IOWA STATE UNIVERSITY
Digital Repository

Retrospective Theses and Dissertations

Iowa State University Capstones, Theses and Dissertations

2007

A framework for adaptive, cost-sensitive intrusion detection and response system

Natalia Stakhanova
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/rtd

Part of the Computer Sciences Commons

Recommended Citation

Stakhanova, Natalia, "A framework for adaptive, cost-sensitive intrusion detection and response system" (2007). *Retrospective Theses and Dissertations*. 15613.
https://lib.dr.iastate.edu/rtd/15613

This Dissertation is brought to you for free and open access by the Iowa State University Capstones, Theses and Dissertations at Iowa State University Digital Repository. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.

A framework for adaptive, cost-sensitive intrusion detection and response system

by

Natalia Stakhanova

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Johnny Wong
Samik Basu
Thomas Daniels
Yong Guan
Wensheng Zhang

Iowa State University

Ames, Iowa

2007

Copyright © Natalia Stakhanova, 2007. All rights reserved.

UMI Number: 3289421

UMI Microform 3289421
Copyright 2008 by ProQuest Information and Learning Company.
All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

TABLE OF CONTENTS

LIST OF TABLES iv

LIST OF FIGURES vi

ABSTRACT viii

CHAPTER 1. Introduction 1

1.1 A Framework for Preemptive, Adaptive, Cost-Sensitive Intrusion Detection and Response System 3

1.2 Contributions of the dissertation 5

1.3 Organization of the dissertation 6

CHAPTER 2. Related Work 7

2.1 Intrusion Detection 7

2.1.1 Signature-based detection 8

2.1.2 Anomaly-based detection 8

2.1.3 Specification-based detection 10

2.2 Intrusion Response 12

2.2.1 Taxonomy of Intrusion Response Systems 12

2.2.2 Existing IRS 19

2.2.3 Summary 26

CHAPTER 3. Detecting anomalous behavior of software system 30

3.1 Intrusion detection component overview 30

3.1.1 Extended Action Graph: `Exact` 31

3.1.2 Monitoring for abnormal behavior 36

3.1.3 Second-level Classifier 39

3.2 Analysis of Exact and SVM Sequences 40

3.2.1 Exact vs. Backend-SVM 40

3.2.2 Exact vs. Stand-alone SVM 41

3.3 Preliminary Simulation results 42

CHAPTER 4. Automated Intrusion Response 49

4.1 Classification of System Resources 50

4.2 Classification of Response Actions 53

4.3 Intrusion Response Selection Model 56

4.3.1 Preemption 57

4.3.2 Effect Identification 58

4.3.3 Response Strategy Selection 59

4.3.4 Adaptability 65

4.3.5 Illustrative example 67

4.3.6 Preliminary Simulation Results 68

CHAPTER 5. Implementation and experiments 74

5.1 Implementation overview 74

5.1.1 Implementation details & parameters. 77

5.2 Experiments 77

5.2.1 Building normal behavioral profile 77

CHAPTER 6. Conclusion 91

6.0.2 Future work 93

APPENDIX A. Resource/Response Correspondence Matrices 95

APPENDIX B. Response Factor Values Tables 99

BIBLIOGRAPHY 114

LIST OF TABLES

Table 2.1 List of common passive and active intrusion responses. 14
Table 2.2 Classification of the surveyed systems. 29
Table 3.1 Information on sendmail normal and intrusive trace data sets 42
Table 3.2 Maximum length of `Exact` binary vectors. 43
Table 3.3 Accuracy of classification with empty `Exact` shown in fixed-length sequences. 46
Table 3.4 Accuracy of our framework classification shown in variable-length sequences. 46
Table 4.1 Attack descriptions 68
Table 5.1 Normal profiles of the used UNIX utilities. 79
Table 5.2 The response actions considered in the experiments 83
A.1 Mapping between data segment resource and response actions 95
A.2 Mapping between stack resource and response actions 96
A.3 Mapping between file system resource and response actions 97
A.4 Mapping between network interface resource and response actions . . . 98
B.1 Impact Values by category for Potential Responses 99
B.2 Assignment of Security Policy Priorities: Public Web Server 99
B.3 Assignment of Security Policy Priorities: Classified 100
B.4 Assignment of Security Policy Priorities: Business Critical 100
B.5 Computed RF values: Public Web Server 100

B.6 Computed RF values: Classified 101

B.7 Computed RF values: Business Critical 101

LIST OF FIGURES

Figure 2.1 Taxonomy of the intrusion response systems 13

Figure 3.1 Architectural model of our framework. 31

Figure 3.2 Example of an `Exact` graph 32

Figure 3.3 Pseudo-code for `Exact` search & insert. 33

Figure 3.4 `Exact` graph after insertion 35

Figure 3.5 Pseudo-code for monitoring behavior. 37

Figure 3.6 Initialization: empty `Exact` (*decode* intrusion). 43

Figure 3.7 Initialization: `Exact` partially populated with 10% of normal sequences (*decode* intrusion). 44

Figure 3.8 Populating `Exact` (*decode* intrusion). 45

Figure 3.9 Mean running time(*decode* intrusion) 45

Figure 4.1 Algorithm for response deployment process. 56

Figure 4.2 Example demonstrating the response strategy selection process 66

Figure 4.3 Average damage reduction (fdformat attack). 68

Figure 4.4 Average damage reduction vs. error (fdformat attack). 69

Figure 4.5 Average damage reduction vs. error (ftp-write attack). 70

Figure 4.6 Average damage reduction vs. error (eject attack). 71

Figure 4.7 Average damage reduction vs. sequence length (ftp-write attack). . . . 72

Figure 5.1 Architecture of IDR system. 75

Figure 5.2 Architecture of HISC 76

Figure 5.3 The parameters of our IDR system. 76

Figure 5.4 Growth of the number of states in `Exact` graph with legal patterns . . 77

Figure 5.5 Growth of the length of binary ID associated with legal patterns. . . . 78

Figure 5.6 SVM algorithm invocation (`cat` utility). 79

Figure 5.7 Growth of the number of states in `Exact` graph with legal patterns . . 80

Figure 5.8 Growth of the length of binary ID associated with legal patterns. . . . 81

Figure 5.9 SVM algorithm invocation (`ls` utility). 82

Figure 5.10 Mean CPU running time of Unix utilities 83

Figure 5.11 System calls generated by *cat* and *mount* utilities. 85

Figure 5.12 Growth of the length of binary ID associated with normal and abnormal patterns. 86

Figure 5.13 Growth of the number of states in Exact graphs with normal and abnormal patterns. 87

Figure 5.14 System calls generated by `ls` utility 89

ABSTRACT

Intrusion detection has been at the center of intense research in the last decade owing to the rapid increase of sophisticated attacks on computer systems. Typically, intrusion detection refers to a variety of techniques for detecting attacks in the form of malicious and unauthorized activities. There are three broad categories of detection approaches: (a) *misuse-based* technique that relies on pre-specified attack signatures, (b) *anomaly-based* approach, that typically depends on normal patterns classifying any deviation from normal as malicious; and (c) *specification-based* technique that although operates in a similar fashion to anomaly-based approach, employs a model of valid program behavior in a form of specifications requiring user expertise.

When intrusive behavior is detected, it is desirable to take (evasive and/or corrective) actions to thwart attacks and ensure safety of the computing environment. Such countermeasures are referred to as intrusion response. Although the intrusion response component is often integrated with the Intrusion Detection System (IDS), it receives considerably less attention than IDS research owing to the inherent complexity in developing and deploying response in an automated fashion. As such, traditionally, triggering an intrusion response is left as part of the administrators responsibility, requiring a high-degree of expertise.

In this work we present an integrated approach to intrusion detection and response based on the technique for monitoring abnormal patterns in the program behavior. The proposed model effectively combines the advantages of anomaly-based and specification-based approaches recognizing a known behavior through the specifications of normal and abnormal patterns and classifying unknown patterns using a machine-learning algorithm. Such combination not only allows adaptation of the specification-based detection to the new patterns, but also provides a

method for automatic development of specifications.

In addition to detection, our framework incorporates preemptive response. By preemption, we imply deploying response before a monitored pattern is classified completely as an intrusion. Such response deployment is likely to stop an intrusion before it can affect the system. However, preemption also inherently suffers from false positives; i.e., responses are deployed to deter correct execution which may look intrusive in its initial phase. To reduce false positives, we have developed a multi-phase response selection and deployment mechanism based on the evaluation of the cost information of the system damage caused by potential intrusion and candidate responses.

CHAPTER 1. Introduction

The information technology (IT) has long become an internal part of our modern society. It is integrated in the infrastructure, vehicles, home appliances, daily communication etc. Although we are dependent on it at home and at workplace, we rarely realize what challenges the rapid IT development brings to the information security and what opportunities it opens for the attackers. The rapid increase of the number, sophistication and impact of computer attacks makes the computer systems unpredictable and unreliable, emphasizing the importance of timely intrusion detection. However, as the intrusion detection alone does not provide necessary assurance of the availability, reliability and security of the computer systems, it becomes necessary to ensure accurate intrusion response.

The current research state in intrusion detection & response fields. Intrusion detection has been at the center of intense research in the last decade owing to the rapid increase of sophisticated attacks on computer systems. Typically, intrusion detection refers to a variety of techniques for detecting attacks in the form of malicious and unauthorized activities. There are three broad categories of detection approaches (Sekar et al., 2002) (a) misuse-based (b) anomaly-based and (c) specification-based. Misuse-based technique relies on pre-specified attack signatures, and any execution sequence matching with a signature is flagged as abnormal. An anomaly-based approach, on the other hand, typically depends on normal patterns, and any deviation from normal is classified as malicious or faulty. Unlike misuse-based detection, anomaly-based techniques can detect previously unknown abnormalities. However, anomaly-based approaches rely on machine learning techniques which can only classify pre-specified, fixed-length behavioral patterns, and suffer from the disadvantage of a high rate of false positives (Lazarevich et al., 2003). Specification-based techniques operate in a similar

fashion to anomaly-based method detecting deviations from the specified legitimate system behavior. However, as opposed to anomaly detection specification-based approach requires user guidance in developing model of valid program behavior in a form of specifications. This process, though tedious and reliant on user-expertise, can handle variable-length sequences and is, therefore, more accurate than anomaly-based techniques.

The research in intrusion detection field for a long time has been mostly focused on anomaly-based and misuse detection (Innella, 2001; Kemmerer and Vigna, 2002). However, nowadays, as the number of attacks on computer systems increases and they become more sophisticated, the focus of research shifts in a direction of highly automatic, dynamic and adaptable intrusion detection solutions integrated with response mechanisms that are able to dynamically adapt and respond to the novel threats (Kemmerer and Vigna, 2002; Gopalakrishna et al., 2005).

While some of the existing intrusion detection systems support adaptation based on learning of new intrusion patterns through application of machine learning algorithms (Dalvi et al., 2004; Eskin et al., 2000) or based on dynamic adjustment of system resources to changing environment (Hinton et al., 1999), the adaptation in intrusion response component is often left unaddressed.

Generally, intrusion response component, although often integrated with the intrusion detection system (IDS), receives considerably less attention than research in the area of intrusion detection. One of the reason to this is the inherent complexity in developing and deploying response in an automated fashion.

Traditionally, triggering an intrusion response is left as part of the administrator's responsibility, requiring a high-degree of expertise. Although in recent years this focus has shifted, automatic intrusion response support is still very limited. Most of the automatic response systems rely on the mapping of attacks to pre-defined responses (Toth and Kruegel, 2002; Carver et al., 2000). These approaches allow the system administrators to deal with intrusions faster and more efficiently. However, they lack flexibility mainly because few of these systems take into account intrusion cost factors. Recent years have seen increased interest in developing cost-sensitive modeling of response selection (Stakhanova et al., 2007b). The primary aim for

applying such a model is to realize a balance between intrusion damage and response cost to ensure adequate response without sacrificing the normal functionality of the system under attack. However, defining accurate measurement of these cost and risk factors is one of the challenges in using these models.

Another aspect that defines the efficiency of the intrusion response is the timeliness of the response deployment. The response action in the majority of the existing response systems is conservatively invoked once the existence of intrusion is confirmed. Though this strategy reduces false-positive response, delayed response action can potentially expose systems to higher level of risk from intrusions, specifically in cases where it becomes impossible to restore systems to its pre-attacked state.

Finally, the response mechanism must allow adaptation in response selection. Specifically, if a triggered response fails to handle the corresponding intrusion multiple times then it can be inferred that the system configuration, under which the triggered response was mapped to the intrusion, has changed; this requires adaptation of response-mappings.

1.1 A Framework for Preemptive, Adaptive, Cost-Sensitive Intrusion Detection and Response System

Our research work focuses on integrated approach to dynamic intrusion detection and response combining the advantages of anomaly-based and specification-based intrusion detection techniques. Such combination allows to recognize a known behavior through the specifications of normal and abnormal patterns and to classify unknown patterns as *correct* or *incorrect* during runtime of the system using a machine-learning algorithm. Thus, the proposed approach not only provides adaptation of the specification-based detection to the new patterns, but also allows automatic development of specifications.

To efficiently maintain the results of classification, we propose a novel structure *EXtended ACTion graph* (`Exact`) that appropriately combines multiple sequences classified by machine learning technique into variable-length patterns and memorizes them for future reference. In our framework, we have two `Exact` graphs: one for storing normal patterns and the other for

abnormal patterns. Sequences are classified using `Exact`, and the machine learning algorithm is only invoked if necessary (Stakhanova et al., 2006).

The proposed `Exact` structure allows easy integration of intrusion detection process with the response mechanism. In our work we focused on an intrusion response model which is

1. *automated*: deploying responses with little or no user-guidance.

2. *cost-sensitive*: assessing the risk and cost of (not) responding.

3. *preemptive*: triggering responses before the anticipated attack completes.

4. *adaptive*: updating the responses on-the-fly on the basis of their success and failure in thwarting previously detected intrusions.

We approach the modeling of intrusion response from the integrated perspective of the attack and the system resources views. On the one hand, the attack perspective allows locally customize a response strategy to a specific attack trace at different attack steps. On the other hand, we focus on an intrusion response from a global view of the attack impact on the system resources.

To achieve this synergy of two perspectives, we develop a taxonomy of the system resources that might potentially become targets during an attack and the responses that can be effectively deployed to either counter possible attacks on these resources or defend system services and regain secure system state. We also build a mapping between system resources, the corresponding responses and specific attack patterns.

The association between system resources, response actions and intrusive patterns is recorded in the abnormal graph; this is done by associating the start state of the pattern to the resource(s) it might affect. Monitored system activity is then continuously matched with patterns in normal and abnormal graphs. Whenever a monitored sequence matches with a prefix of any intrusive behavior (in abnormal graph), our algorithm decides whether or not to respond immediately. Note that, the monitored activity is not yet classified as intrusive – it just matches with a prefix of one or more intrusive behavior. If a response action is invoked at this stage, we are forcing *preemptive response* to a *possible* intrusive behavior. However, we do not allow

blind preemption, instead preemption depends on the probability of the potential intrusion. If the prefix-pattern matched with the monitored sequence exhibits high probability (greater than a pre-specified threshold or tolerance factor) of ending up in an intrusive behavior then our algorithm decides to deploy a response action.

As a number of intrusive patterns can have the same prefix, there can be multiple candidate response actions that can be deployed preemptively. Therefore, it is required to identify the response that will have the *least negative affect* on the system. This is done by evaluating costs and benefits of the available responses based on the *utility theory* (Coyle, 1972), that focuses on providing maximum benefit at the lowest risk. In particular, we take into consideration the past effectiveness of the possible responses and their respective severity. By severity, we imply the negative impact the response can have on legitimate users.

If intrusive behavior continues after the preemptive response was deployed, the triggered response's effectiveness factor is lowered to *adjust response selection* in the future use of this response action. This response selection and deployment are performed automatically without any user intervention which allows fast containment of the intrusion and thus makes system defense more effective.

1.2 Contributions of the dissertation

We see the following contributions of our work:

1. Development of an adaptive and proactive specification-guided anomaly based intrusion detection enhanced with automatic development of specifications.

2. Development of intrusion response system with the following characteristics:

    (a) response actions are triggered automatically.

    (b) response actions are triggered proactively.

    (c) response selection is based on probabilistic cost-benefit analysis.

    (d) response selection mechanism is adaptive.

3. Demonstration of the practical applicability of the proposed technique by
   (a) developing a prototype of the proposed intrusion detection and response system
   (b) performing experiments that demonstrate system effectiveness in detecting known and novel attacks and in handling these intrusions
   (c) performing experiments that test system performance, specifically, the overhead that our model brings to the system while monitoring its behavior

1.3 Organization of the dissertation

The rest of the thesis is organized as follows:

1. In Chapter 2 we discuss the related work in intrusion detection and present a taxonomy of intrusion response systems, together with a review of current trends in intrusion response research.

2. In Chapter 3 we present an overview of the proposed model for monitoring program behavior and discuss its components: `Exact` graph structure and machine learning-based classification followed by the simulation results.

3. In Chapter 4 we provide the classifications of system resources, intrusion responses and the correspondence between them. In this chapter we also explain the details of our cost-sensitive model for intrusion response selection that incorporates preemptive deployment of the response and the adaptive mechanism for adjusting response deployment at run-time of the system.

4. In Chapter 5 we provide an implementation details together with the experiments results.

5. In Chapter 6 we summarize our results and discuss our future work.

CHAPTER 2. Related Work

2.1 Intrusion Detection

Intrusion detection has been the center of intense research in the last decade owing to the rapid increase of sophisticated attacks on computer systems. It typically refers to a variety of techniques for detecting attacks in the form of malicious and unauthorized activity.

In general, the intrusion detection techniques can be broadly classified into three classes (Sekar et al., 2002)

- **Signature-based (misuse) detection** relies on pre-specified *attack signatures*, rules or events that precisely describe intrusion behavior. Any execution sequence matching attack signature is marked as abnormal. While signature-based approach provides low false alarm rate, it cannot identify novel intrusions, attacks that do not have pre-specified signatures.

- **Anomaly-based detection** is based on the model of normal system behavior usually developed using statistical or machine learning techniques. During the detection any deviation from this model is considered abnormal and classified as potential intrusion. Unlike signature-based technique, anomaly detection can recognize previously unknown abnormalities.

