char largeBuffer[LARGEBUFSIZE];
  int i;

  // Initialize buffer to null
  for(i = 0; i < LARGEBUFSIZE; i++)
  {
    largeBuffer = '\0';
  }

  // Read in the command line argument, if any
  if(argc > 1)
  {
    strncpy(largeBuffer, argv[1], LARGEBUFSIZE);
  }
  else
  {
    printf("usage: %s msg", argv[0]);
    return 0;
  }

  // Call the vulnerable function
  vulnbufcopy(largeBuffer);

  // Echo out the input
  printf("echoed: %s\n", largeBuffer);
  return 0;
}
```

A.2 TCPecho, TCPechod Source Code

```
/*
 * Network.h
 * authored by Douglas E. Comer & David L. Stevens
 * "Internetworking with TCP/IP Vol III"
 *
 * Modified by Mark Reith
 */

#ifndef NETWORK_H
#define NETWORK_H

int connectsock(const char* host, const char* service,
        const char* transport);
int errexit(const char* format, ...); int passivesock(const char*
        service, const char* transport, int qlen);

#endif

/*
 * Network.c
 * authored by Douglas E. Comer & David L. Stevens
 * "Internetworking with TCP/IP Vol III"
 *
 * Modified by Mark Reith
 */

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <string.h>
#include <netdb.h>
#include <stdarg.h>
#include <stdio.h>
#include "network.h"

#ifndef INADDR_NONE\\
#define INADDR_NONE  0xffffffff\\
#endif
```

```
u_short portbase = 0;

extern int errno;

int connectsock(const char* host, const char* service,
        const char* transport)
{
  struct hostent*    phe;
  struct servent*    pse;
  struct protoent*   ppe;
  struct sockaddr_in sin;
  int s, type;

  memset(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;

  /* Map service name to port number */
  if( pse = getservbyname( service, transport) )
  {
    sin.sin_port = pse->s_port;
  }
  else if( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
  {
    errexit("can’t get \"%s\" service entry\n", service);
  }

  /* Map host name to IP address, allowing for dotted decimal */
  if( phe = gethostbyname(host) )
  {
    memcpy(&sin.sin_addr, phe->h_addr, phe->h_length);
  }
  else if( (sin.sin_addr.s_addr = inet_addr(host)) == INADDR_NONE )
  {
    errexit("can’t get \"%s\" host entry\n", host);
  }

  /* Map transport protocol name to protocol number */
  if( (ppe = getprotobyname(transport)) == 0 )
  {
    errexit("can’t get \"%s\" protocol entry\n", service);
  }

  /* Use protocol to choose a socket type */
  if( strcmp( transport, "udp") == 0 )
  {
    type = SOCK_DGRAM;
  }
  else
  {
    type = SOCK_STREAM;
  }
  /* Allocate a socket */
  s = socket(PF_INET, type, ppe->p_proto);
  if( s < 0 )
  {
    errexit("can't create socket: %s\n", strerror(errno));
  }

  /* Connect the socket */
  if( connect(s, (struct sockaddr *)&sin, sizeof(sin)) < 0 )
  {
    errexit("can't connect to %s.%s: %s\n", host, service,
        strerror(errno));
  }

  return s;
};

int errexit(const char* format, ...)
{
  va_list args;

  va_start(args, format);
  vfprintf(stderr, format, args);
  va_end(args);
  exit(1);
};

int passivesock(const char* service, const char* transport,
    int qlen)
{
  struct servent*    pse;
  struct protoent*   ppe;
  struct sockaddr_in sin;
  int s, type;

  memset(&sin, 0, sizeof(sin));
  sin.sin_family = AF_INET;
  sin.sin_addr.s_addr = INADDR_ANY;

  /* Map service name to port number */
  if( pse = getservbyname( service, transport) )
  {
    sin.sin_port = htons(ntohs((u_short)pse->s_port) + portbase);
  }
  else if( (sin.sin_port = htons((u_short)atoi(service))) == 0 )
  {
    errexit("can't get \"%s\" service entry\n", service);
  }

  /* Map protocol name to protocol number */
  if( (ppe = getprotobyname(transport)) == 0 )
  {
    errexit("can't get \"%s\" protocol entry\n", service);
  }