- **Specification-based detection** operates in a similar fashion to anomaly-based method detecting deviations from the specified legitimate system behavior. However, as opposed to anomaly detection specification-based approach requires user guidance in developing model of valid program behavior in a form of specifications.

2.1.1 Signature-based detection

In the past two decades a number of misuse techniques have been proposed. Among these are methods based on state-transition analysis of anomalous system behavior (d'Auriol and Surapaneni, 2004; Ilgun, 1993; Ilgun et al., 1995), rule-based expert systems (Garvey and Lunt, 1991; Habra et al., 1992; Jean-Philippe Pouzol, 2002; Sebring et al., 1988) and attack graph-based approaches (Kumar, 1995; Kumar and Spafford, 1994; Lin et al., 1998; Sheyner et al., 2002; Staniford-Chen et al., 1996).

In recent years, several methods have been proposed to represent intrusion signatures through attack graphs which can be constructed from the alerts reported by intrusion detection system (Ning et al., 2004; Noel et al., 2005, 2004; Sheyner et al., 2002). These graphs precisely model attack paths in the network through nodes representing host vulnerabilities and edges showing connectivity between these hosts (Sheyner et al., 2002). While attack graphs are exhaustive and precise, their manual construction is tedious and often error-prone. Recently, several projects have focused on automatic generation of such graphs (Sheyner et al., 2002; Swiler et al., 2001). Another concern related to attack graphs is their scalability. While it became possible to build attack graphs for large networks using automatic tools, it is still quite difficult to manage their complexity. Several visualization techniques have been proposed to cope with this problem (Noel et al., 2005; Noel and Jajodia, 2004).

While signature-based approaches provide low false alarm rate, they cannot identify novel intrusions, i.e. attacks that do not have a pre-specified signatures.

2.1.2 Anomaly-based detection

An anomaly-based approach relies on a model of normal system behavior. During the detection any significant deviations from this model are considered abnormal and classified as potential intrusions. Unlike signature-based techniques, anomaly detection can recognize previously unknown abnormalities. However, since the detection process relies on potentially incomplete model of legitimate behavior, anomaly detection suffers from the high false positive rate.

A model of normal system behavior is typically developed through statistical or machine learning techniques (Axelsson, 2000). Statistical methods build normal model by collecting statistical values from data (such as mean, standard deviation, time series behavior etc.) (Smaha, 1988; Lunt et al., 1992; Huang et al., 2003). One disadvantage of these methods is the necessity to select a threshold distinguishing normal and abnormal values which is not a trivial task.

Machine learning techniques usually construct model by analyzing normal system behavior for potential patterns (Debar et al., 1992; Lane and Brodley, 1999). In this case, the constructed model is highly dependent on the underlying data. Since system behavior is sequential in nature, the common data modeling technique is a sliding window approach that breaks data stream into window-length sequences. Most methods employ fixed-length window size (Forrest et al., 1996; Hofmeyr et al., 1998; Wang et al., 2004). An overview and comparison of such methods is given in (Warrender et al., 1999). However, the common challenge in these approaches is the selection of optimal window size that provides the best anomaly detection performance (Tan and Maxion, 2002). While window size can potentially affect the detection process (Hofmeyr et al., 1998), no algorithm for selection of optimal length was developed.

Debar (Debar et al., 1998) proposed generation of variable-length sequences based on suffix trees augmented with a number of occurrences of each subsequence. The resulting tree is pruned based on the frequency of patterns to reduce a number of false positives during detection process. Thus the final tree contains only the most frequent patterns, and therefore prone to high false positive rate.

Similar approach was proposed by Marceau (Marceau, 2000). However, the suffix tree was employed as the underlying structure for constructing finite state machine with states representing predictive sequences of variable length. Kosoresow and Hofmeyr (Kosoresow and Hofmeyr, 1997) also employed finite automaton based on variable-length patterns for detection process. As the automaton construction was done manually it does not seem scalable for real systems.

Eskin et al. (Eskin et al., 2001) proposed an alternative algorithm for determining optimal window size depending on the data context motivating the approach by the fact that different

window sizes might be considered as optimal at different points in the process. Their approach is based on Sparse Markov Transducers (SMTs), extension of probabilistic suffix trees. SMTs allow to consider a mixture of possible trees and estimate the best tree for the given data. The topology of the final tree defines the context dependent window sizes and wild card placement. While proposed approach provides a good prediction for variable-length patterns in a particular data set, it is static in its nature as the approach does not allow to update the prediction tree as more system data becomes available.

2.1.3 Specification-based detection

Specification-based detection aims to overcome difficulties of both signature-based and anomaly-based approaches. Specification-based technique, though tedious and reliant on user-expertise, is more accurate than anomaly detection. At the same time, it does not rely on the knowledge of all possible system vulnerabilities as signature-based technique, and thus able to detect novel attacks.

One of the first specification-based models were introduced by Ko (Ko et al., 1994, 1997; Ko, 2000). Ko's initial work (Ko et al., 1994) has been focused on the monitoring of the privileged programs executions using audit trails. While the number of potential program vulnerabilities is unknown, the intended program behavior is limited and can be specified in a concise fashion in a form of specifications. Ko also introduced a specification language based on predicate logic and *parallel environment grammars (PE-grammars)*. While his initial work focused on host-based intrusion detection, he later applied the proposed approach to distributed scenario (Ko et al., 1997).

Another specification-based approach focused on monitoring program behavior has been proposed by Sekar (Sekar et al., 1998; Sekar and Uppuluri, 1999). In addition to the monitoring program executions through system calls, this technique aims to enforce specified legal behavior through *isolation technique*. It allows to isolate compromised process by preventing its execution operations that can potentially damage the system. Examples of such prevention are return of error code for suspicious system call or restriction of file access.

While the above techniques are mostly based on the monitoring of the program behavior, SHIM (System Health and Intrusion Monitoring) approach (Ko et al., 2001) employs specifications in form of constraints that describe valid system behavior. Employing hierarchy of such constraints and checking their validity during run-time, although does not directly detect a potential attack, catches its manifestation.

One of the major downsides of the specification-based approach is the necessity to develop the system specifications manually. As this process is time-consuming and error-prone, automatic generation of specifications is highly beneficial. As such, Ko(Ko, 2000) presented a machine learning approach for developing security specifications automatically. He built an induction engine based on *Inductive Logic Programming*(IPL) method that generates program behavioral specifications at the system call level.

Wagner and Dean(Wagner and Dean, 2001) employed a static analysis to automatically derive program specifications. They proposed three methods for generating the specifications: *callgraph model*, based on control-flow analysis of code, *abstract stack model* represented as pushdown automaton and *digraph model* based on all possible 2-gram sequences of system calls derived from control-flow graph.

While ensuring completeness of the developed specifications is a common difficulty of the specification-based models, only a few approaches have attempted to address this problem. Song at el. (Song et al., 2003) proposed a formal framework based on ACL2 for analysis and verification of specifications. Since the system specifications are developed based on certain assumptions, deploying a mechanism to secure these assumptions will improve the security of the system.

Our approach was inspired by the specification-based anomaly detection technique proposed by Sekar et al. (Sekar et al., 2002). Their work aimed at augmenting machine learning techniques with high-level specifications to achieve a high degree of precision in detecting anomalies in software. The authors showed that the sliding window technique (Forrest et al., 1996) using a machine learning algorithm may be excessively error-prone due to its inability to classify sequences of varying length. They thus manually developed high-level specifications

(as finite state machines) of software systems and annotated them using statistical information learnt via machine learning technique. During detection process, they computed specified statistics and compared them for deviations from the values indicated in specifications.

Several works have been focused on the specification languages (Uppuluri and Sekar, 2000; AsmL, 2007). The Behavior Monitoring Specification Language (BSML) (Uppuluri and Sekar, 2000) was designed to capture event-based security properties of the program through the system calls and their arguments. Another language, AsmL (Abstract State Machine Language) (AsmL, 2007) was developed by Microsoft Research for software specification and verification. It is an executable specification language that models system as a series of states. One of the recent approaches used AsmL to describe attack scenarios in misuse intrusion detection (Raihan and Zulkernine, 2005).

More recently, several techniques applied specification-based approach to detect attacks against AODV (Tseng et al., 2003), OLSR (Tseng et al., 2005) and cryptographic protocols (Joglekar and Tate, 2005).

2.2 Intrusion Response

Intrusion response generally refers to evasive and/or corrective actions taken in the event of detected intrusive behavior to thwart attacks and ensure safety of the computing environment.

2.2.1 Taxonomy of Intrusion Response Systems

To organize existing research efforts in the field of intrusion response we developed a taxonomy of IRS. The proposed taxonomy is given in Figure 2.1. In the reminder of this section we provide details on each of the categories in the given classification. Intrusion response systems can be classified according to the following characteristics:

Activity of triggered response.

- **Passive:** Passive response systems do not attempt to minimize damage already caused by the attack or prevent further attacks. Their main goal is to notify the authority

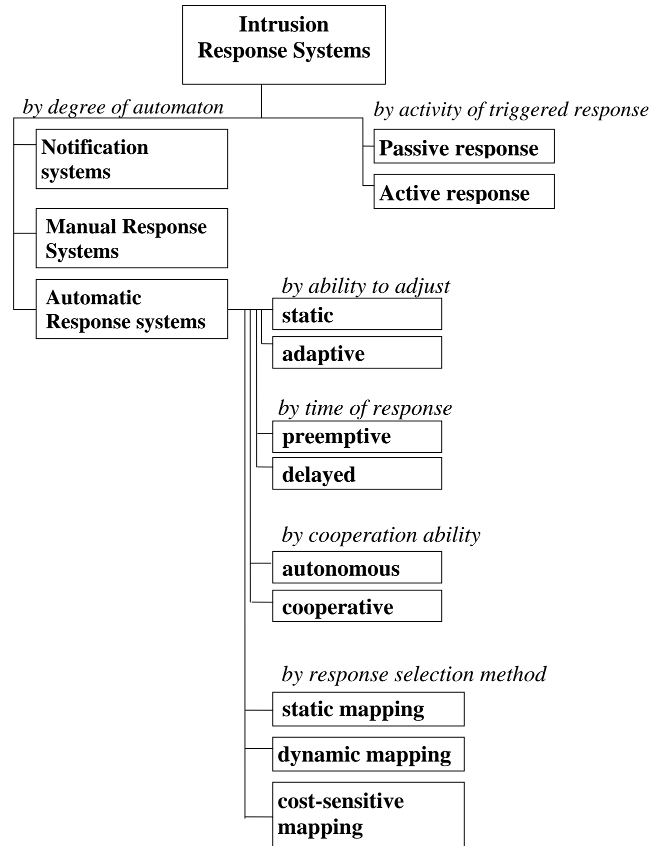


Figure 2.1 Taxonomy of the intrusion response systems

and/or provide attack information.

- **Active:** As opposed to passive systems, active systems aim to minimize the damage done by the attacker and/or attempt to locate or harm the attacker.

Majority of the existing intrusion detection systems provide passive response. Among 20 IDS evaluated by Axelsson (Axelsson, 2000), 17 systems supported passive response while only 3 systems were designed to mitigate the damage or harm the attacker. Table 2.1 gives an overview of the passive and active approaches used in the existing response systems.

Passive	Active
Administrator notification:	Host-based response actions:
generate alarm	deny full/selective access to file
(through email, online/pager notification, etc.)	delete tampered with file
generate report	allow to operate on fake file
(can contain information about one intrusion	restore tampered with file from backup
such as attack target, criticality, time, source IP,	restrict user activity
user account, description of suspicious packets, etc.	disable user account
as well as intrusion statistics for a period of time)	shutdown compromised service/host
such as number of alarms from each IDS,)	restart suspicious process
attack targets grouped by IP etc)	terminate suspicious process
enable additional IDS	disable compromised services
enable local/remote/network activity logging	abort suspicious system calls
enable intrusion analysis tools	delay suspicious system calls
backup tampered with files	
trace connection for information gathering purposes	Network-based response actions:
	enable/disable additional firewall rules
	restart targeted system
	block suspicious incoming/outgoing
	network connection
	block ports/IP addresses
	trace connection to perform attacker
	isolation/quarantine[1]
	create remote decoy[1]

Table 2.1 List of common passive and active intrusion responses.

Level of automation. The classification according to the level of automation has been presented in early works by several authors (Toth and Kruegel, 2002; Carver et al., 2000; Ragsdale et al., 2000). However, employing only these categories gives a very broad view of the response systems and hence does not provide enough information about the existing research efforts. The taxonomy presented here, on this categorization, also includes additional principles that emphasize differences between various existing approaches.

- **Notification systems:** Notification systems mainly provide information about the intrusion which is then used by system administrator to select an intrusion response. Majority of the existing IDSs provide notification response mechanism.

- **Manual response systems:** Manual response system provides higher degree automation than notification-only systems and allows system administrator to launch an action from a predetermined set of responses based on the reported attack information.

[1]Borrowed from (Wang et al., 2001a)

- **Automatic response systems:** As opposed to manual and notification approaches, automatic response systems provide immediate response to the intrusion through automated decision making process. Although intrusion detection systems are greatly automated nowadays, automatic intrusion response support is still very limited.

2.2.1.1 Automatic response systems

Response ability to adjust.

- **Static:** Majority of the IRS are static as the response selection mechanism remains the same during the attack period. These systems can be periodically upgraded by the administrator, however, such support is manual and often delayed till the moment when considerable amount of intrusions exposes the inadequacy of current response mechanism. Although this approach takes a conservative view of the system and environment, it is simple and easy to maintain.

- **Adaptive:** The adaptability of the response is an ability of system to dynamically adjust response selection to the changing environment during the attack time. Adaptation capability can be represented in several ways including (a) adjustment of system resources devoted to intrusion response such as activation of additional IDS, or (b) consideration of success and failure of responses previously made by the system. The latter can refer to both detection and response mechanisms. Failure of the response can be due to the mistake of IDS that falsely flagged normal activity as intrusion or due to the mistake of IRS that triggered an inappropriate response.

Time instance of the response.

- **Proactive (preemptive):** Proactive response system allows to foresee the incoming intrusion before the attack has affected the resource. Such prediction is generally hard and often relies on the probability measures and analysis of current user/system behavior. Proactiveness of the response also requires that the detection and response mechanisms

are tightly-coupled such that responses can be fired as soon as a likelihood of attack is identified. Although proactive detection of the attack and early response is a desired feature, it is often hard to guarantee 100% correctness of the triggered action. The trade-off between the correctness of the attack detection and timely response to the possible attack is an inherent characteristic of intrusion response systems.

- **Delayed:** The response action is delayed until the attack has been confirmed. Such assurance may be provided through the confidence metrics of IDS or full match of the intrusive trace with an existing attack signature. Although, majority of the existing systems use delayed response approach, it may not be suitable for safety-critical systems. For example, for systems relying on checkpoints as fault tolerance mechanism, a delayed response might lead to inability of a system to roll back to the safe state.

  Generally, the delayed response leaves more time to the attacker, consequently allowing more damage to occur and therefore putting the greater burden of system recovery on the system administrator.

The proactive and delayed intrusion response has also been considered by several authors as *incident prevention* and *intrusion handling* of intrusion response (Fisch, 1996; Bishop, 2003) respectively. Proactive response is merely an incident prevention that takes place before attack has succeeded, while delayed response is intrusion handling that is performed after the intrusion and includes actions to restore system state. While these two steps should be performed sequentially to provide full system defense and repair, often systems fall back into one of approaches.

Cooperation capabilities.

- **Autonomous:** Autonomous response systems handle intrusion independently at the level it was detected. As such, a host-based IDS detecting an intrusion on a single machine will trigger a local response action such as terminating a process, shutting down the host, etc.

- **Cooperative:** Cooperative response systems refer to a set of response systems that combine efforts to respond to an intrusion. Cooperative system can consist of several autonomous systems that are capable of detecting and responding to intrusions locally, however the final or even additional response strategy is determined and applied globally. Often network IRS are built in such cooperative manner. It allows to achieve better performance in terms of speed of the response and volume of the contained damage. Although cooperative systems provide more effective response than autonomous systems alone, they are also more complex requiring strong coordination and communication among their components.

Response selection mechanism. A step into distinguishing various response selection principles was taken by Toth et al. (Toth and Kruegel, 2002; Toth, 2003). The authors noted that majority of the existing approaches use static mapping tables or rules based dynamic engines which we define as static and dynamic mapping approaches.

- **Static mapping:** Static mapping systems are essentially automated manual response systems that map an alert to a predefined response. For example, alert about attack on a host can trigger dropping incoming/outgoing network packets. These systems are easy to build and maintain. However, they are also predictable and therefore, vulnerable to intrusions, in particular, denial-of-service attacks. Another weakness of the static mapping systems is their inability to take into account the current state of the whole system. In static mapping systems the triggered response actions present isolated efforts to mitigate the attacks without considering current condition and the impact on other services and system in general. Additionally, as it has also been noted by Toth (Toth and Kruegel, 2002), this approach seems to be infeasible for large systems where the volume of threat scenarios to be analyzed and the constant changes in system policies make the process of building such decision tables cumbersome and prone to errors.

- **Dynamic mapping:** Dynamic response mapping systems are more advanced than static mapping systems as the response selection is based on the certain attack metrics

(confidence, severity of attack, etc). In the dynamic mapping setting an intrusion alert is associated with a set of response actions. The exact action is chosen in real-time based on the characteristics of the attack. Generally the selection mechanism for an alert can be presented by a set of "if-then" statements. For example,

if unauthorized user gains access to the password file then
if confidence of attack is greater than 50% then
disable user account and restore password file from backup
if confidence of attack is smaller than 50% then
give a fake password file

Generally, by adjusting attack metrics we can provide more flexibility in intrusion response selection. For example, attacks with low confidence and severity level can be ignored; moderately severe intrusions with low certainty can be traced while high severity attacks can be responded with appropriate actions. Although this approach can still be potentially exploited by an adversary, it provides much more fine-grained control in response to an attack.

- **Cost-sensitive:** Cost-sensitive response systems are the only response systems that attempt to balance intrusion damage and response cost. The optimal response is determined based on the cost-sensitive model that incorporates several cost and risk factors. Usually these factors are divided into factors related to the intrusion such as damage cost and factors characterizing response part such as response action cost. Accurate measurement of these factors is one of the challenges in using these cost models. Numeric values such as monetary values, probabilistic measurement or percentages that correspond to some objective metrics are not always suitable, as more effective solution based on relative measurements can be applied (Peltier, 2001). The relative measurements can be contracted based on organization security policies, risk factors, etc (Lee et al., 2000). One of the downsides of this approach is the necessity to update cost factor values with time. In most cases it is done manually which also puts additional burden on the system administrator.

2.2.2 Existing IRS

In this section we will discuss the existing intrusion response systems in relation to the proposed taxonomy.

Static vs. Adaptive. The response models proposed by Foo et al. (Foo et al., 2005) and Carver et al. (Carver and Pooch, 2000; Carver et al., 2000; Ragsdale et al., 2000) are examples of an adaptive approach. AAIRS, due to (Carver and Pooch, 2000; Carver et al., 2000; Ragsdale et al., 2000), provides adaptation through confidence metric associated with each IDS and through success metric corresponding to the response component of the system. The confidence metric indicates the rate of false positive alarms to correct number of intrusions generated by each IDS employed by the system. Similarly, the success metric indicates response actions and response plans that were more successful in the past.

Similar adaptation concept based on the feedback is presented in ADEPTS (Foo et al., 2005). In this case, effectiveness index, a metric showing effectiveness of a response action against particular attack, is decreased if the action fails. While ADEPTS supports automatic update of the response effectiveness metric, AAIRS requires system administrator intervention after each incident.

Unlike these two solutions, other models considered in Table 2.2 offer no adaptation support in response mechanism.

Proactive vs. Delayed. Among the existing response systems presented in the literature, the majority fall into delayed response category. One of the solutions in these models is suspension of the suspicious processes until the intrusion has been confirmed (Balepin et al., 2003; Somayaji and Forrest, 2000). Such suspension can be temporal until further response strategy is formulated (Balepin et al., 2003) or permanent until the system decides to abort delayed program (Somayaji and Forrest, 2000). Another approach in delayed response is allowing the execution of the suspicious behavior until the observed pattern has matched an intrusive signature (White et al., 1996; Wang et al., 2001a).

A rare example presented in recent work by Foo et al. (Foo et al., 2005) investigates a proactive approach to response deployment. The proposed system employs *an intrusion graph* (I-Graph) to model attack goals and consequently to determine possible spread of the intrusion. The mechanism maps alarms provided by the involved IDS to I-Graph nodes and estimates the likelihood of the attack spread based on the alarm confidence values. Finally, appropriate response actions are deployed targeting identified attack goals.

Another proactive handling of response was recently proposed by Locasto et al. (Locasto et al., 2005). FLIPS, intrusion prevention system, is based on STEM technique (Sidiroglou et al., 2005) that allows to create unique environment for emulation of selected application pieces prior to their real execution. Using this approach for code injected attacks, malicious code can be recognized within a few bytes and prevented from execution.

The Cooperating Security Manager system (CSM) proposed by White et al. (White et al., 1996), although not specifically designed to be proactive, can yield proactive reaction to intrusive behavior in certain cases. This is a distributed approach that combines individual hosts equipped with CSM. While each host performs a local intrusion detection, it is also responsible for notifying other CSMs about suspicious activity. Clearly, instead of waiting for intrusive activity from a user, notified host can take a proactive action to prevent it. An example of such situation is when attacker attempts to gain unauthorized access to an account by trying different passwords. However, instead of checking all possible passwords on one machine, attacker moves to a different host after each failed attempt. While several unsuccessful logins can raise an alarm, single attempt will not be significant enough to be flagged as suspicious. Therefore, reporting such activity to other CSM hosts allows to detect this attack.

Autonomous vs. Cooperative. There are several examples of the cooperative response systems in the published literature. One such example, Survivable Autonomic Response Architecture (SARA) (Lewandowski et al., 2001) was developed as unified approach to coordinate fast automatic response. It consists of several components that function as sensors (information gathering), detectors (analysis of sensor data), arbitrators (selection of appropriate response actions) and responders (implementation of response). These components can be arranged

among participating machines in a manner that provides the strongest defense. Thus, each host of the system can be equipped with arbitrator which can provide local intrusion response and at the same time participate in a global response selection strategy.

Another cooperative model is EMERALD - a distributed framework for network monitoring, intrusion detection and automated response proposed by Porras and Neumann (Porras and Neumann, 1997). The framework introduces a layered approach allowing to deploy independent monitors through different abstract layers of the network. The response component of the framework is represented by the *resolver* that is responsible for analyzing attack reports and coordinating response efforts. While *resolvers* are responsible for response strategy on their local level, they are also able to communicate with resolvers at other EMERALD layers, participating in global response selection.

The Cooperative Intrusion Traceback and Response Architecture (CITRA) presented in (Schnackenberg et al., 2001) provides an example of cooperative agent-based system. This architecture utilizes neighborhood structure where the information about detected intrusion is propagated back through the neighborhood to the source of the attack and submitted to the centralized authority. The centralized authority, referred to as Discovery Coordinator, finally determines an optimal system response. While the Discovery Coordinator is responsible for coordinating global response, local CITRA agents can issue a local response action on a local intrusion detection report.

All of the cooperative approaches to response selection and deployment tend to be distributed network-oriented systems. While CSM system (White et al., 1996), discussed in the previous section, presents an example of autonomous response system, it is a distributed IDS. CSM system allows hosts to share information and detect intrusive user activity in a cooperative manner, however the response actions are determined and deployed by each machine locally.

Other examples of autonomous response system include (Somayaji and Forrest, 2000; Bowen et al., 2000; Uppuluri and Sekar, 2000). These are host-based systems specifically oriented to handle local intrusion detection and response.

Static mapping vs. Dynamic mapping. Most tracing techniques fall into static mapping category and automatically respond to an intrusion by tracing it back to the source and applying predetermined response actions (Wang et al., 2001b; Schnackenberg et al., 2000). Although automated, these approaches have a spirit of notification intrusion response systems as they mainly report about the intrusion source.

Several recent tracing mechanisms take one step further offering a combination of static and dynamic mapping techniques (Wang et al., 2001a; Schnackenberg et al., 2001). TBAIR (Wang et al., 2001a) framework suggests to trace the intrusion back to the source host and dynamically select the suitable response such as remote blocking of the intruder, isolation of the contaminated hosts, etc.

Similar approach was taken by CITRA (Schnackenberg et al., 2001). This framework integrates network-based intrusion detection, security management systems and network infrastructure (firewalls, routers) to detect the intrusion, trace it back to the source and coordinate local response actions based on the attack report. The response mechanism is based on two factors: certainty and severity of the intrusion. While certainty represents the likelihood that reported event is an intrusion, severity defines potential damage to the system and is mainly based on the policy of the particular site. Depending on the reported certainty and severity values, a response action is chosen from a predetermined set.

While these dynamic techniques rely on the underlying predefined set of responses, as opposed to static mapping techniques, the actual action is determined dynamically based on additional factors specific to the current intrusion attempt (intrusion confidence and severity).

Based on agent architecture SoSMART approach (Musman and Flesher, 2000) is an example of statically mapped response selection system. User-designed incident cases mapped to the appropriate responses present an available set of response actions. In addition to this response decision set, SoSMART model employs a case-base reasoning (CBR) as an adaptation mechanism that matches current system state to the situations previously identified as intrusive. Based on the past experience an additional set of responses can be selected and deployed. Dynamic addition of the new cases allows CBR system to evolve over time.

The next two discussed approaches also offer static mapping response selection mechanism as they rely on the deployment of the prespecified response actions. Authors of (Bowen et al., 2000; Uppuluri and Sekar, 2000) proposed an approach to intrusion detection and response based on the specifications of normal behavior expressed in BMSL (Behavioral Monitoring Specification Language). BMSL specifies system behavior in a finite state machine automata fashion and augments each intrusion specification path with a response action. This action can be represented by invocation of a response function, assignment to a state variable or a set of rules for process isolation.

The pH system developed by Somayaji and Forrest (Somayaji and Forrest, 2000) is an intrusion detection and response system. Its detection component is based on the normal behavioral profile of the system consisting of N-gram sequences of system calls. Sequences of calls deviating from the normal behavior are considered anomalous and can be either aborted or delayed. Although, these two response actions are simple and computationally not expensive, authors acknowledge that they are not suitable for all applications and additional response might need to be considered.

The detection component of this approach is closely related to our model. Although both works automatically develop normal behavior of the system, our approach allows more flexibility in the detection part, representing specifications of both normal and anomalous behavior in terms variable-length patterns, and supports easy extension to the sophisticated response mechanism.

Dynamic mapping vs. Cost-sensitive. CSM (White et al., 1996) and EMERALD (Porras and Neumann, 1997) are dynamic mapping systems. In both approaches the selection of the response strategy is based on confidence information about detected intrusive behavior produced by the detection component and severity metrics associated with an attack.

Another dynamic mapping technique specifically aimed at intrusion damage control and assessment, DC&A, is proposed by Fisch (Fisch, 1996). DC&A tool contains two primary components: *damage control processor* responsible for actions necessary to reduce or control the damage done by the intruder while the intrusion is still in progress and *damage assessment*

processor that performs post-attack measures aimed at system recovery. A specific response action to an intrusion is selected by damage control unit based on a suspicion level of user's activity provided by IDS and from the responses available for the given suspicion level. If user's suspicion level increases with time a different response action can be later selected. After intruder leaves the system, damage assessment processor will determine necessary actions to restore original system state based on final suspicion level associated with the intruder. For example, the assessment procedure can include analysis of log files followed by replacement of the stolen files from backup storage.

One of the most complex dynamic mapping approaches is Adaptive, Agent-based Intrusion response system based on agent architecture (AAIRS) (Carver and Pooch, 2000; Carver et al., 2000; Ragsdale et al., 2000). Framework agents represent the layers of the response process. Intrusion alarms are first processed by the Master analysis agent which computes confidence level and classifies the attack as new or ongoing. This classification is mainly based on the preset decision tables. This information is then passed to the Analysis agent which generates action plan based on the response taxonomy. Authors proposed 6-dimentional taxonomy (Carver and Pooch, 2000): timing, type of attack, type of attacker, degree of suspicion, attack implications and environmental constraints. Finally, the Tactics agent decomposes the response plan into specific actions and invokes the appropriate components of the response toolkit. This work mainly presents a foundation for intrusion response system as no specific techniques or algorithms necessary for AAIRS are provided.

Compared to the amount of work published on static and dynamic response selection mechanisms, the category of cost-sensitive selection is relatively small.

The approach to intrusion response proposed by Lee at al. (Lee et al., 2000) is based on a cost-sensitive modeling of the intrusion detection and response. Three cost factors were identified: *operational cost* that includes the cost of processing and analyzing data for detecting intrusion, *damage cost* that assesses the amount of damage that could potentially be caused by attack and *response cost* that characterizes the operational cost of reaction to intrusion. These factors present the foundation of intrusion cost model, i.e total expected cost of intrusion

detection, and consequently provides a basis for a selection of an appropriate response.

Models proposed by Toth and Kruegel (Toth and Kruegel, 2002) and Balepin el al. (Balepin et al., 2003) not only consider costs and benefits of the response actions, but also attempt to model dependencies between services in the system. Such modeling reveals priorities in response targets and evaluates the impact of different response strategies on dependent services and system.

The approach proposed by Toth and Kruegel (Toth and Kruegel, 2002) is a network-based response mechanism that builds dependency tree of the resources on the network. The proposed algorithm for optimal response selection takes into account *a penalty cost* of a resource being unavailable and *capability of a resource* that indicates the resource performance if specified response strategy is triggered, compared to the situation when all resources are available. Clearly, the set of response actions with the least negative impact on the system (lowest penalty cost) is chosen to be applied in response to the detected intrusion.

Similar approach, based on host-intrusion detection and response, was proposed by Balepin el al. (Balepin et al., 2003). In this system, local resource hierarchy is represented by a directed graph. Nodes of the graph are specific system resources and graph edges represent dependencies between them. Each node is associated with a list of response actions that can be applied to restore working state of resource in case of an attack. A particular response for a node is selected based on *the cost of the response action* (sum of the resources that will be affected by the response action), *the benefit of the response* (sum of the nodes, previously affected by intrusion and restored to working state) and *the cost of the node or resource.*

Graph-based approach called ADEPTS, Adaptive Intrusion Response using Attack Graphs, as discussed in the previous section, is proposed by Foo et al. (Foo et al., 2005). Modeling intrusion using graph approach allows to identify possible attack targets and consequently shows objectives of suitable responses. The response actions for the affected nodes in the graph are selected based on the effectiveness of this response to the particular attack in the past, the disruptiveness of the response to the legitimate users and confidence level that indicates the probability that real intrusion is taking place.

This work is the most closely related to our approach to intrusion response. Although the approaches employ similar metrics, our schema allows more flexibility in determining the response action. ADEPTS although calculates the attack confidence, does not use it in response selection. This might lead to a situation when responses for nodes that are less likely to be reached are more severe than the responses for the nodes currently under attack. ADEPTS also relies on the I-graph in determining the intrusion spread, however, it does not allow dynamic additions of the intrusion patterns which makes I-graph static and the system vulnerable to the attacks not present in the graph. Though ADEPTS allows to trigger response preemptively, it does not take any follow-up steps to contain the intrusion if the selected response failed. In contrast, we continue a response selection algorithm until no traces of the intrusion can be detected.

2.2.3 Summary

An overview of the research and development of intrusion response system in the last decade is given in Table 2.2 and can be summarized as follows:

- Recent years have seen increased interest in developing cost-sensitive modeling of response selection. The primary aim for applying such a model is to ensure adequate response without sacrificing the normal functionality of the system under attack. Our survey shows that though a number of response frameworks often offer facilities responsible for these mechanisms, very few works provide the detailed algorithms.

- In terms of response-deployment time, majority of proposed frameworks conservatively invoke responses once the existence of intrusion is a certainty. Though this reduces false-positive response, delayed responses can potentially expose systems to higher level of risk from intrusions with no mechanism for restoring system to its pre-attacked state. Therefore, a few research effort developed proactive response mechanisms to enable early response to intrusions, notably, most of them appeared just recently. It should be also mentioned that developing an optimal proactive response mechanism is difficult as it can prohibitively increase false positives.

- Another elusive characteristic of response systems is adaptiveness. It is a powerful feature required to ensure normal functionality while still providing effective defense against intrusive behavior, and to automatically deploy different responses on the basis of the current system state. At the same time, adaptiveness brings system into the higher level of complexity and poses new questions such as "How can we automatically classify a response as success or failure? If the response has failed how can we determine whether the system state changed due to triggered (failed) response or continuance of the attack? How can we separate the beginning of new intrusion and continuance of the old attack?" As such, very few of the existing response mechanisms incorporate adaptation.

- Finally, we have seen the presence of both cooperative and autonomous response systems. Typically, host-based intrusion response techniques are autonomous while cooperative methods are deployed in network IDS. Although techniques presented here are existing research efforts, several commercial products with limited automatic response support are also available today (TippingPoint, 2007; Enterasys, 2007). While the research approaches employ a range of different response selection principles, commercial tools provide only static mapping response as simplest and easily maintainable solution.

An ideal intrusion response system. In light of the above discussion, we see the following features as necessary requirements for a viable intrusion response system.

- **Automatic**. The volume and the intensity of the nowadays intrusions require rapid and automated response. The system must be reliable to run without human intervention. Human supervision often brings a significant delay into intrusion handling; the response system alone should have means to contain incurred damage and prevent harmful activity. Although complete automation may not be achievable in practice due to presence of newer and novel intractable intrusions, significant reduction of human effort and expert knowledge is desirable.

- **Proactive**. The modern software systems are built on multiple heterogeneously-developed components that have complex interactions with each other. Because of these interac-

tions, intrusions are likely to spread faster in the system, causing more damage. Proactive approach to response is the most practical in intrusion containment.

- **Adaptable**. The presence of multiple components, that constitute a software system, also results in a dynamic environment owing to the complex interactions between components. As such, intrusive behavior can affect system in a way which is unpredictable. The intrusion response system should be equipped with means to recognize and react to the changes in the dynamic environment.

- **Cost-sensitive**. Response to intrusions in dynamic and complex systems requires careful consideration of the trade-offs among cost and benefits factors. A simple basic response action triggered every time certain symptom is observed might be a wasteful effort and can cause more damage.

Considering the discussed above points as fundamental features of the intrusion response system, we propose an intrusion response selection model that provides automatic and preemptive selection and deployment of the optimal response strategy. The selection of the optimal response takes into account a potential impact of the detected intrusion and possible damage of the severe or incorrect response action and allows to adapt the response selection according to the failure or success of the previously deployed responses. The details of the proposed intrusion response selection model are presented in the following chapters.

IRS	Year published	Response Selection	Response time	Adjustment ability	Cooperation ability
DC&A (Fisch, 1996)	1996	dynamic mapping	delayed	static	cooperative
CSM (White et al., 1996)	1996	dynamic mapping	delayed/proactive	static	autonomous
EMERALD (Porras and Neumann, 1997)	1997	dynamic mapping	delayed	static	cooperative
BMSL-based response (Bowen et al., 2000)	2000	static mapping	delayed[1]	static	autonomous
SoSMART (Musman and Flesher, 2000)	2000	static mapping	delayed[2]	static	cooperative
pH (Somayaji and Forrest, 2000)	2000	static mapping	delayed	static	autonomous
Lee's IRS (Lee et al., 2000)	2000	cost-sensitive	delayed	static	autonomous
AAIRS (Carver and Pooch, 2000)	2000	dynamic mapping	delayed	adaptive	autonomous
SARA (Lewandowski et al., 2001)	2001	static/dynamic mapping[3]	delayed	static	cooperative
CITRA (Schnackenberg et al., 2001)	2001	static/dynamic mapping	delayed	static	cooperative
TBAIR (Wang et al., 2001a)	2001	static/dynamic mapping	delayed	not defined[4]	cooperative
Network IRS (Toth and Kruegel, 2002)	2002	cost-sensitive	not defined[5]	static	cooperative
Specification-based IRS (Balepin et al., 2003)	2003	cost-sensitive	delayed	static	autonomous
ADEPTS (Foo et al., 2005)	2005	cost-sensitive	proactive	adaptive	autonomous
FLIPS (Locasto et al., 2005)	2005	static mapping	proactive	static[6]	autonomous

Table 2.2 Classification of the surveyed systems.