  /* Use protocol to choose a socket type */
  if( strcmp( transport, "udp") == 0 )
  {
    type = SOCK_DGRAM;
  }
  else
  {
    type = SOCK_STREAM;
  }

  /* Allocate a socket */
  s = socket(PF_INET, type, ppe->p_proto);
  if( s < 0 )
  {
    errexit("can't create socket: %s\n", strerror(errno));
  }

  /* Bind the socket */
  if( bind( s, (struct sockaddr *)&sin, sizeof(sin)) < 0 )
  {
    errexit("can't bind to %s port: %s\n", service, strerror(errno));
  }
  if( type == SOCK_STREAM && listen(s, qlen) < 0 )
  {
    errexit("can't listen on %s port: %s\n", service, strerror(errno));
  }

  return s;
};


/*
 * TCPecho.c
 * authored by Douglas E. Comer & David L. Stevens
 * "Internetworking with TCP/IP Vol III"
 *
 * Modified by Mark Reith
 */

#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include "network.h"

extern int errno;
```

```
int TCPecho(const char* host, const char* service, char* mesg);
int connectTCP(const char* host, const char* service);

#define BUFSIZE 256

int main(int argc, char* argv[])
{
  char* host = "localhost";
  char* service = "echo";
  char  largeBuffer[BUFSIZE];
  int   i;

  if( argc % 2 != 0 )
  {
    fprintf(stderr, "usage: TCPecho mesg\n");
    exit(1);
  }

  for( i = 0; i < BUFSIZE; i++ )
  {
    largeBuffer[i] = '\0';
  }

  strncpy(largeBuffer, argv[1], BUFSIZE);
  TCPecho(host, service, largeBuffer);
  exit(0);
};

int TCPecho(const char* host, const char* service, char* mesg)
{
  int  s, n, i;
  int  outchars, inchars;
  char buf[BUFSIZE];

  s = connectTCP(host, service);
  outchars = strlen(mesg);
  printf("transmitted: %d\n", outchars);
  (void) write(s, mesg, outchars);

  for( i = 0; i < BUFSIZE; i++ )
  {
    buf[i] = '\0';
  }

  n = read(s, &buf, BUFSIZE);
  if( n < 0 )
  {
    errexit("socket read failed: %s\n", strerror(errno));
  }
  if( n >= BUFSIZE )
  {
    buf[BUFSIZE-1] = '\0';
  }

  printf("%s\n", buf);
};

int connectTCP(const char* host, const char* service)
{
  return connectsock( host, service, "tcp" );
};

/*
 * TCPechod.c
 * authored by Douglas E. Comer & David L. Stevens
 * "Internetworking with TCP/IP Vol III"
 *
 * Modified by Mark Reith
 */

#include <sys/types.h>
#include <sys/signal.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/wait.h>
#include <sys/errno.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <stdarg.h>
#include <netdb.h>
#include "network.h"

#define QLEN            5
#define LARGEBUFSIZE    256
#define SMALLBUFSIZE    100

extern int errno;

void reaper(int); int  TCPechod(int fd);
int  passiveTCP(const char* service, int qlen);
int vulnbufcopy(char* buf);

unsigned long get_sp(void)
{
  __asm__("movl %esp,%eax");
}
```

```
int main(int argc, char* argv[])
{
  char*  service = "echo";
  struct sockaddr_in fsin;
  int    alen;
  int    msock;
  int    ssock;

  switch(argc)
  {
    case 1:  { break; }
    case 2:  { service = argv[1]; break; }
    default: { errexit("usage: TCPechod [port]\n"); }
  }

  msock = passiveTCP(service, QLEN);

  (void) signal(SIGCHLD, reaper);

  while(1)
  {
    alen = sizeof(fsin);
    ssock = accept(msock, (struct sockaddr *)&fsin, &alen);

    if( ssock < 0 )
    {
      if( errno == EINTR )
      {
        continue;
      }
      errexit("accept: %s\n", strerror(errno));
    }

    switch(fork())
    {
      case 0:  { /*child*/  (void) close(msock); exit(TCPechod(ssock)); }
      default: { /*parent*/ (void) close(ssock); break; }
      case -1: { /*error*/  errexit("fork: %s\n", strerror(errno)); }
    }
  }
};

int TCPechod(int fd)
{
  char buf[LARGEBUFSIZE];
  int  cc;
  int  i;