[a] Although not clearly described, the approach can be extended to proactive response.

[b] Although use of *case-based reasoning* technique can be adjusted to recognize repetitive attacks in advance.

[c] The authors also mention application of more complex response strategies based on some decision-making process.

[d] Proposed work only describes the general principles of framework.

[e] The paper only presents an algorithm for evaluation of response impact.

[f] Although the approach is called "hybrid adaptive intrusion prevention", adaptiveness mainly refers to the detection of future attacks based on the feedback, and hence does not fall into adaptive response selection category

CHAPTER 3. Detecting anomalous behavior of software system

In general, behavior of software systems can be specified in two ways: *by internal system state*, for example, using value of the variables, or *through observable interactions between the system and the environment*, for example, via commands issued by a controller or system calls invoked by a device driver (Uppuluri, 2003). The main challenge of monitoring internal system state is that it requires modification of the system execution which can be prohibitively expensive. In the latter approach, program behavior can be captured by the monitoring sequences of observable actions generated by the system. In this context, system behavior is represented by a set of all possible sequences observed during any program execution in the system.

In our work we adapt the second approach and focus on the monitoring system behavior represented in terms of system calls.

3.1 Intrusion detection component overview

Our model for monitoring system-call sequences issued by a program consists of a two-level classification mechanism (Figure 3.1). Specification of normal and abnormal[1] behavioral patterns are provided in the first level. In the event that the sequence to be monitored matches the specification, the second level classification is not invoked. The sequence that match legal specifications is allowed to execute un-altered while an anomalous sequence is blocked and appropriate response actions are triggered. If the sequence is *not* found in the specification module, the second-level classifier is used. We then rely on machine learning techniques to determine whether the sequence is normal or anomalous. In either case, the sequence is recorded in the corresponding specification for future reference. One of the important features of our

[1] We will use terms normal and legal, and similarly, terms abnormal and anomalous interchangeably.

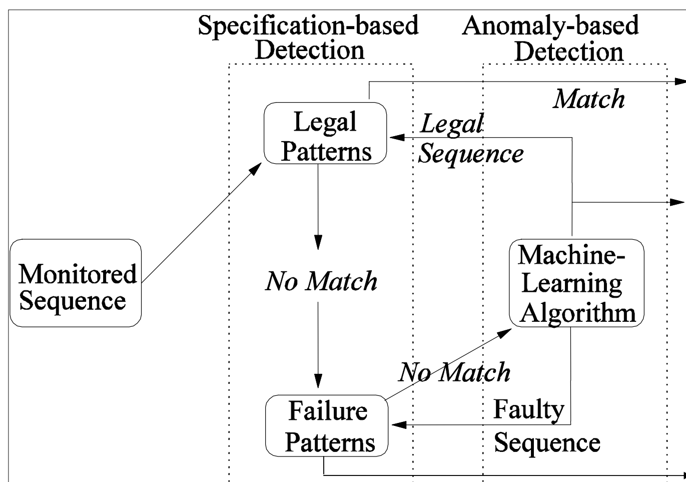


Figure 3.1 Architectural model of our framework.

model is that the technique can be deployed with empty or partial specifications in the first level. As more sequences are classified by the second level, the specifications are populated automatically. This flexibility reduces the overhead of developing the specifications manually.

3.1.1 Extended Action Graph: `Exact`

We model specifications in terms of Extended Action Graph (`Exact`) which is defined as follows:

Definition 1 (`Exact`) *An Extended Action Graph is a tuple* $E = (S, S_0, \rightarrow, \Sigma, L)$ *where* S *is the set of states,* $S_0 \subseteq S$ *is the set of start states,* Σ *is a set of binary numbers used to represent transition,* $\rightarrow \subseteq S \times \Sigma \times S$ *is the set of transition relations, and* $L : S_0 \rightarrow \Sigma$ *is a mapping of start states to a binary vector.*

A *sequence* in `Exact` is represented by $s_1, s_2, \ldots s_n$ where each s_i has a transition to s_{i+1}. Consider the example in Figure 3.2. Each transition and the start states are labeled by a

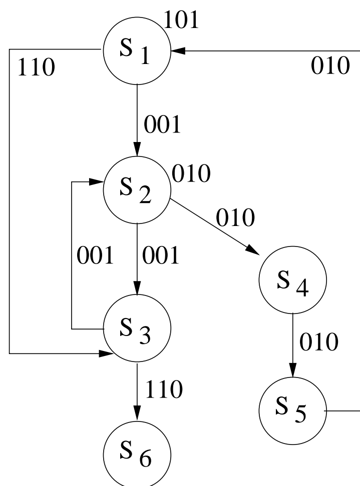


Figure 3.2 Example of an `Exact` graph that was generated by three sequences s_1, s_2, s_3, s_2, and $s_2, s_4, s_5, s_1, s_3, s_6$, and s_1, s_3, s_6. The graph has six states and two start states s_1 and s_2.

binary vector; e.g., $L(s_1) = 101$.

It is worth mentioning here that all the sequences in `Exact` are not classified as valid and valid sequences form a superset of the known sequences (sequences from which the `Exact` was constructed in the first place). In the above example, s_1, s_2, s_3 and s_1, s_3, s_6 are valid patterns, and the graph also contains the sequence s_1, s_2, s_3, s_6 which is not valid.

To rule out invalidity, we use the transition label σ, *a binary vector*, whose k-th element is denoted by $\sigma[k]$. If there exists a transition $s_i \xrightarrow{\sigma_i} s_j$ where $\sigma_i[k] = 1$, then s_i, s_j are said to be consecutive alphabets in the k-th *known* sequence. Note that the first sequence is identified by setting the *rightmost* bit to 1; i.e., 001 is the identifier for the first sequence, 010 is the identifier for the second sequence and so on. In Figure 3.2, s_1, s_2 are consecutive states in the first pattern while s_1, s_3 are consecutive states in the second and third known sequences. Every known sequence is also assigned a start state: s_2 is the start state of the second known sequence. Formally, using the known sequences, we define validity as follows:

Definition 2 (Validity) *A sequence $s_1, s_2, \ldots, s_n$ is said to be valid if $s_1 \in S_0$, $L(s_1) = \sigma_s$*

```
1: bool match(s_{l... k}, m, E) {
2:  if (s_l ∈ S_0) && (s_l -σ_l-> s_{l+1}) {
3:    m=(m & σ_l & L(s_l))
4:   if (m!=0) {
5:    int i=l+1; visited(s_l) = true;
6:    while (i<k) {
7:     if (s_i -σ_i-> s_{i+1}) {
8:      if (m=m & σ_i)&&(m!=0) {
9:        i=i+1; visited(s_i)=true; }
10:      else if visited(s_i) {
11:        reset(visited); m=set(1);
12:        return match(s_{i... k}, m, E);
13:      }
14:    }  else return false;
15:   } // end-while
16:    return true;
17:  }  else return false;
18: }  else return false;
19:}
```

(a)

```
1: void insert(s_{l... k}, m, E) {
2:  int m1=set(1); // m1 is all 1's
3:  if match(s_{l... k}, m1, E)  return;
4:  make_start(s_l, S_0); update_L(s_l, L, m);
5:  int i=l;
6:  while (i<k) {
7:     if (visited(s_i)) {
8:      reset(visited); m=m<<1;
9:      insert(s_{i... k}, m, E);
10:      return;
11:    } // end of if-then
12:     else {
13:      visited(s_i)=true;
14:      update(s_i, s_{i+1}, m, E);
15:      i=i+1;
16:    } // end of if-else
17: } // end of while
18:  return;
19:}
20: void update(s_i, s_j, m, E) {
21:  if (s_i -σ-> s_j ∈E) σ = σ | m;
22:  else connect(s_i -m-> s_j, E);
23:}
```

(b)

Figure 3.3 (a) Pseudo-code for `Exact` search. (b) Pseudo-code for `Exact` insert.

and

$$\exists k \ \forall i : 1 \leq i < n,\ s_i \xrightarrow{\sigma_i} s_{i+1} \Rightarrow (\sigma_s[k] = 1 \wedge \sigma_i[k] = 1)$$

In other words, there exists a specific element in the vector-label of each transition in this sequence and the vector-identifier of the start state which is equal to 1. Furthermore, via transitivity, if $S_1 = s_i, s_{i+1}, \ldots, s_{i+n}$ *is a valid sequence and* $S_2 = s_j, s_{j+1}, \ldots, s_{j+m}$ *is another valid sequence such that* $s_{i+n} = s_j$ *then* $s_i, s_{i+1}, \ldots, s_{(i+n)-1}, s_j, s_{j+1}, \ldots, s_{j+m}$ *is also a valid sequence.*

Validity takes care of unbounded (one or more) repetition of the alphabets in a sequence; e.g., in Figure 3.2 $s_1, s_2, s_3, s_2, s_3, \ldots$ is a valid sequence. In the above, s_2 is said to be the *root of the loop* and $s_2, s_4, s_5, \ldots$ represents the *exit* from a loop. Note that the transitivity relation in Definition 2 can be used to identify valid sequences with bounded repetition (from a valid sequence with finite looping and a valid exit sequence). For example, in Figure 3.2 s_1, s_2, s_3, s_2 and s_2, s_4, s_5 are valid sequences and they form, via transitivity, a new valid sequence $s_1, s_2, s_3, s_2, s_4, s_5$. Note that newly formed sequence might be a known as well as

unknown sequence.

Searching `Exact`. Based on the validity definition, we present in Figure 3.3(a) the algorithm to find out whether a given sequence is a valid sequence in `Exact`. Recursive procedure `match` takes as input the given sequence $s_{l...k}$, a bit-vector `m` and the `Exact` and returns true if the sequence is present as a valid sequence in `Exact`. The top-level call to `match` is invoked with `m` set to its maximal value (all bits set to 1). Line 2 checks whether the first state in the sequence is a start state in `Exact` and records in `m` the possible sequence identifiers between s_l and s_{l+1} (Lines 3-4). The algorithm iteratively checks in Lines 6–15 for the validity of states in the sequence by considering each transition. Lines 9–11 handle the situation where a given sequence is comprised of several valid sequences obtained via transitivity (see Definition 2). Line 11 identifies the preceding valid sequence and initializes `m`. Note that we use `visited`(s) to track whether the state s is already visited in the search path. Line 12 recursively invokes procedure `match` on obtained valid sequences.

Complexity of match. `Exact` is deterministic; i.e., for every pair of states there exists at most one transition. Absence of non-determinism makes the complexity of searching for a valid sequence linear in the size of the given sequence.

Constructing `Exact`. Figure 3.3(b) presents the algorithm for insertion of a new sequence in `Exact` graph. Procedure `insert` takes as arguments the sequence to be inserted, a bit vector `m` identifying the new sequence and the graph `Exact`.

Lines 2–3 check whether the sequence to be inserted is already present in the graph (`match` invoked). Otherwise, the first alphabet in the sequence is marked as the start state of the sequence and the corresponding labeling function is updated (Line 4). For example, if s_l had a prior start-state label σ_s, then `update_L` sets its new label to σ_s|`m` (bitwise OR-ing).

The remaining sequences of alphabets are introduced iteratively (Lines 6–17). We need to consider the case where an alphabet repeats in the sequence. Recall from Definition 2 that repetitions are handled via transitivity, i.e., repetition results in new sequences. As

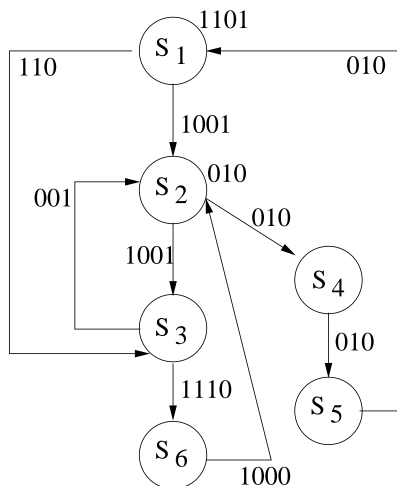


Figure 3.4 Exact graph in Figure 3.2 shown after insertion of $s_1, s_2, s_3, s_6, s_2, s_4, s_5$.

in the matching algorithm, we use the auxiliary information `visited` to record whether a state/alphabet has been already seen in the sequence. In the event of repetition (Line 7), all the `visited` information is reset to false, a new sequence identifier is generated by left shifting `m` (Line 8) and `insert` is invoked recursively on the rest of the sequence. Otherwise, as shown in Lines 13–15, the transition between s_i and s_{i+1} is updated.

The `update` procedure elaborates this operation (Lines 20–23). If a transition already exists between the source and destination states, its transition label is updated via bitwise OR-ing of the existing transition-label and the new sequence identifier; otherwise a new transition is inserted.

Complexity of insert. In the above algorithm, the procedure `match` is always invoked before inserting any new sequence to avoid duplicate insertions. As such, the worst case complexity of the insertion algorithm is $O(r \times n)$ where r is the number of repeated occurrences of alphabets in the sequence of length n.

Illustrative Example. Let $s_1, s_2, s_3, s_6, s_2, s_4, s_5$ be a sequence to be inserted in the example `Exact` in Figure 3.2.

1. Procedure `match` identifies two substrings s_1, s_2, s_3, s_6, s_2 (up to the first repeated alphabet s_2) and s_2, s_4, s_5 (following the transitivity rule) and recursively invokes `match` algorithm on each substring.

   The sequence s_1, s_2, s_3, s_6, s_2 is not present in the `Exact` graph shown in Figure 3.2 as there is no transition from s_6 to s_2. The `match` algorithm (Figure 3.3(a)), in this case, makes an *early* detection of its absence as the prefix s_1, s_2, s_3, s_6 of the given sequence is not valid in the exact. Observe that, for this prefix, bit-wise "and"-ing of the transition labels and the start state label results in 0 (101 AND 001 AND 001 AND 110 = 000, see Figure 3.2) and our `match` algorithm returns false. As such, s_1, s_2, s_3, s_6, s_2 is inserted as a new sequence. Further, as s_1 is already present in the set of start states, its start-state label is updated using the new sequence identifier 1000. Recall that the identifier for the first sequence is 001, identifier for the second sequence is 010 and for the third sequence is 100. As such the identifier for the new (fourth) sequence is 1000.

2. Each transition of the new sequence is added to `Exact` graph with identifier 1000. We start with transition $s_1 \rightarrow s_2$. It already exists in the graph and its identifier is 001. Applying *bitwise-OR* of the existing and new transition label we obtain 1001 and update the transition with this new label (Figure 3.4). We continue in this fashion until we reach transition substring s_6, s_2. There is no transition between s_6 and s_2. A new transition ($s_6 \rightarrow s_2$), therefore, is added with the transition 1000.

3. Due to the repeated appearance of s_2, the second sequence s_2, s_4, s_5 is set up to be inserted as a new sequence with a new sequence number, 10000. However, its insertion is avoided as the sequence s_2, s_4, s_5 is already a valid sequence in `Exact`. The updated `Exact` graph is shown in Figure 3.4.

3.1.2 Monitoring for abnormal behavior

We presented two main algorithms: `match` and `insert` for generating and maintaining the `Exact` graph structure. In this section we present a procedure for monitoring system behavior against patterns stored in `Exact` graphs.

```
1: seq monitor(seqIn, s_k, E) {
2:     if ((seqIn) == null) {
3:             seq = s_k;
4:             start(seq) = s_k;
5:     }
6:     else {
7:             start(seq) = start(seqIn);
8:             seq = seqIn ⊕ s_k;
9:     }  //   (end-if)
10:    bool seqBad = false;
11:    bool seqGood = false;
12:    int m=set(1);    // m is all 1's
13:    if (match(seq, m, E_bad))
14:            seqBad = true;
15:    m=set(1);    // m is all 1's
16:    if (match(seq, m, E_good))
17:            seqGood=true;
18:    if (seqBad==true) {
19:           if (seqGood==true) {
20:                   delay = true;
21:                   return seq;
22:           }
23:           else {
24:                   invokeResponse(seq);
25:                   return null;
26:           }
27:      }   // (end-if)
28:      else {    // (seqBad == false)
29:           if (seqGood==true)
30:                   return null;
31:           else {
32:                   delay = false;
33:                   invokeClassifier(seq);
34:                   return null;
35:           }
36:      }
37:}
```

Figure 3.5 Pseudo-code for monitoring behavior.

Figure 3.5 describes an algorithm for monitoring system behavior for preemptive detection of abnormal patterns of system calls.

System calls generated by the system are continuously matched with `Exact` containing normal and abnormal patterns. System calls that represent normal behavior are allowed to execute while suspicious calls are buffered until anomalous behavior is confirmed and response mechanism is invoked.

Delay of suspicious pattern of system calls is necessary in cases when the observed prefix matches patterns contained in normal and abnormal `Exact` graphs. To avoid a possibility of a false alarm in this case, suspicious system calls are delayed until the observed sequence

becomes truly abnormal, i.e. does not match `Exact` with normal patterns, or until sequence becomes truly normal, in other words does not represent patterns contained in the abnormal `Exact`. In the case, when encountered pattern is unknown which is indicated by its absence in the `Exact`, the sequence is allowed to execute and later classified by the machine-learning algorithm.

Procedure `monitor` describes the monitoring of the delayed system calls. This procedure takes as input currently delayed sequence of system calls `seqIn`, the latest system call generated by the system s_k, and the `Exact` and returns the sequence `seq`.

The algorithm can be invoked with the empty input sequence `seqIn`, then current call s_k will be set as the start state of this sequence (Lines 2-4), or with the previously delayed sequence. In the latter case, the start state of the sequence `seqIn` is set as the start of `seq` (Line 7) and s_k is added as a last call forming a sequence `seq` (Line 8).

Lines 10-35 determine the classification of the considered sequence.

In Lines 13-14 the `match` procedure is invoked to check whether a newly formed sequence `seq` is a valid pattern of `Exact` with anomalous sequences and Lines 16-17 determine if this sequence constitutes a prefix of a valid pattern of normal `Exact` E_{good}. In both cases `match` is invoked with a bit vector `m` set to its maximal value (all bits set to 1).

If a newly formed sequence matches prefixes of patterns of normal and abnormal `Exact`, the processing of the sequence should be delayed. This is indicated by global variable `delay` set to true (Lines 18–21). Lines 24–25 handle the situation when a newly formed sequence represents a prefix of the abnormal pattern and should be treated as truly normal.

If the sequence represents a truly normal pattern, the procedure return an empty sequence indicating that the further monitoring of this sequence is not necessary (Lines 29-30). The last lines (32-33) represent a case when observed pattern is unknown and needs to be processed by the classification algorithm.

3.1.3 Second-level Classifier

We rely on machine learning technique to classify behavioral patterns as normal and abnormal depending on how well they fit in the learnt data domain. While any machine learning technique that provides fast and accurate classification can be applied as a second-level classifier in our framework, in our study, we used support vector machines (SVM).

SVM were initially introduced by Vapnik (Vapnik, 1998) and have exhibited excellent accuracy on test sets in practice while having a strong theoretical motivation in statistical learning theory. There have been proposed two versions of SVM algorithm: a supervised version, called two-class SVM and unsupervised referred to as one-class SVM. A supervised version of SVM works with labeled data sets and finds hyperplanes also called support vectors that maximally separate the data belonging to different classes (Vapnik, 1998). As opposed to two-class SVM, one-class SVM relies on separating all data from the origin using a hyperplane (Scholkopf et al., 1999).

For the purpose of discussion, we illustrate the application of SVM classifier via an example. Let the observed input stream be $\texttt{Istream} \equiv s_1, s_2, s_3, s_2, s_3, s_4, s_2$ and `Exact` in Figure 3.3(b) failed to recognize `Istream` as a valid sequence. First, we break-up `Istream` following the transitivity relationship in Definition 2 of Section 3.1.1; i.e., $\texttt{Seq}_1 \equiv s_1, s_2, s_3, s_2$ and $\texttt{Seq}_2 \equiv s_2, s_3, s_4, s_2$. Note that the break-up point is at s_2 which appears in $\texttt{Seq}_1$ and $\texttt{Seq}_2$, and is the first alphabet to be repeated in `Istream`. SVM can only take fixed length sequences as input and as such we apply classic sliding window technique to provide inputs to the SVM.

Let the sliding window size be 3, then SVM is fed with subsequences: (i) s_1, s_2, s_3, (ii) s_2, s_3, s_2 (from $\texttt{Seq}_1$), (iii) s_2, s_3, s_4 and (iv) s_3, s_4, s_2 (from $\texttt{Seq}_2$). Finally, $\texttt{Seq}_1$ and $\texttt{Seq}_2$ are termed as normal if and only if all their subsequences are classified by SVM as normal. Note that, break-up of $\texttt{Seq}_1$ and $\texttt{Seq}_2$ using sliding window does not adversely effect end result, i.e., if any subsequence of $\texttt{Seq}_1/\texttt{Seq}_2$ is classified as anomalous, then the corresponding sequence is conservatively classified as anomalous. Furthermore, the sequences $\texttt{Seq}_1$ and $\texttt{Seq}_2$ provide an easy way of inserting `Istream` in the corresponding `Exact`.

3.2 Analysis of Exact and SVM Sequences

As `Exact` represents variable-length sequences, the comparison with models based on the fixed-length patterns is challenging; the main challenge being the difference in the number of fixed-length and variable-length sequences generated from the same data set. The comparison is also aggravated by the fact that classification of variable-length patterns in `Exact` depends entirely on the underlying fixed-length classifier (SVM in this case).

In this section, we present a comparative study of number of sequences being examined by `Exact` and the SVM classifier. We consider two possible cases: one where the SVM, used in conjunction with `Exact`, acts as the backend for our framework (backend SVM) and the other where SVM acts alone (stand-alone SVM).

For the purpose of analysis, we will consider average length of `Exact` sequences; the average length computed using the weighted mean where the weight of a length denotes the number (frequency) of sequences of the corresponding length. We represent this length as L. Let the fixed sliding window size of SVM be W.

3.2.1 Exact vs. Backend-SVM

The two possible scenarios of interest are $W > L$ and $W < L$. For $W = L$, the number of `Exact` sequence and SVM sequence is identical.

1. $W > L$: In this case, several `Exact` sequences are combined to form one SVM sequence. Consider first the case where x `Exact` sequences fit *exactly* in one SVM sequence of size W. In other words, $xL - (x-1) = W$ (the subtraction of $x - 1$ from xL is required to account for overlap between two consecutive `Exact` sequences). Therefore,

$$x = \frac{W-1}{L-1} \tag{3.1}$$

   In other words, the number of `Exact` sequences is greater than the number of SVM sequences and classification of one SVM sequence influences the classification of x `Exact` sequences.

Secondly, consider the case where $(W-1)/(L-1)$ is not a whole number; i.e., the `Exact` sequences is not a integer-multiple of SVM sequences. Let x be the smallest number of `Exact` sequences such that $xL-(x-1)>W$ and $\forall y<x: yL-(y-1)<W$. Therefore, the number of SVM sequences corresponding to x `Exact` sequences is,

$$xL-(x-1)-W+1 = x(L-1)-W+2$$

Then the number of `Exact` sequences is greater than the number of SVM sequences if $x<(W-2)/(L-2)$; otherwise the number of SVM sequences is greater than the number of `Exact` sequences. In case of former, one SVM sequence classification influences one `Exact` sequence classification while in latter, one SVM sequence can potentially effect x `Exact` sequences.

2. $W<L$: In this case, the number of `Exact` sequences is less than the number of SVM sequences. Specifically, the number of SVM sequences corresponding to one `Exact` sequence is $(L-W+1)$ and therefore, one SVM sequence classification can effect the classification of one `Exact` sequence.

3.2.2 Exact vs. Stand-alone SVM

Next, we consider the number of sequences examined by SVM if it is deployed alone. Given that the total length of the input stream is IS, the total number of SVM sequences is $N=IS-W+1$. If the same input stream is input to our framework – `Exact` with backend-SVM – the total number of `Exact` sequences is $(IS-1)/(L-1)$; i.e., $(N+W-2)/(L-1)$. The number of sequences examined by SVM alone is greater than the number of `Exact` sequences in our framework if $N>(W-2)/(L-2)$.

Also, note that if $W<L$, the number of sequences examined by SVM, when deployed alone, can be potentially greater than the number of sequences examined by SVM, when deployed in conjunction to `Exact`. Specifically, the situation requires $N>L-W+1$ and can be explained from the fact that number of sequences classified by backend-SVM depends on the number of `Exact` sequences when $W<L$.

3.3 Preliminary Simulation results

	stand-alone SVM			`Exact` (variable-length sequences)		
	snsndmailcp	decode	fwdloop	snsndmailcp	decode	fwdloop
Number of normal sequences in train data set	30792	30792	30792	3314	3314	3314
Number of sequences in test data set	1098	2983	2499	78	405	204
Number of anomalous sequences in test data set	264	741	387	24	92	43

Table 3.1 Information on sendmail normal and intrusive trace data sets

We evaluated our model in the security domain using synthetic sendmail data provided by the UNM (Forrest, 2005). Sendmail data is an unlabeled collection of system calls. It consists of a normal data sets which contain only legal patterns and trace data sets containing normal patterns as well as anomalies. We considered three intrusion trace data sets: snsndmailcp, decode and fwdloops. The one-class SVM classifier was trained on the normal data set (training set), tested on the trace sets (test sets) and implemented using libsvm tool (Chang and Lin, 2001) and the window size of 8.

Data Sets. Table 3.1 presents the pattern of data being used for evaluation purpose in terms of number of sequences. The training data set contains 30792 normal fixed-length sequences. On the other hand, using `Exact`, the number of variable-length sequences is 3314. The decrease in the number of sequences is due to the fact that `Exact` partitions sequences using repetitions and as such can handle variable-length sequences (see Section 3.1.1). We then processed the normal and abnormal patterns of the test data set to generate two test sets: one for stand-alone SVM, containing fixed-length sequences obtained through sliding window technique, and one for `Exact`, containing variable-length sequences generated in `Exact` fashion (row 2 in Table 3.1).

Finally, the last row shows the number of sequences that are in the test data set but are not present in the training data set. For example, out of 1098 fixed-length sequences for `snsndmailcp`, there are 264 sequences which are not present in fixed-length sequences of

	snsndmailcp	decode	fwdloop
`Exact` of normal specifications	5	16	13
`Exact` of faulty specifications	14	31	39

Table 3.2 Maximum length of `Exact` binary vectors.

training data. For the purpose of evaluation, we can conservatively assume that sequences not present in the training data set are anomalous; the goal is to identify all such anomalous sequences.

Table 3.2 presents the maximum length of binary vectors after building `Exact` graphs on data sets. In other words, this shows the number of distinct variable-length sequences in `Exact`.

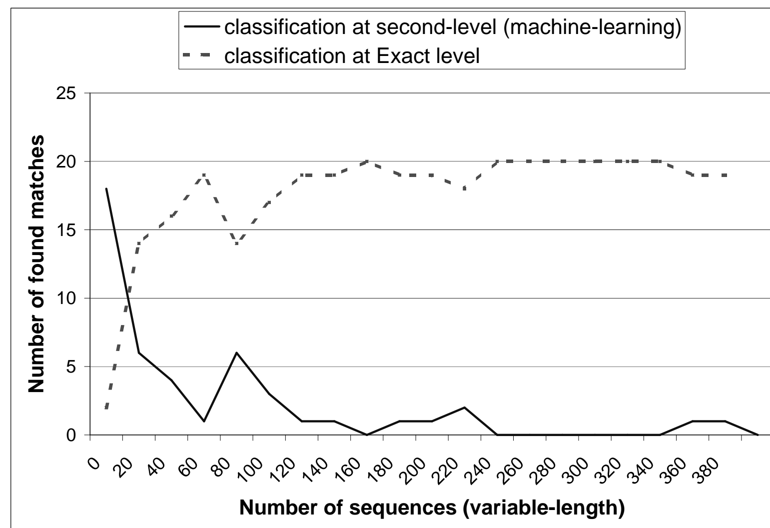


Figure 3.6 Initialization: empty `Exact` (*decode* intrusion).

Efficiency. In these experiments we focused primarily on the rate of populating the `Exact` with normal(legal) and abnormal(anomalous) patterns. To evaluate our technique we monitored the stage at which each sequence was classified. We examined two scenarios:

1. Both `Exact` graphs representing normal and abnormal specifications are initially empty

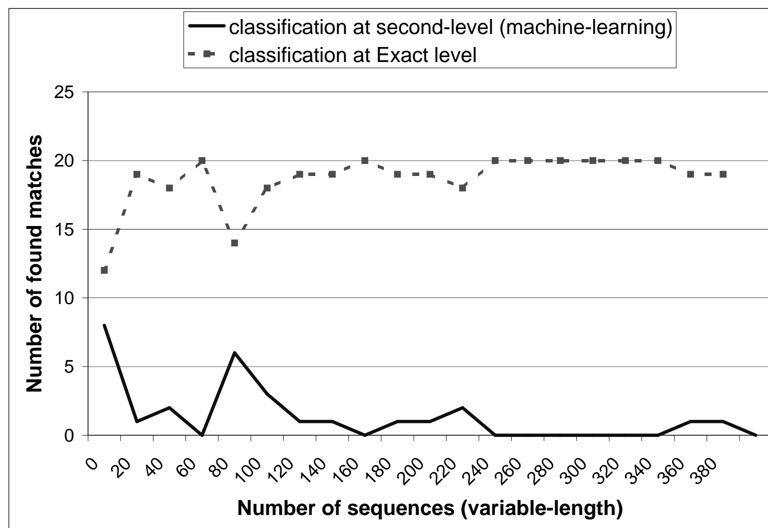


Figure 3.7 Initialization: `Exact` partially populated with 10% of normal sequences (*decode* intrusion).

2. Partial specification is available initially, i.e., the `Exact` graph corresponding to a normal specification is populated with 10% of the patterns from the normal data set.

The results for both scenarios are presented in Figures 3.6, 3.8 and 3.7.

Figure 3.6 shows the frequency at which both levels of classifiers (`Exact` and backend SVM) were invoked for classifying the incoming sequences. Since simulation started with an empty `Exact` graph, almost every incoming sequence is classified at the second-level classifier. However, the access rate of second-level classifier rapidly decreases as more patterns were stored in the `Exact`. Consequently, the number of sequences classified at the `Exact` graph level increases. Figure 3.8 shows the number of new patterns added to the empty `Exact` over the same run of decode trace set. The majority of patterns were recorded within about 200 sequences (out of 405 total sequences). After that, almost all patterns were found at the `Exact` level.

The result corresponding to the second scenario where the normal `Exact` graph is partially populated is shown in Figure 3.7. As opposed to the previous figure, the access rate of the second-level classifier in the beginning of the run is low while the `Exact` graph access rate is

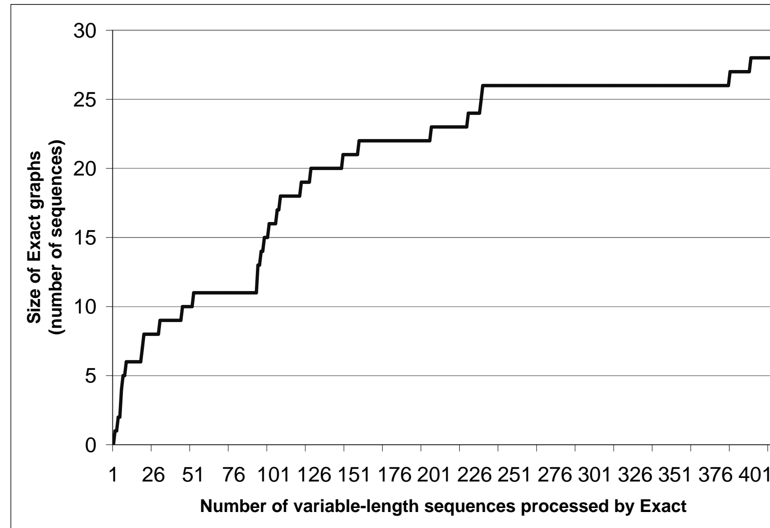


Figure 3.8 Populating `Exact` (*decode* intrusion).

	Total time(sec)	Backend SVM running time(sec)
`Exact` with empty specs	16	12
`Exact` with partially populated specs	7	5

Figure 3.9 Mean running time(*decode* intrusion. Average over 10 runs.

high. This is explained by the partial presence of the sequences in the normal `Exact` specifications. However, since only partial normal patterns were added to the specifications, the second-level classifier was still accessed whenever new normal or anomalous sequence is found.

In this scenario we benefited from the available specifications having populated the `Exact` in advance. This shortened the start-up time necessary to store a sufficient number of patterns (Table 3.9). In fact, the processing time for 405 sequences was 2 times faster with the populated specifications (7 sec) than with the empty specifications (16 sec). Note that it is the SVM classifier access that requires most of this time.

	stand-alone SVM			Our framework (results from the backend SVM based on fixed-length sequences)		
	snsndmailcp	decode	fwdloop	snsndmailcp	decode	fwdloop
Detection rate	98%	99%	99%	98%	100%	100%
FP rate	11%	7%	10.7%	13%	8%	10%

Table 3.3 Accuracy of classification with empty `Exact` shown in fixed-length sequences.

	empty `Exact`			`Exact` populated with 10% of normal sequences		
	snsndmailcp	decode	fwdloop	snsndmailcp	decode	fwdloop
Number of detected sequences	24 out of 24	90 out of 92	42 out of 43	24	90	42
FP sequences	21 out of 54	62 out of 313	75 out of 161	0	1	9

Table 3.4 Accuracy of our framework classification shown in variable-length sequences.

Accuracy. As the `Exact` graph provides a succinct representation of learned through machine-learning technique variable-length patterns, we focused in these experiments on the comparison of the accuracy of our structure to the accuracy of SVM tested on a model built using the sliding window technique.

For evaluation purpose we considered *detection rate* (ratio of detected anomalies to the total number of anomalies presented in the set) and *false positive rate* (FP) (number of normal instances incorrectly identified as anomalous).

As Table 3.3 show, classification results of fixed-length patterns for stand-alone SVM and `Exact` integrated with SVM are similar. For example, for `snsndmailcp`, the detection rate is 98% for both stand-alone SVM and back-end SVM used in `Exact`. The results confirm the fact that `Exact` structure, while recording variable-length patterns, truly represents information given by the backend machine-learning based classifier in compact fashion. The existing 1–2% variation in the results is explained by the potential difference in the number of sequences examined by stand-alone SVM and back-end SVM used in `Exact` as discussed in Section 3.2.

The classification results are also given in terms of variable-length patterns stored by `Exact` (Table 3.4). Examining Table 3.4, we notice that prediction results are slightly different from the corresponding percentages given in Table 3.3. For example, the detection rate of `Exact` for snsndmailcp intrusion given in fixed-length patterns is 98% while the corresponding number of detected variable-length sequences is 24 out of 24. This happens when several SVM sequences, including those that are correctly classified as anomalous and those that represent missed

intrusions, are effectively combined into one `Exact` sequence resulting in an anomalous `Exact` sequence and providing a higher detection rate.

An opposite scenario is represented by `decode` intrusion, where the detection rate in fixed-length patterns is 100% which corresponds to 90 out 92 variable-length `Exact` sequences. Closer inspection reveals that the result is as expected and can be explained by the fact that `Exact` records sequences depending on the classification result from backend SVM classifier. There are a couple of occurrences of one particular `Exact` sequence in the test data set which is not present in the training data set. Hence, this sequence is classified as an anomaly (counted as one of the anomalous patterns among 92 anomalies: see Table 3.1). It turns out that the length of the sequence is 2 due to two consecutive identical system call-invocations. As such the SVM using sliding window size 8 does not consider this sequence independently; instead it combines the sequence with another (next) `Exact` sequence and performs classification. As the combined sequences are classified as normal by SVM, the `Exact` also records the combined sequence as normal. This is acceptable as the main purpose of `Exact` is to memorize variable length sequence and closely follow SVM classifier. Note that if the SVM classifier used window size of 2, then the above scenario would be removed.