  for( i = 0; i < LARGEBUFSIZE; i++ )
  {
    buf[i] = '\0';
  }

  while( cc = read(fd, buf, sizeof(buf)) )
  {
    if( cc < 0 )
    {
      errexit("echo read: %s\n", strerror(errno));
    }

    printf("received: %d\n", cc);

    vulnbufcopy(buf);

    if( write(fd, buf, cc) < 0 )
    {
      errexit("echo write: %s\n", strerror(errno));
    }

    for( i = 0; i < LARGEBUFSIZE; i++ )
    {
      buf[i] = '\0';
    }
  }

  return 0;
};

int vulnbufcopy(char* buf)
{
  // Create a smaller buffer to overflow
  char smallbuf[SMALLBUFSIZE];

  // Copy the large into the small
  strcpy( smallbuf, buf );

  return 0;
};

int passiveTCP(const char* service, int qlen)
{
  return( passivesock(service, "tcp", qlen) );
};

void reaper(int sig)
{
  int status;

  while( wait3(&status, WNOHANG, (struct rusage *)0) >= 0 );
};
```

A.3 Ntbo Source Code

```
/* Ntbo.c */
#include <stdlib.h>
#include <stdio.h>

#define BUFSIZE 256

int main(int argc, char* argv[])
{
  /* Adjacent buffers in stack */
  char buffer1[BUFSIZE];
  char buffer2[BUFSIZE];

  // Clean buffers
  memset(buffer1, 0, BUFSIZE);
  memset(buffer2, 0, BUFSIZE);

  if( argc < 3 )
  {
    printf("usage: %s <arg1> <arg2>\n", argv[0]);
    exit(0);
  }

  /* strncpy places a null terminating character
   * when the input is < BUFSIZE, but does not do
   * so when the input is >= BUFSIZE */
  strncpy(buffer1, argv[1], BUFSIZE);
  strncpy(buffer2, argv[2], BUFSIZE);

  vulnfunction(buffer2);

  return 0;
}

int vulnfunction(char* p) {
  /* Since the input size was checked earlier,
   * we can assume that this buffer will be the same
   * size plus 9 bytes for the string "message: " */
  char buffer3[BUFSIZE+9];

  // Clean buffer
  memset(buffer3, 0, BUFSIZE);

  // Let the user see how long the input string was
  fprintf(stderr, "p is %d bytes long\n", strlen(p));

  /* Assume that input p is <= BUFSIZE; if not
   * then sprintf will continue copying through
   * buffer2 into buffer1 until it finds a null
   * character */
  sprintf(buffer3, "message: %s", p);

  printf("%s\n", buffer3);

  return 0;
}
```

A.4 Race Condition Source Code

```
/* rcond.c */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main( int argc, char* argv[] ) {
  struct stat st;
  FILE* fp;

  // Test for appropriate number of arguments
  if( argc != 3 )
  {
    fprintf( stderr, "usage: %s file message\n", argv[0] );
    exit(EXIT_FAILURE);
  }

  // Test existance of file
  if( stat( argv[1], &st ) < 0 )
  {
    fprintf( stderr, "can't find %s\n", argv[1] );
    exit(EXIT_FAILURE);
  }

  // Test for appropriate permissions
  if( st.st_uid != getuid() )
  {
    fprintf( stderr, "not the owner of %s\n", argv[1] );
    exit(EXIT_FAILURE);
  }

  // Test for regular file (not symbolic link)
  if( ! S_ISREG (st.st_mode) )
  {
    fprintf( stderr, "%s is not a normal file\n", argv[1] );
    exit(EXIT_FAILURE);
  }

  // Artificial pause to allow easy intrusion
  sleep(30);

  // Open the file
  if( (fp = fopen( argv[1], "w" )) == NULL )
  {
    fprintf( stderr, "can't open" );
    exit(EXIT_FAILURE);
  }

  // Write to the opened file
  fprintf( fp, "%s\n", argv[2] );
  fclose(fp);
  fprintf( stderr, "Write Ok\n");

  return 0;
};
```

Bibliography

[Alep96] AlephOne. “Smashing The Stack For Fun And Profit,” *Phrack*, *7(49)* 1996. Available electronically at `http://www.phrack.org/show.php?p=49&a=14`.

[Almg01] Almgren, Magnus and Ulf Lindqvist. “Application-Integrated Data Collection for Security Monitoring.” *Proceedings of the Fourth International Symposium on Recent Advances in Intrusion Detection, RAID 2001*. 22–36. Davis, CA: Springer-Verlag, October 2001.

[Amor99] Amoroso, Edward. *Intrusion Detection: An Introduction to Internet Surveillance, Correlation, Trace Back, Traps, and Response*. Sparta, NJ: Intrusion.Net Books, 1999.

[Ande95] Anderson, Debra, et al. *Next-generation Intrusion Detection Expert System (NIDES): A Summary*. SRI International, 1995. Available electronically at `http://www.sdl.sri.com/papers/4sri/`.

[Anon01] Anonymous. “Once upon a free()...,” *Phrack*, *11(57)* 2001. Available electronically at `http://www.phrack.org/show.php?p=57&a=9`.

[Asla96] Aslam, Taimur, et al. *Use of A Taxonomy of Security Faults*. Technical Report TR-96-051, Purdue University, 1996.

[Axel00] Axelsson, Stefan, “Intrusion Detection Systems: A Survey and Taxonomy.” World Wide Web, 2000. Available electronically at `http://citeseer.nj.nec.com/axelsson00intrusion.html`.

[Bace00] Bace, Rebecca G. *Intrusion Detection*. Indianapolis: Macmillan Technical Publishing, 2000.

[Back00] Bäck, Thomas, et al., editors. *Evolutionary Computation 1: Basic Algorithms and Operators*. Institute of Physics, 2000.

[Bala98] Balasubramaniyan, Jai Sundar, et al. *An Architecture for Intrusion Detection Using Autonomous Agents*. Technical Report 98/05, Purdue University, 1998.

[Bish99] Bishop, Matt. “Vulnerabilities Analysis.” *Proceedings of the 2nd International Workshop on Recent Advances in Intrusion Detection*. 1999.

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

[Bold03] Boldyshev, Konstantin, “Linux Assembly.” World Wide Web, 2003. Observed on 31 January 2003 at `http://linuxassembly.org/`.

[CERT02] CERT Coordination Center. *CERT/CC Statistics 1988-2002*, 2002. Observed on 31 January 2003 at `http://www.cert.org/stats/cert_stats.html`.

[Cowa01] Cowan, Crispin, et al. “RaceGuard: Kernel Protection from Temporary File Race Conditions.” *Proceedings of the 10th USENIX Security Symposium, August 2001*. 2001. Available electronically at URL `http://www.cs.virginia.edu/~evans/usenix01-abstract.html`.

[Cowa00] Cowan, Crispin, et al., “Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade.” World Wide Web, 2000. Available electronically at `http://www.cse.ogi.edu/DISC/projects/immunix/publications.html`.

[Cros03] Crosbie, Mark, “Intrusion Detection Systems.” World Wide Web, 2003. Observed on 31 January 2003 at URL `http://www.cerias.purdue.edu/coast/intrusion-detection/ids.html`.

[Cros95] Crosbie, Mark and Eugene H. Spafford. *Active Defense of a Computer System using Autonomous Agents*. Technical Report 95-0008, Purdue University, 1995.

[Dasg99] Dasgupta, Dipankar, editor. *Artificial Immune Systems and Their Applications*. Berlin: Springer-Verlag, 1999.

[Dasg96] Dasgupta, Dipankar and Stephanie Forrest. "Novelty Detection in Time Series Data using Ideas from Immunology." *Proceedings of the 5th International Conference on Intelligent Systems, June 1996*. 1996. Available electronically at URL `http://www.msci.memphis.edu/~dasgupta/publications.html`.

[Denn87] Denning, Dorothy E. "An Intrusion Detection Model," *IEEE Transactions on Software Engineering*, *13*(2):222–232, 1987.

[Eski00] Eskin, Eleazar. "Anomaly Detection over Noisy Data using Learned Probability Distributions." *Proceedings of 17th International Conference on Machine Learning*. 2000. Available electronically at URL `http://citeseer.nj.nec.com/eskin00anomaly.html`.

[Forr94] Forrest, Stephanie, et al. "Self-Nonself Discrimination in a Computer." *Proceedings of 1994 Symposium on Research in Security and Privacy*. 1994. Available electronically at URL `http://citeseer.nj.nec.com/forrest94selfnonself.html`.