The number of variable-length sequences falsely recognized as positive in `Exact` is also different from the corresponding percentages given for fixed-length sequences. This is due to the fact that several SVM sequences can represent one `Exact` sequence, thus significantly reducing the total number of variable-length sequences in comparison to those in fixed-length. The detailed analysis of this dichotomy was presented in Section 3.2. At the same time, manual inspection of these results showed that a number of FP sequences in `Exact` graph fully comes from the backend SVM.

While the trade-off between the number of detected and the false positives is inherently present in many machine learning algorithms including SVM, this error can be effectively reduced with guidance from normal specifications. In fact, populating `Exact` even with the small number of normal patterns reduces the number of false positives significantly (Table 3.4). Since the overall variability of sendmail behavior is small, even approximately 10% of normal

sequences leads to recognition of majority of normal patterns. However, generally a greater variability in process behavior might require a larger set of normal patterns to improve the accuracy of classification. Note that an `Exact` with partially populated normal specification does not affect the number of detected sequences. This is because abnormal, incoming sequences are still recognized as unknown and processed by SVM algorithm as they would be if the `Exact` graph were empty.

CHAPTER 4. Automated Intrusion Response

In the past chapter we introduced a model for detecting anomalous patterns in program behavior based on the normal profile of the program. While this is a fundamental component of the security system, it presents a challenge on how the system should react to the exposed anomaly. Therefore, next we focus on the design on the response component that is tightly coupled with the detection part of our IDR system.

Generally, the design of a response system can be approached from the attack point of view when the response is determined at each attack step based on the available attack signatures or from the system perspective that considers the impact of the successful attack on the system services (Wu et al., 2007). The first approach is potentially limited as it only provides local response to the current manifestations of the attack. While such response strategy works well for simple attacks, it might not present necessary defense for the complex multi-stage attacks. The latter approach focuses on the system state rather than an attack; and thus, is able to see "the bigger picture", for example, in a form of system services degradation. However, since its primary goal is the system services, it might not be able to stop an intrusion, but rather limit its capabilities to effect the system further.

Both of these approaches complement each other providing on the one hand, a locally customized response strategy to a specific attack trace at different attack steps, and on the other, a global view of the attack impact.

In our model we combine both approaches taking advantage of their strengths while avoiding their weaknesses. We develop taxonomy of the system resources that might potentially become targets during an attack and the responses that can be effectively deployed to either counter possible attacks on these resources or defend system services and regain secure system

state. We also present a mapping between system resources, the corresponding responses and specific attack patterns (Appendix A).

4.1 Classification of System Resources

A process running on an OS updates or reads the following resources at a high level: memory, file systems and I/O devices such as network cards. Hence, any damage due to an attack can be classified as an unauthorized read or update operation on these resources. Based on the taxonomy of system resources proposed by (Bazaz and Arthur, 2005) we developed classification of the system resources used by a process and the corresponding behavior targeted into damaging the resource:

Memory Every process is allocated a process address space. Depending on the architecture this address space process is divided into several logical segments. Traditionally, these segments include: text (code) segment, data segment and stack (Younan et al., 2006). *Text segment*, sometimes also referred to as code segment, stores the code of the program and is read-only memory. *Data segment* contains global, static and dynamically allocated data. Functions and local variables are located on the *stack*. Both data and stack segments may be readable and writable. As a large number of attacks are against data segment (heap) or stack, we will further consider these two subcategories of the process memory.

Data segment. Typically, data segment includes the following sections *heap*, *BSS (block storage segment)* that contains static and global variables, *GOT (global offset table)* that stores pointers to dynamically linked functions and *.dtors* table that includes pointers to functions executed on process exit (Younan et al., 2006). In some architectures the BSS table and the heap are organized as separate memory segments (Krishnakumar, 2005).

Generally, any section of the data segment can be a target of an attack. These exploitation efforts can be broadly divided into the following areas:

1. *Illegal write to a process data segment:* Overwriting memory locations in the data segment a process can obtain access to the memory space of the current process as well as memory

allocated for another executing processes.

This violation can occur due to several reasons:

- If the data accepted as input by the process is larger than a dynamically allocated buffer space, then process overwrites memory that lies beyond bounds of the buffer (Simon, 2002; Fayolle and Glaume, 2002; Younan, 2003; Younan et al., 2006; Yong, 2004).
- If a pointer variable is overwritten to reference incorrect variable or memory location outside of the process memory space (C0ntex, 2005; Younan, 2003).
- If a specially crafted data injected into the dynamic memory and interpreted by a process as an executable code (Linn et al., 2005; Younan et al., 2005).
- Data interpreted by process as a format string can cause system to reveal process stack and in case of specially constructed format string to write data to the process stack (Younan, 2003).

2. *Incorrect variable format/values:* The following manipulations with a format or value of the variables can result in vulnerable system state:

- Unauthorized modifications of the format or values of environment variables (Secunia, 2004; Wheeler, 2003).
- The process stores a value of an integer variable larger/smaller than the maximum/minimum value allocated to it. This operation can result in the incorrect value stored in the assigned location as the system will discard the higher bits of the value; or in the possible heap overflow condition if the size of the integer is given as `0xffffffff` which will cause program to perform `malloc(0)` (Younan, 2003).
- The process using a negative value of integer variable/expression the object can create an error condition in the system. This type of misuse is possible due to abuse of signed and unsigned integer types (Younan, 2003).

Stack segment. The stack segment contains the local variables and return addresses of the functions for the particular process. Attacks can either write or read data from a stack illegally as follows:

1. *Illegal write to the stack memory:* If the process writes the data which size is larger that an allocated space on the stack, then the return address of the function requesting the write might be overwritten which cause function upon its return to jump to a random address in the stack. Such behavior is also known as *stack overflow* and might cause a variety of system errors (AlephOne, 1996; Ogorkiewicz and Frej, 2002).

2. *Illegal read of the stack memory:* Data belonging to a process in execution can be revealed if the content of the stack is observed by another process. Generally, this violation is possible as a result of a stack overflow attack (AlephOne, 1996) or through crashing a system and revealing its core file.

Input/Output resources Another main category of the presented classification of the system resource vulnerabilities is I/O resources. We confine our discussion to the two most widely used I/O resources: *File system* and *Network interface.*

File system Malicious behavior with regard to the file system can be broadly classified into the following three areas (Tsyrklevich and Yee, 2003):

1. *Illegal creation/deletion of files and file links:* There are several possibilities to damage the system through illegal creation or deletion of files and file links. One common method used by the attackers is the creation of symbolic links in the temporary directory during the execution of setuid program which as a result gives attacker an illegal access to the privileged files (Wheeler, 2003). A number of *Time of Use to Time of Check* (TOCTOU) attacks which exploit a race condition in file accesses are also based on creation/deletion of file links (Bishop and Dilger, 1996).

2. *Illegal read or write of file content:* There are several ways a process can write to a file or read from a file for which it does not have access. Most of them are possible due to the

assumption that sequential operations on the file are performed 'atomically'. This can be exploited by an attacker who accesses the file between successive operations performed by a legitimate process (Uppuluri et al., 2005). Such exploits are known as *race conditions* attacks. A detailed overview of these attacks can be found in (Uppuluri et al., 2005; Tsyrklevich and Yee, 2003).

3. *Incorrect file permissions:* This violation often happens due to improper access permissions assigned to a file or directory by a process and can result in unauthorized access to the file or directory.

Network Interface This subcategory refers to the interface that system uses to send and receive data from the network. As this interface is represented in a form of ports, network interface from the process perspective is another resource that can be used to transfer the data (Bazaz and Arthur, 2005). Thus, it can be classified into two broad subcategories:

1. *Illegal access to the data:* Unauthorized process can intercept the data sent by another process via the network interface before it reaches its destination.

2. *Wrong format/values of the data:* The data received by the process through the network interface are corrupted or do not have expected format.

Presented classification of the resources gives a general overview of possible violations resulting from abnormal behavior of the process; thus, allowing to determine the potential damage from the process actions.

4.2 Classification of Response Actions

Intrusion response is a reaction of the system directed to protect the system resources threatened by an attack. Based on system resource classification presented in Section 4.1, we developed the following classification of response actions that can be deployed to counter an attacks. Note that the granularity of these responses will vary depending on the severity of an

intrusion. For instance, filesystem isolation could imply isolation of one file as well isolation of the entire filesystem.

Generally, by the activity of the triggered response, intrusion responses can be classified into *Passive responses* and *Active responses*.

Passive response Passive response actions do not attempt to minimize damage already caused by the attack or prevent further attacks. Their main goal is to notify the authority and/or provide an attack information. This kind of responses includes:

1. *Administrator notification:* generation of alarm (through email, online/pager notification, etc.), generation of report.

2. *Other responses:* enabling detailed/additional logging, intrusion analysis tools and backing up tampered with files.

Active As opposed to passive response, active response actions aim to minimize the damage done by the attacker and/or attempt to locate or harm the attacker. We can distinguish three types of active response:

1. *Process termination:* A process can be terminated in response to an intrusion with or without the following automatic restart. Termination with automatic restart can be useful for remote servers such as sshd and httpd.

2. *Process isolation:* Process isolation refers to the execution of the process in the closed environment with a goal to control and thus contain the undesired behavior. We can distinguish complete and partial process isolation.

   - Complete isolation: execution of the entire process in an isolated environment.

     (a) Process migration: transferring the execution of the process to a different environment. Usually in distributed setting process migration indicates the execution of the process in a different machine.

     (b) Process emulation: execution of the process in a virtual environment.

- Partial isolation: execution of the process when its parts or used resources are located in an isolated environment.
  (a) Alternative methods to complete isolation: remote code execution, cloning, use of mobile agents etc.
  (b) Memory isolation
    i. Create a separate memory for the process to make updates
    ii. Disallow read of certain portions of memory
    iii. Disallow read of certain portions of memory during certain times of its execution
    iv. Disallow read/write of certain portions of memory for certain processes
  (c) Global Offset table (GOT) isolation. Global Offset table is mainly used by processes linked to dynamic libraries by resolving the memory location of dynamic library functions used by the process. An intrusive process can change GOT to force the program to execute incorrect library functions. To avoid this the GOT portion of a process can be isolated so that the illegal GOT changes will result in unharmful behavior or will serve as a trap for an attacker.
  (d) File system isolation
    i. File writes are made in a chroot environment (option: file system rollback)
    ii. File attribute changes are made in a chroot environment.
    iii. Disallow full/selective access to file
    iv. Allow read/write on sanitized/empty/dummy version of (privileged) file
  (e) Network isolation
    i. Isolate process by capturing at the packet filter level the data sent to certain IP addresses/subnets
    ii. Prevent the process from sending data
      – to certain IP addresses/subnets

1: **Preemption:**
-*determine when to start response selection*:
based on the preemption conditions defined in (Stakhanova et al., 2007a)
2: **Effect identification:**
-*identify the resources affected by an attack and the available responses*

3: **Response strategy selection**
-*determine whether response action should be taken at this point*:
select responses such that
damageCost*confidence level > responseCost

- *compute the costs of the selected responses*:
compute expected value of the selected responses

-*build a MinCostSAT formula*
-*find satisfiable solution through MinCostSAT solver*

Figure 4.1 Algorithm for response deployment process.

– to any server other than the one from which a connection was made to the process

iii. Prevent complete network utilization

iv. Prevent process access to all/some of the packet traffic

3. *Protection measures:* These actions are directed to prevent intrusive behavior or restore the pre-attacked state of the system.

- Delete created/accessed tampered with file
- Restore original/backed version of tampered with file
- Delay suspicious process

4.3 Intrusion Response Selection Model

Our intrusion response technique relies on the existence of a pattern-based intrusion detection framework, i.e., execution sequence is monitored against known/already examined normal (abnormal) patterns and is classified as anomalous or intrusive if the execution matches with

(deviates from) the known patterns. While we specifically focus on IDS proposed in Section 3, other pattern-based detection systems for example, specifying patterns of anomalous activities in a form of rules (Kumar and Spafford, 1994) or describing behavioral patterns using a specification language (Sekar and Uppuluri, 1999; Uppuluri and Sekar, 2000) can be employed.

Our intrusion response system involves the three steps of *preemption*, *effect identification* and *response strategy selection* (Figure 4.1).

4.3.1 Preemption

In this context, preemption, amounts to identifying the condition under which our system decides to *eagerly* respond before the classification of the execution pattern is complete. In other words, preemption leads to a deployment of response against *prefix* of a potentially intrusive execution sequence. The decision for preemptive response depends on *probability threshold*. We define this notion to indicate an acceptable level of confidence that some attack is in progress and a response action should be triggered. Once the probability of occurrence of a particular sequence, given its prefix, exceeds the pre-specified probability threshold , it can be inferred that an attack is imminent. Formally, probability of occurrence of a sequence, referred to as *confidence level* is defined as:

$$\texttt{confidenceLevel} = \frac{\texttt{number Of Sequence-Occurrences}}{\texttt{total Number Of Sequences With This Prefix}} \tag{4.1}$$

It should be noted that responses can be also fired *lazily*, i.e. after confirmation of intrusion, as opposed to preemptively (eagerly) as described above. This can be achieved, in our model, by setting the probability threshold value to 1. Lazy reaction to intrusion will trigger accurate response to an intrusion but can be risky as it allows more time to exploit a vulnerable system. On the other hand, eager response provides preemption but might be aggravated by a high probability of mistake (responding to falsely identified intrusions).

The probability threshold can be also viewed as tolerance level of system before a response must be deployed. For example, for security critical systems, where ensuring security and integrity is a priority, the tolerance level is low and as such the probability threshold is set to

a low value as well. On the other hand, for service based systems where ensuring persistence is important, the system may be tolerant to certain attacks and decide against preemptive response that can potentially shutdown the system. In either case, to deploy responses intelligently, response selection becomes one of the most important mechanism in our model. It is based on the following two steps: *effect identification* and *response strategy selection.*

4.3.2 Effect Identification

An attack is directed (a) to render one or more resources of the system unavailable or (b) to obtain unauthorized access to some resources. Response to an attack must be directed to protect the resources that are in danger of being unusable or compromised under that attack. Thus, to ensure safety of the system resources that might be effected by an intrusion it is necessary to identify the responses that can be the most effective defending the effected resources. To archive that we associate known intrusive patterns with the corresponding resources. Whenever the prefix of monitored sequence matches with a set of intrusive patterns, we say that the possible impact of the monitored sequence will be on the set of the resources associated with the matching intrusive patterns.

Furthermore, each attack to each resource is associated with possible responses (see Appendix A). For each resource, the partial ordering of responses is obtained and they are arranged in a form of a lattice. The ordering is done using the notion of *response cost.* Informally, responses with high cost has higher probability of stopping a possible attack but may lead to unavailability of resources. The top element of the lattice is the most severe response (highest cost) and the bottom element is the most passive response (lowest cost).

In summary, when a monitored sequence matches with the prefix of a set of known intrusive pattern, the corresponding effect in terms of resources is identified and from the resources, the possible set of responses is synthesized. The next step is to determine an optimal response strategy.

4.3.3 Response Strategy Selection

As noted above, intrusions effect resources and each resource is associated with a set of responses organized in a form of a lattice. It is necessary to select a subset of candidate responses from the corresponding lattices that are most likely to benefit the system, if deployed preemptively against the detected attack. The selection is performed by analyzing the *cost* information of damage to the system caused by (a) un-attended intrusion and (b) incorrect and/or severe response. The first is referred to as *damage cost* (`DC`) of intrusion and the second as *response cost* (`RC`). Both costs are computed based on the degree of loss of availability, confidentiality and integrity of system resources due to an intrusion.

4.3.3.1 Assigning Damage and Response Cost

One of the main challenges in developing our cost-sensitive response selection model and deployment mechanism is to accurately quantify with respect to the effected resources the damage and the response costs. In the following section, we present an overview of the process of assigning costs to intrusions and responses. Numeric values such as monetary values or percentages that correspond to some objective metrics are not always suitable, more effective solution based on relative measurements might be applied (Peltier, 2001). The relative measurements can be constructed based on company-specific security policies, existing threats, risk factors, etc. (Lee et al., 2000).

General classification of systems. Two important factors that guide the cost values are the *organizational role* of the system and the *resources available* in a considered environment. Often systems in an environment will be categorized into different broad classes such as public access systems, personal workstations, safety critical systems, business critical systems, etc.. The boundaries between these categories separate areas by acceptable risk, and therefore the cost for various damage types. For example, a public web server will be primarily focus on the availability of the service, while a business critical system will put more emphasis on data confidentiality.

In practice, setting the measurements of these costs is the responsibility of system administrator and is usually a manual process consisting of an informal series of questions such as "Will data be exposed?", "How critical is the confidentiality of the data?", "How concerned are we with data integrity?", and "Will service availability be impacted?". Although this gives an ad-hoc relative assessment of potential damage, it is not sufficient to yield a quantitative estimate.

Relative measurement of damage/response cost. Quantifying these assessments in a manner suitable for automated decision making requires a numeric abstraction of damage and a similar abstraction of system security goals (i.e., the security policy applied to the system).

This direction was taken by the Common Vulnerability Scoring System (CVSS) (Schiffman et al., 2004) that we adopted in our work for the estimation of possible system damage due to an attack (damage cost). CVSS allows to assess the *severity* of the existing software vulnerability and if exploited its *impact* on the system resources.

CVSS combines three metrics: base, temporal and environmental, with each metric producing a value ranging from 0 to 10 and a vector, representing compressed information used to derive this value.

1. The *base metric* represents the constant characteristics of a vulnerability such as impact on the availability, integrity and confidentiality of the resources due to an attack and the methods on how this vulnerability can be exploited (complexity, necessity of authentication etc.).

2. The *temporal metric* reflects the vulnerability features that change over time such as existence of an exploit, remediation level and confidence in vulnerability technical details.

3. Finally, the *environmental metric* indicates the risks on the company level such as potential damage to physical property, company productivity etc.

The approach we take for quantifying response cost is inspired by CVSS scheme. In our model, we independently develop the numeric abstractions of damage and system security goals

and combine them to provide a response-specific weighted cost value based on a system and policy specific metric.

Response Impact estimation. Our approach to response damage assessment is based on the following system security goals: *service availability*, *data integrity*, *data confidentiality*, *system performance*, *human resources* (such as administrator time) and *additional storage*. To estimate the impact of the specific response action, for each response we determine the security goals it may affect. For instance, *detailed logging* consumes storage resources, requires man-hours for review, and can also impact the performance.