[Forr96b] Forrest, Stephanie, et al. "Computer Immunology," *Communications of the ACM*, *40(10)*:88–96 1996. Available electronically at URL `http://www.cs.unm.edu/~forrest/papers.html`.

[Forr96] Forrest, Stephanie, et al. "A Sense of Self for Unix Processes." *Proceedings of 1996 Symposium on Research in Security and Privacy*. 1996. Available electronically at URL `http://citeseer.nj.nec.com/forrest96sense.html`.

[Free03] Free Software Foundation, Inc. *The GNU C Library*, 2003. Observed on 31 January 2003 at URL `http://www.gnu.org/manual/glibc-2.2.3/html_node/libc_toc.html`.

[Fuku98] Fukuda, Toyoo, et al. *Immunity-Based Management System for a Semiconductor Production Line*, 278–288. Springer-Verlag, 1998.

[Germ02] Germanow, Abner, et al., "The Injustice of Insecure Software." World Wide Web, 2002. Available electronically at `http://www.atstake.com/research/reports/acrobat/atstake_injustice.pdf`.

[Gonz02] González, Fabio and Dipankar Dasgupta. "An Immunogenetic Technique to Detect Anomalies in Network Traffic." *Gecco 2002: Proceedings of the Genetic and Evolutionary Computation Conference*. 1081–1088. Morgan Kaufmann Publishers, 2002. Available electronically at URL `http://www.cs.memphis.edu/~gonzalef/`.

[Harm02] Harmer, Paul, et al. "A Distributed Agent Based Architecture for Computer Security Applications," *IEEE Transactions On Evolutionary Computation, June 2002*, *6(3)* 2002.

[Hart99] Hart, Emma and Peter Ross. "An immune system approach to scheduling in changing environments." *Proceedings of the Genetic and Evolutionary Computation Conference*. 1559–1566. Morgan Kaufmann, 1999. Available electronically at URL `http://www.dcs.napier.ac.uk/~peter/grl22232/papers.html`.

[Head90] Heady, Richard, et al. *The Architecture of a Network Level Intrusion Detection System*. Technical Report CS90-20, University of New Mexico, 1990.

[Hofm00] Hofmeyr, Steven and Stephanie Forrest. "Architecture for an Artificial Immune System," *IEEE Transactions on Evolutionary Computation*, *7(1)*:45–68 2000.

[Hof98p] Hofmeyr, Steven, et al., "Distributed Network Intrusion Detection: An Immunological Approach." World Wide Web, 1998. Presentation available electronically at URL `http://cs.unm.edu/~steveah/`.

[Hofm98] Hofmeyr, Steven, et al. "Intrusion Detection using a Sequence of System Calls," *Journal of Computer Security*, *6*:151–180 1998.

[Howe03] Howe, Denise, "The Free On-line Dictionary of Computing." World Wide Web, 2003. Observed on 31 January 2003 at `http://www.foldoc.org/`.

[Hyde03] Hyde, Randall, "WEBster: Linux Assembly Language Programming." World Wide Web, 2003. Observed on 31 January 2003 at `http://webster.cs.ucr.edu/Page_Linux/0_Linux.html`.

[Ilgu95] Ilgun, Koral, et al. "State Transition Analysis: A Rule-Based Intrusion Detection Approach," *IEEE Transactions on Software Engineering, March 1995*, *21(3)*:181–199 1995.

[Inse02] INSECURE.ORG. *Insecure*, 2002. Observed on 31 January 2003 at URL `http://www.insecure.org/`.

[Irvi93] Irvine, Kip. *Assembly Language for the IBM-PC*. Prentice-Hall, Inc., 1993.

[Jone99] Jones, Anita and Robert S. Sielken, "Computer System Intrusion Detection: A Survey." World Wide Web, 1999. Available electronically at `http://www.cs.virginia.edu/~jones/IDS-research/`.

[Kaem01] Kaempf, Michel. "Vudo - An object superstitiously believed to embody magical powers," *Phrack*, *11(57)* 2001. Available electronically at `http://www.phrack.org/show.php?p=57&a=8`.

[Land94] Landwehr, Carl E., et al. "A Taxonomy of Computer Program Security Flaws, with Examples," *ACM Computing Surveys*, *26(3)*:211–254 1999. Available electronically at URL `http://chacs.nrl.navy.mil/publications/CHACS/1994/1994landwehr-acmcs.pd%f`.

[Laro01] Larochelle, David and David Evans. "Statically Detecting Likely Buffer Overflow Vulnerabilities." *Proceedings of the 10th USENIX Security Symposium, August 2001*. 2001. Available electronically at URL `http://www.cs.virginia.edu/~evans/usenix01-abstract.html`.