To determine the numeric value of each response impact on a specific security goal, we separately consider security goals and the responses falling into each goal category. Specifically, all response actions effecting certain security goal are ordered based on their relative impact on the considered goal and assigned a numeric value according to their place in that category. These numeric values are evenly distributed between $1/n$ and 1, where n is the number of responses in the considered goal category.

For example, all responses effecting data confidentiality security goal are ordered based on their impact on the data confidentiality and assigned a score. Assume that the following responses fall into this category: *backup tampered with files*, *remote code execution: full code*, *remote code execution: partial code* and *migrate process to a different environment*. Based on the historical data and the experience the system administrator will rank these responses according to their impact on the data confidentiality. Let the ranking be the following (from the least severe impact to the highest impact): *backup tampered with files*, *migrate process to a different environment*, *remote code execution: partial code* and *remote code execution: full code*. Since there are four responses in this category, the numerical values are given as follows *backup tampered with files*: 0.25, *migrate process to a different environment*: 0.5, *remote code execution: partial code*: 0.75 and *remote code execution: full code*: 1.

Security Policy Goals weight. To abstract the security policy goals, the categories are ranked according to their importance (a value between 0 *no importance* and 1 *absolute*

importance) based on the overall goals of the system. These decisions can be based on the monetary values or other established business metric for the cost of failure to meet system goals (i.e. the estimated dollar cost of a confidentiality breach). In the case of a classified data processing system, for instance, data confidentiality may be a 0.9, indicating the extreme importance of this security facet for this system. If response team is employed to analyze and respond to potential threats on a continual basis, then controlling man-hour resources may rank a 0.05, indicating little concern over how much time is spent ensuring the security of the system.

Computation of Response Cost value. Using these two abstractions, the overall impact rating of the response per category is computed by multiplying the category importance value by the assigned impact for that category. The sum of the overall impact ratings for all categories the response affects is the overall impact rating for the response or in other words response cost which we define as *Response Factor* (RF). Formally, RF value of response r is computed as follows:

$$RF_r = \frac{1}{\texttt{MAX}(RF_r)} * \sum_{c \in \texttt{PolicyCategory}} impact_{r,c} \times weight_c \tag{4.2}$$

where $impact_{r,c}$ is the estimated impact of the response r on security category c, and $weight_c$ is the importance value assigned to category c. $1/\ \texttt{MAX}(RF_r)$ is the normalization term necessary to ensure range of RF value from 1 to 0.

For example, the RF value for response *backup tampered with files* for a public web server (B.5) can be computed as follows : $RF_{backupfiles} = 0.25 \times 0.1 + 0.143 \times 0.7 + 1 \times 0.1 = 0.225$. This value is then normalized using the maximum RF value for this system (in this case, *complete network isolation* which has a pre-normalized value of 1) to obtain the final value of 0.225.

Computed RF values are provided for systems representing a public web server, sensitive data storage system, and a business critical system (see Tables B.5, B.6, B.7).

4.3.3.2 Identifying the Candidate Responses

The suitable costs to intrusions and responses are assigned by the system administrator with respect to the specific security policies of the system under consideration before the intrusion response system deployment.

Based on these costs a subset of responses is selected to ensure that the damage due to possible incorrect deployment of response does not overshadow the benefits of early response. The selected responses conform the following condition

$$\mathtt{DC} \times \mathtt{confidenceLevel} > \mathtt{RF} \tag{4.3}$$

where `DC` is the damage cost corresponding to the possible intrusive behavior whose prefix matches with the monitored sequence, `RF` is the response factor that indicates response cost of a response associated with the resource being effected by the intrusive behavior, and `confidenceLevel` is the probability that the monitored sequence indeed leads to some intrusive behavior.

4.3.3.3 Evaluating Response Effectiveness

Though all the candidate responses satisfy the Equation 4.3, the effectiveness of the responses to adequately contain an attack will vary. We introduce the notion of response effectiveness based on the prior experience of deploying the considered response and the probability of successfully identifying the intrusion. Let r be a response corresponding to a resource being effected by a known intrusion I. Let $Pr(I)$ be the probability that the intrusion I will result from the currently monitored pattern which matches with the prefix of I. Finally, let `SF` be *Success Factor* defined as an expected success of the response r in handling the intrusion I and computed as follows:

$$SF = Pr_{success}(r) \times S_{level} \tag{4.4}$$

where $Pr_{success}(r)$ is the percentage of successful deployment of the response r in handling the intrusion I and S_{level} is the *Success level* that denotes the degree indicating how successful is the response r in handling the intrusion I.

Generally, the common approach to response success quantification is the differentiation of two response outcomes: *response success*, if the deployed response archived expected result (e.g., blocked intrusion, collected the data), and *response failure*, otherwise. One of the limitations of this approach is inability of the response system to distinguish intrusion steps disabled as a result of response which may later result in a response wastefully deployed to counter some of the disabled intrusive behaviors.

Another limitation lies on the response side, as the intrusion response is often represented by the set of multi-targeted actions labeling such response strategy as failed essentially indicates that none of the response actions succeeded, which also underestimated the value of the deployed response.

While for the purpose of this work we adopt a common view of the response outcome, considering more complex strategies employing partial response success is the important direction of the future work. As such we consider *Success level* as binary variable which takes a value of 0 in case of the response failure and a value of 1 if the response succeeds.

Then, the effectiveness of the response, $Ef(r,I)$ is computed as follows:

$$Ef(r,I) = [\ Pr(I) \times S_{gain}] \ - \ [\ (1 - Pr(I)) \times RF] \tag{4.5}$$

Therefore effectiveness is the difference between the response being *correctly* (accurately identifying the intrusion for which the response is selected) and *successfully* deployed (*Success Factor*) and the response being incorrectly deployed resulting in (additional) damages caused by unsuccessful deployment of the response (denoted by `RF` or *Response Factor*). The above response effectiveness computation is based on the utility theory (see (Coyle, 1972) for details).

4.3.3.4 Deploying the Most Effective Response

Finally, from the set of candidate responses, a response is selected which maximizes the chances of containing the possible intrusion. This is realized using the effectiveness estimation of the responses. The goal is to obtain the subset of candidate responses which can protect all possible resources effected by all the potential intrusions and deploy the one from the subset

which has the highest effectiveness.

The above problem can be reduced to cost-based satisfiability problem. Let monitored execution sequence be E. Assume it matches with the prefix of known intrusive patterns which effect n resources $R_1, R_2, \ldots, R_n$. Further let a resource R_i be associated with m responses: $r_{i_1}, r_{i_2}, \ldots, r_{i_m}$; any of r_{i_j} $(1 \leq j \leq m)$ is capable of protecting the corresponding resource R_i from the associated attack patterns $I_{i_1}, I_{i_2}, \ldots, I_{i_l}$ (R_i is being effected by l known attack patterns and E matches with prefix of each of these attacks). Therefore, for intrusions effecting resource R_i, a disjunctive formula $\bigvee_{j \leq m} r_{i_j}$ denotes the candidate responses. Finally, for intrusions effecting resources $R_1, R_2, \ldots, R_n$, the candidate responses can be obtained from the satisfiability of the formula:

$$\bigwedge_{i \leq n} [\bigvee_{j \leq m} r_{i_j}] \tag{4.6}$$

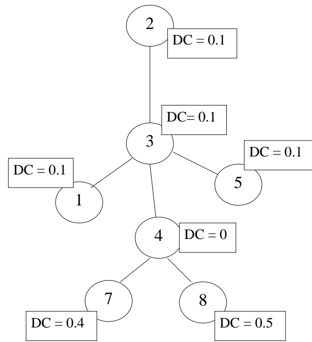
Proceeding further, for each i, response r_{i_j} corresponds to attack patterns $I_{i_1}, I_{i_2}, \ldots, I_{i_l}$ via the resource R_i. The effectiveness of the response r_{i_j}, denoted by $Ef(r_{i_j})$ for handing all these attacks is

$$\sum_{k=1}^{l} Ef(r_{i_j}, I_{i_k}) \tag{4.7}$$

The set of responses that are likely to be most effective against the attacks whose prefix matches with execution sequence E is, therefore, the set of r_{i_j}, denoted by Δ, which satisfies the Equation 4.6 and maximizes the objective function $\sum_{j=1}^{m} Ef(r_{i_j}, I_{i_k})$. The deployed response is the one which has the highest Ef value in Δ.

4.3.4 Adaptability

We consider the adaptability of the response mechanism in terms of its ability to adjust to the changing environment according to the response decisions previously issued by the system, in particular, success and failure of the triggered before responses. In the event that the selected response succeeds in blocking the attack, its *Success Factor* is automatically increased by one, otherwise, if the response fails to stop or divert a potentially anomalous pattern, its *SF* is decreased by one to reflect this result. Note that the update of the *SF* in case of failed response is performed after the monitored sequence fully matched the corresponding anomalous pattern to exclude the possibility of an error. This approach allows to adjust response selection



#	Sequence	# of repetitions of each attack-pattern	RF	SF	Response #
1	2, 3, 1	2	0.9	1.0	r11
			0.3	0.5	r12
2	2, 3, 5	1	0.5	0.8	r21
3	2, 3, 4, 8	2	0.4	0.7	r31
			0.2	0.6	r32
			0.7	0.5	r33
4	2, 3, 4, 7	1	0.7	0.8	r41
			0.1	0.4	r42

(a) Example setting

Pattern seen: $\langle 2 \rangle$ or $\langle 2, 3 \rangle$

Sequence	Confidence level
2, 3, 1	(2/6) 0.33
2, 3, 5	(1/6) 0.16
2, 3, 4, 8	(2/6) 0.33
2, 3, 4, 7	(1/6) 0.16

(b) Steps 1 & 2

Pattern seen: $\langle 2, 3, 4 \rangle$

Sequence	Confidence level
2, 3, 4, 8	**(2/3) 0.66**
2, 3, 4, 7	(1/3) 0.33

(c) Step 3

Select the responses

#	Sequence	Confidence level	DC	RF	SF	Response #	DC vs. RF
3	2, 3, 4, 8	(2/3) 0.66	0.7	0.4	0.7	r31	(0.7*0.66)>0.4
				0.2	0.6	r32	(0.7*0.66)>0.2
				0.7	0.5	r33	(0.7*0.66)<0.7 → **do not respond**
4	2, 3, 4, 7	(1/3) 0.33	0.6	0.7	0.8	r41	(0.6*0.33)<0.7 → **do not respond**
				0.1	0.4	r42	(0.6*0.33)>0.1

(d) Step 4&5

Compute EV

#	Sequence	Confidence level	RF	SF	Response #	EV
3	2, 3, 4, 8	(2/3) 0.66	0.4	0.7	r31	0.66 * 0.7 + (1-0.66)* 0.4 = 0.326
			0.2	0.6	r32	0.66 * 0.6 + (1-0.66)* 0.2 = 0.464
4	2, 3, 4, 7	(1/3) 0.33	0.1	0.4	r42	0.33 * 0.4 + (1-0.33)* 0.1 = 0.199

(e) Step 5

$$\textit{MinCostSAT formula:}\ (r31 \vee r32) \wedge (r42) = (0.326 \vee 0.464) \wedge (0.199)$$

(f) Step 6

Figure 4.2 Example demonstrating the response strategy selection process

for the future occurrences of this sequence and consequently, effectively adapts our response system to the changing environment.

4.3.5 Illustrative example

Let sequences presented in Figure 4.2(a) together with their graphical representation be a part of specifications representing anomalous patterns. Assuming that complete monitored pattern is $\langle 2, 3, 4, 8 \rangle$ and `probability threshold` $= 0.5$ we determine optimal response strategy following the steps defined in Figure 4.1:

1. **Pattern observed:** $\langle 2 \rangle$. We compute confidence level for all sequences starting with prefix 2 (Figure 4.2(b)). Since none of the considered sequences has confidence level exceeding `probability threshold` we continue to monitor system executions without taking any action.
2. **Pattern observed:** $\langle 2, 3 \rangle$. Now confidence level is computed for sequences starting with prefix $\langle 2, 3 \rangle$ (Figure 4.2(b)). Similarly, confidence level of the considered sequences has not reached `probability threshold`.
3. **Pattern observed:** $\langle 2, 3, 4 \rangle$. Now confidence level is computed for only two sequences starting with prefix $\langle 2, 3, 4 \rangle$ (Figure 4.2(c)). Since confidence level $>$ `probability threshold`, we need to identify the resources possibly affected by intrusive patterns with prefix $\langle 2, 3, 4 \rangle$ and the corresponding set of responses. For example purposes let us assume that responses provided in the Figure 4.2(a) represent the corresponding to the sequences response actions. Thus, a set of candidate responses that we will consider further includes $r31, r32, r33, r41, r42$.

   As our next step we select response actions from this set that follow the condition (4.3)(Figure 4.2(d)). Two response actions $r33$ and $r41$ do not comply with this requirement which means that deploying them may cause more damage, and hence these responses are eliminated from the further consideration.

   The rest of responses will constitute a satisfiability formula. We compute EV values for the selected set of responses (Figure 4.2(e)) and build a *cost-satisfiability* formula

(Figure 4.2(f)). Finally, an optimal response strategy consisting of two responses $r31$ and $r42$ is deployed.

4.3.6 Preliminary Simulation Results

Attack	**Description**	**Number of instances**	**Costs**
Eject attack	exploits a buffer overflow vulnerability in 'eject' program.	16	DC = 1.0 RF = 0.4
Fdformat attack	exploits a buffer overflow using the 'fdformat' UNIX system command.	5	DC = 1.0 RF = 0.4
Ftp-write attack	exploits FTP server misconfiguration. Remote FTP user creates .rhosts file in world writable anonymous FTP directory and obtains local login.	2	DC = 0.5 RF = 0.2

Table 4.1 Attack descriptions (MIT, 1998).

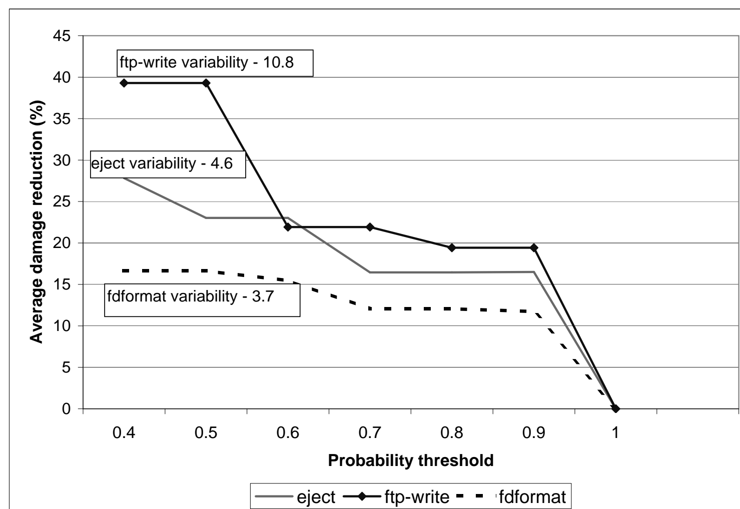


Figure 4.3 Average damage reduction (fdformat attack).

Data. We evaluated our model using the 1998 DARPA/Lincoln Lab offline evaluation data. We used Basic Security Module (BSM) audit records representing system calls and corresponding network traffic data indicating the attack starting points. For our experiments we used two User-to-Root attacks that exploit buffer overflows: *eject* and *fdformat* and one Remote-

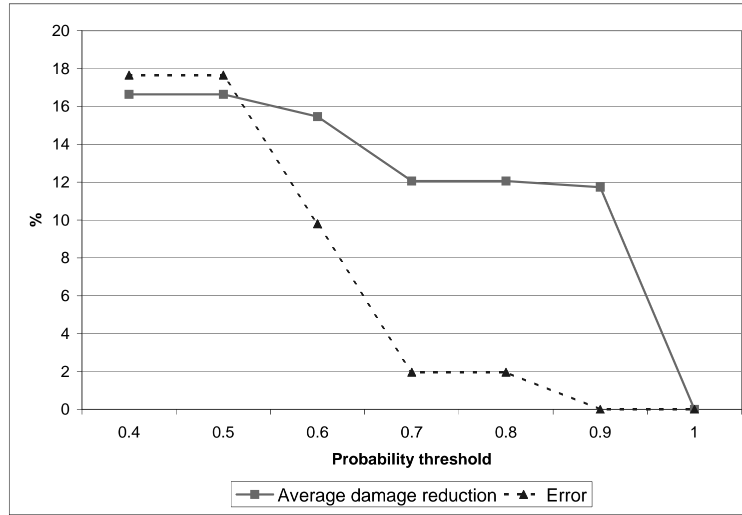


Figure 4.4 Average damage reduction vs. error (fdformat attack).

to-Local attack: *ftp-write*. The descriptions of these attacks and corresponding damage and response costs according to the attack taxonomy by (Lee et al., 2000) are given in Table 4.1. Each state in an attack trace is associated with a damage cost; the overall damage cost of the trace being the sum of damage costs of the states in the trace. Although our model allows associating multiple response actions with one anomalous sequence, for evaluation purposes we experimented with one response per sequence.

Results. Since the primary goal of this work is the intrusion response model we assume that repository of anomalous patterns is provided by the intrusion detection framework. We performed several experiments focusing on the effect of the cost-sensitive modeling and the preemptiveness of the response in our model. As a primary criterion we defined *damage reduction* metric that shows the difference between damage cost incurred by a full attack and a damage cost caused by the prefix of the attack sequence (at the time of the response). These measurements will be used to compare the systems without the cost-sensitive modeling with the one equipped with our response approach.