[Lea03] Lea, Doug, "A Memory Allocator." World Wide Web, 2003. Observed on 31 January 2003 at URL `http://gee.cs.oswego.edu/dl/html/malloc.html`.

[Lipp99] Lippmann, Richard P., et al. *DARPA Intrusion Detection Evaluation*, 1999. Massachusetts Institute of Technology World Wide Web Site, URL `http://www.ll.mit.edu/IST/ideval/`.

[Mats02] Matsumoto, Makoto, "Mersenne Twister Home Page." World Wide Web, 2002. Available electronically at URL `http://www.math.keio.ac.jp/~matumoto/emt.html`.

[Maxi00] Maxion, Roy A. and Kymie M.C. Tan. "Benchmarking Anomaly-Based Detection Systems." *Proceedings of the 1st International Conference on Dependable Systems & Networks, June 2000*. 623–630. IEEE Computer Society Press, 2000. Available electronically at URL `http://citeseer.nj.nec.com/maxion00benchmarking.html`.

[MITR00] The MITRE Corporation. *CVE Common Vulnerabilities and Exposures: CVE-2000-0949*, 2000. Observed on 31 January 2003 at URL `http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2000-0949`.

[MITR02] The MITRE Corporation. *CVE Common Vulnerabilities and Exposures*, 2002. Observed on 31 January 2003 at URL `http://www.cve.mitre.org/cve/`.

[Mukh94] Mukherjee, Biswanath, et al. "Network Intrusion Detection," *IEEE Network*, *8(3)*:26–41 1994.

[Muns01] Munson, John and Scott Wimer. "Watcher: The Missing Piece of the Security Puzzle." *Proceedings of 17th Annual Computer Security Applications Conference*. 2001. Available electronically at URL `http://www.acsac.org/2001/abstracts/thu-1030-a-munson.html`.

[OSDN03] Open Source Development Network. *Project: strace: Summary*, 2003. Observed on 31 January 2003 at URL `http://sourceforge.net/projects/strace/`.

[Paig96] Paige, Emmett Jr., "The Future of Information Security." Speech, 1996. World Wide Web Page. URL `http://www.defenselink.mil/speeches/1996/s19960625-paige.html`.

[Pan92] Pan, Hsin and Eugene H. Spafford, "Heuristics for Automatic Localization of Software Faults." World Wide Web, 1992. Available electronically at URL `http://citeseer.nj.nec.com/pan92heuristics.html`.

[Phra02] Phrack Magazine. *Phrack*, 2002. Observed on 31 January 2003 at URL `http://www.phrack.org`.

[Powe02] Power, Richard. "2002 CSI/FBI Computer Crime and Security Survey," *Computer Security Issues & Trends*, *3(1)* 2002. Available electronically at URL `http://www.gocsi.com/press/20020407.html`.

[Ranu01] Ranum, Marcus J., "Coverage in Intrusion Detection Systems." World Wide Web, 2001. Available electronically at `http://www.nfr.com/publications/`.

[Robe02] Roberts, Paul, "Is Microsoft Serious About Security?." World Wide Web, 2002. Observed on 31 January 2003 at `http://www.pcworld.com/news/article/0,aid,105648,00.asp`.

[SANS02] SANS Institute. *Intrusion Detection FAQ*, 2002. Observed on 31 January 2003 at URL `http://www.sans.org/resources/idfaq/what_is_id.php`.

[Shel02] Shellcode Research. *Shellcode*, 2002. Observed on 31 January 2003 at URL `http://www.shellcode.com.ar/`.

[Soma00] Somayaji, Anil and Stephanie Forrest. "Automated Response Using System-Call Delays." *Proceedings of the 9th USENIX Security Symposium, August 2000*. 2000. Available electronically at URL `http://citeseer.nj.nec.com/somayaji00automated.html`.

[Sun03] Sun Microsystems. *JavaTM 2 SDK, Standard Edition Documentation*, 2003. Observed on 31 January 2003 at URL `http://java.sun.com/j2se/1.3/docs/index.html`.

[Tama03] Tama Communications Corporation. *Linux Kernel Source Tour*, 2003. Observed on 31 January 2003 at URL `http://tamacom.com/tour/linux/`.

[Teso02] TESO. *Teso*, 2002. Observed on 31 January 2003 at URL `http://www.team-teso.net`.

[Warr99] Warrender, Christina, et al. "Detecting Intrusions Using System Calls: Alternative Data Models." *Proceedings of 1999 Symposium on Security and Privacy*. 1999. Available electronically at URL `http://citeseer.nj.nec.com/christina99detecting.html`.

[Wata98] Watanabe, Yuji, et al. *Decentralized Behavior Arbitration Machanism for Autonomous Mobile Robot Using Immune Network*, 187–209. Springer-Verlag, 1998.

[Will01a] Williams, Paul D., et al. "CDIS: Towards a Computer Immune System for Detecting Network Intrusions." *Proceedings of the Fourth International Symposium on Recent Advances in Intrusion Detection, RAID 2001*. 117–133. Davis, CA: Springer-Verlag, October 2001.

[Zamb01] Zamboni, Diego. *Using Internal Sensors for Computer Intrusion Detection*. PhD dissertation, CERIAS TR 2001-42, Center for Education and Research in Information Assurance and Security, Purdue University, August 2001.

Vita

First Lt Mark G. Reith graduated from the University of Portland in 1999 with a B.S. in Computer Science and was immediately commissioned as a second lieutenant in the United States Air Force. His first assignment was to MacDill Air Force Base, FL where he worked as officer in charge of Network Security for the MacDill wide area network. Following his assignment to MacDill, he was selected to attend the Air Force Institute of Technology at Wright-Patterson Air Force Base, OH. During his three years of service, Lt Reith has been awarded the Air Force Commendation Medal and two Air Force Achievement Medals. Upon graduation Lt Reith will be assigned to the 453rd Electronic Warfare Squadron at Lackland Air Force Base in San Antonio, TX.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 074-0188*

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 SEARCHING SYSTEM CALL INFORMATION FOR CLUES: THE EFFECTS OF INTRUSIONS ON PROCESSES	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Reith, Mark, G., 1st Lieutenant, 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/03-16
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 DSN: 426-6565 Arlington VA 22203-1954 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 United States Air Force extensively uses information systems as a tool for managing and maintaining its information. The increased dependence on these systems in recent years has necessitated the need for protection from threats of information warfare and cyber terrorism. One type of protection utilizes intrusion detection systems to provide indications that intrusive behavior has occurred. Other types of protection may include packet filtering, cryptography and strong user authentication. Traditional approaches toward intrusion detection rely on features that are external to computer processes. By treating processes as black-boxes, intrusion detection systems may miss a wealth of information that could be useful for detecting intrusions. This thesis effort investigates the effectiveness of anomaly-based intrusion detection using system call information from a computational process. Previous work uses sequences of system calls to identify anomalies in processes. Instead of sequences of system calls, information associated with each system call is used to build a profile of normality that may be used to detect a process deviation. Such information includes parameters passed, results returned and the instruction pointer associated with the system call. Three methods of detecting deviations are evaluated for this problem. These include direct matching, relaxed matching and artificial immune system matching techniques. The test data used includes stack-based buffer overflows, heap-based buffer overflows and file binding race conditions. Results from this effort show that although attempted exploits were difficult to detect, certain actual exploits were easily detectable from system call information. In addition, each of the matching approaches provides some indication of anomalous behavior, however each has strengths and limitations. This effort is considered a piece of the defense-in-depth model of intrusion detection.

15. SUBJECT TERMS

Intrusion Detection, Computer Security, Immunology, Stochastic Processes, Antibodies

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON** Dr. Gregg H. Gunsch
a. REPORT U	**b. ABSTRACT** U	**c. THIS PAGE** U	UU	112	**19b. TELEPHONE NUMBER** *(Include area code)* (937) 255-6565, ext 4281; e-mail: Gregg.Gunsch@afit.edu