Figure 4.3 shows the average damage reduction for *eject, fdformat* and *ftp-write* attacks

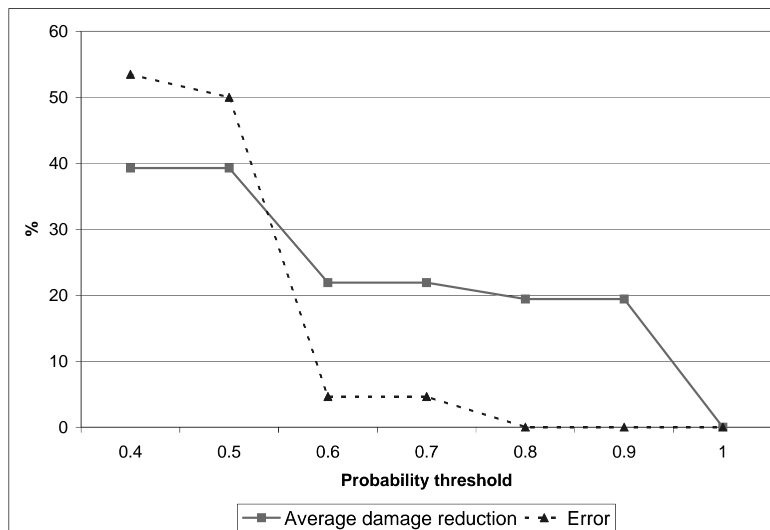


Figure 4.5 Average damage reduction vs. error (ftp-write attack).

with respect to the `probability threshold`. The most significant damage reduction for all attacks occurs with the lowest value of `probability threshold` (0.4). A high value of `probability threshold` forces system to require more confirmation of the attack occurrence before a response can be deployed and, consequently, may increase the damage incurred by the late response. As such `probability threshold` = 1.0 does not carry any damage reduction and thus corresponds to the "cost-insensitive" system.

It is important to note that all considered attacks have similar damage reduction patterns with only curve difference. The difference in the graph can be correlated to the average variability in attack-patterns with identical prefix. The variability is computed as the weighted mean of the `prefixLength` valuations where the weights are defined by the frequency or number of attack-patterns with the same prefix of a specific `prefixLength`.

$$Variability = \frac{1}{N} \sum \texttt{(prefixLength * numOfSeq)} \tag{4.8}$$

In the above N is the total number of unique sequences of attack-patterns. A high value of variability indicates a high number of sequences with unique prefixes. Clearly, in this case

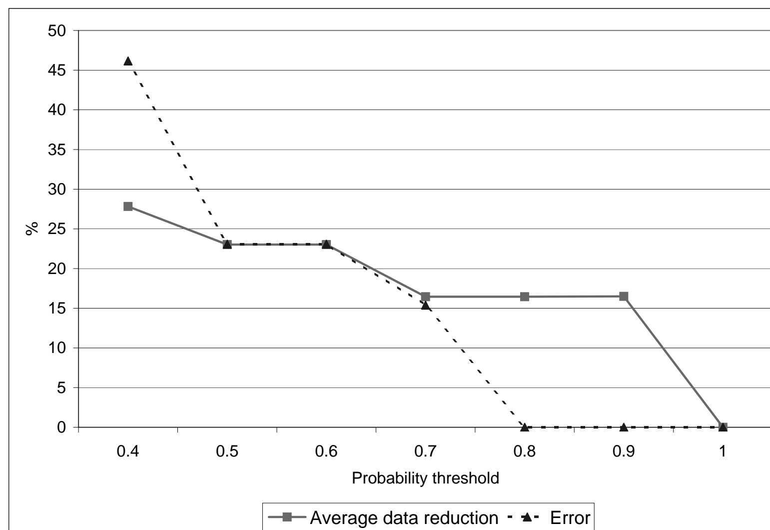


Figure 4.6 Average damage reduction vs. error (eject attack).

the correct attack pattern can be recognized early which would allow less damage to occur and therefore, would correspond to the higher damage reduction.

Figures 4.4 and 4.5, 4.6 present a damage reduction against two types of error:

- *Early response error*: Low threshold may force preemption long before the distinguishing aspects of different attack-patterns are seen (prefix of multiple patterns matches the sequence being monitored). As a result, monitored sequence may eventually end up in one attack-pattern while the response, selected and deployed preemptively, may correspond to some other attack. We will focus primarily on these type of errors; however it must be noted that any preemption can lead to such problems – the main challenge is to allow flexibility and capability to fine tune thresholds to vary preemption depending on prior results and security requirements.

- *Never-seen-before sequence*: when monitored sequence represents a novel pattern, although matches a prefix of some existing anomalous sequence. We claim that intrusion detection system is responsible for detecting new attacks and their addition to the attack-

pattern repository. Responses can be attached only after the attacks are identified by the intrusion detection system.

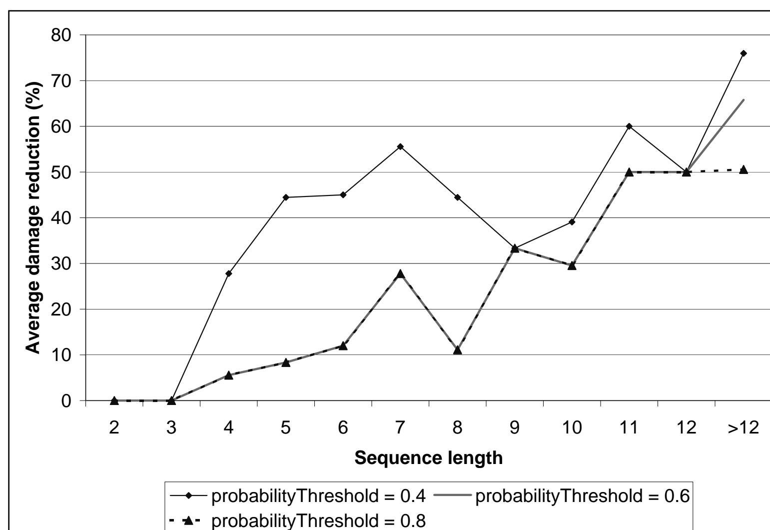


Figure 4.7 Average damage reduction vs. sequence length (ftp-write attack).

As expected, the results show that the error decreases with the increase of `probability threshold` value. With the delay of response selection process increases a confidence in some attack and consequently increases the accuracy of prediction of anomalous sequence.

Figure 4.7 presents a damage reduction from the perspective of average sequence length on the example of the *ftp-write* attack. Generally, damage reduction increases with the increase in sequence length. We can also note that this increase is more gradual for higher values of `probability threshold` (0.6, 0.8) and becomes very rapid for `probability threshold` = 0.4. This again confirms the observation that low `probability threshold` gives the highest amount of damage reduction. There are two sudden changes in graph patterns: one at sequence of size 8 and another at sequence of size 9. They represent two data extremes. Sequences of length 8 are almost identical with slight differences (low variability) at the end. Therefore, to

reach specified `probability threshold` system waits until almost the end of these patterns, resulting in low damage reduction. On the other hand, a sequence of size 9 has a very unique pattern, allowing system to recognize it early with little influence of `probability threshold` value.

For the above experiments, it can be inferred that the systems focused on the significant damage reduction with the ability to tolerate higher error level can set up a lower probability threshold while systems more oriented on the accuracy of the deployed response should lean towards `probability threshold` value that is closer to 1.0.

CHAPTER 5. Implementation and experiments

Previous chapters presented the design of the detection and the response components of the proposed IDR system. This chapter describes the implementation details of these components, explains how the normal profiles of the programs are generated, what anomalous behavior can be detected and examines several example of malicious program behavior.

5.1 Implementation overview

Our system implementation consists of the three components: *Hybrid interposition of system calls component (HISC)*, *Detection module* and *Response module* (Figure 5.1).

Hybrid interposition of system calls component (HISC) (Uppuluri et al., 2006) is a system call interception mechanism built for Linux architecture. It serves as an additional layer between the kernel and our IDR system by intercepting all system calls made by the processes in the kernel and delivering them to the user space. The HISC architecture is presented in Figure 5.2 (Uppuluri et al., 2006). It includes the following major components:

- HISCD: the user-level daemon responsible for intercepting all newly created processes.
- URT: the user-level monitor, forked by HISCD for a new process, invokes the execution of UDE facility.
- UDE: the user-level detection engine that intercepts the system calls.
- KRT: the kernel-level runtime environment that mainly functions as a control center and intermediate node for communication between KDE, HISCD and URT.
- KDE: the kernel-level detection engine that intercepts the system calls at the kernel level.

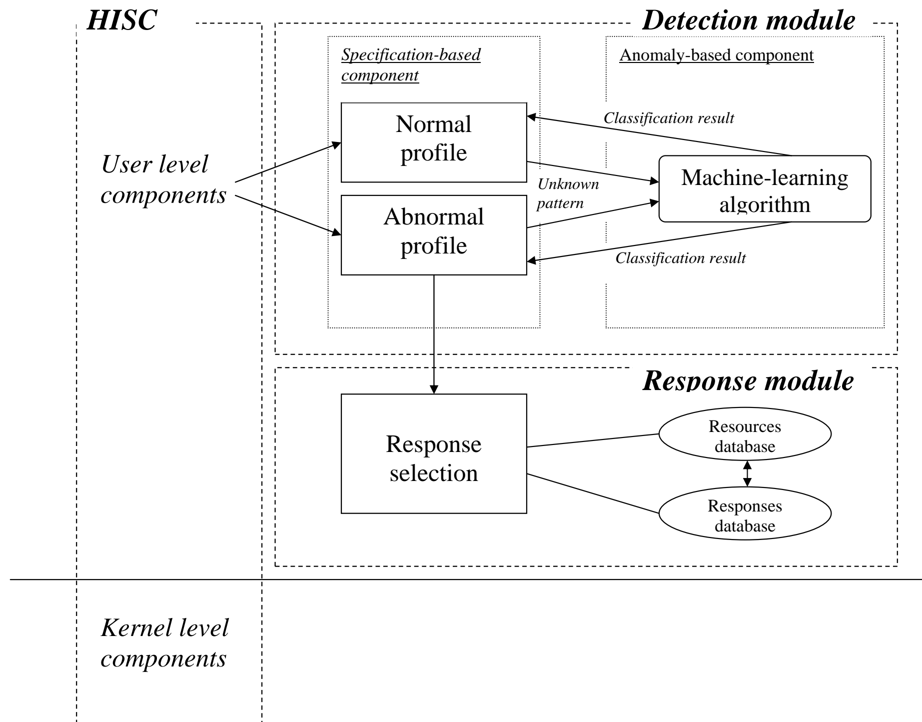


Figure 5.1 Architecture of IDR system.

- Device Driver: mainly used for communication between URT and KRT components.

Detection module is the main component that directly interacts with the UDE module of HISC infrastructure. It is responsible for monitoring and processing the system calls intercepted by the kernel-level components of HISC. The observed behavior is matched with the normal and abnormal profiles contained in the `Exact` graphs. In case an unknown sequence is encountered, the detection module processes the pattern through the classifier (i.g. SVM classifier) and makes necessary updates in the `Exact` graphs. If the observed pattern represents abnormal behavior, detection module invokes the response component and redirects to it the suspicious sequence.

Response module is responsible for the response selection mechanism that is invoked

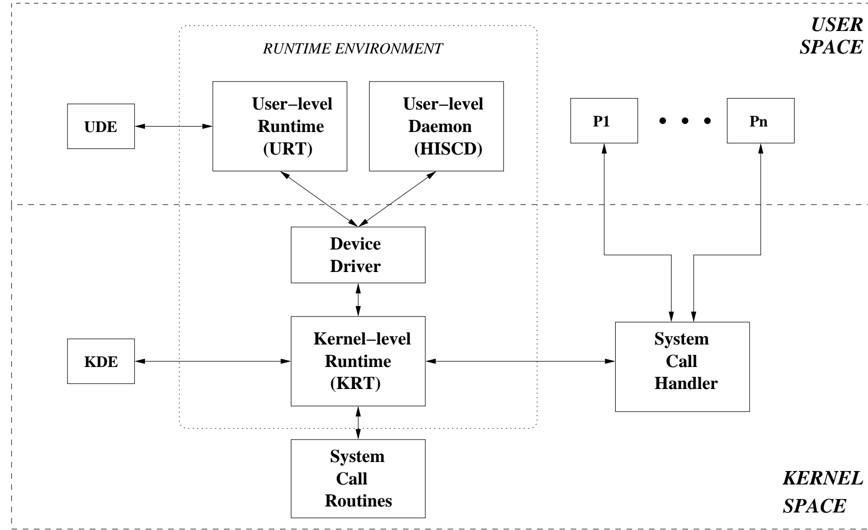


Figure 5.2 Architecture of HISC

when an observable behavior matches abnormal program's profile. This component determines the existence of the necessary conditions for the preemptive response deployment and selects the most effective response action based on the risks involved in deploying considered response and the potential damage from the monitored intrusive pattern. Current implementation of the response component only provides the selection of the optimal response action. It does not include the cost-satisfiability modeling of the responses and does not provide response deployment. Currently the optimal response action is selected based on the value of the effectiveness of the response, while the success or failure of selected response action is determined randomly.

Parameter	Default value
probabilityThreshold	0.4
windowSize (SVM algorithm)	3
delay	10

Figure 5.3 The parameters of our IDR system.

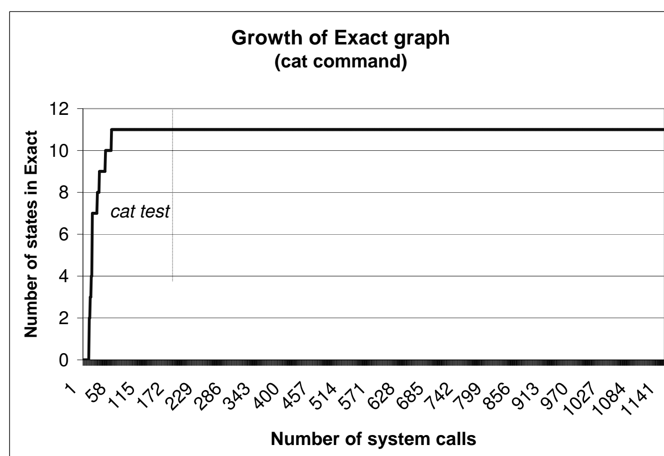
5.1.1 Implementation details & parameters.

All components of the IDR system were implemented using C/C++ language. In our IDR system we used HISC version for Linux kernel 2.4-18 (Uppuluri et al., 2006). The anomaly-based classifier in the detection module was implemented using libsvm tool, version 2.84 (Chang and Lin, 2001).

Table 5.3 summarizes the parameters and their default values used in our implementation. *ProbabilityThreshold* parameter indicates an acceptable level of confidence that some attack is in progress and a response action should be triggered. *WindowSize* is the size of the sliding window used to break up variable-length `Exact` patterns into sequences of fixed length for classification by the machine-learning algorithm. *Delay* parameter shows how many system calls system waits before processing the monitored sequence in case the first system call does not have any repetitions in this pattern (Section 3.1.2, algorithm 3.5).

5.2 Experiments

5.2.1 Building normal behavioral profile



Other options of *cat* command (such as *cat -b, cat -e, cat -E, cat -n, cat -s, cat -t, cat -T, cat -v, cat –help*) have no impact on its normal profile.

Figure 5.4 Growth of the number of states in `Exact` graph with legal patterns.

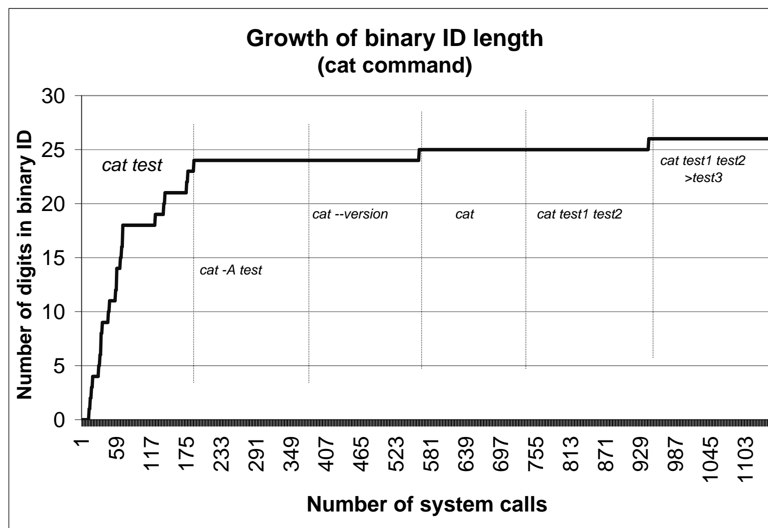


Figure 5.5 Growth of the length of binary ID associated with legal patterns.

As we previously discussed, the detection module maintains two `Exact` structures: a structure containing normal patterns and abnormal or known intrusive sequences. If the monitored sequence does not match any of the existing patterns, anomaly-based classification is triggered. According to the result of the classification, the new pattern is recorded in the corresponding `Exact` graph. To be able to rely on the anomaly-based classification, the algorithm needs a training set of patterns. The presence of labels indicating normal or abnormal sequence, as well as the existence of the intrusive patterns in the training set depends on the actual machine learning algorithm. In the implementation of our detection model we experimented with two-class SVM, trained on the instances of normal and abnormal behavior.

In this section we focus primarily on the profiles of programs' normal behavior and the ability of our system to capture them.

In the evaluation of our model we used the following Unix utilities:

- The `cat` utility concatenates and prints files reading them sequentially and writing them to the standard output (cat, 2007).

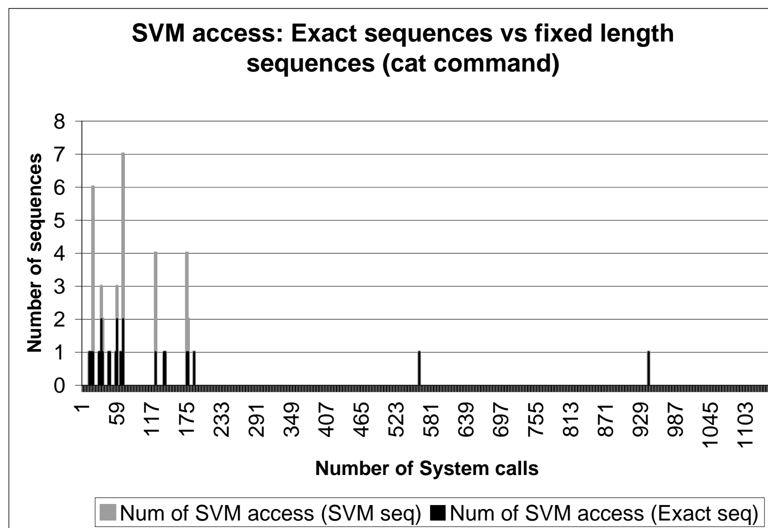


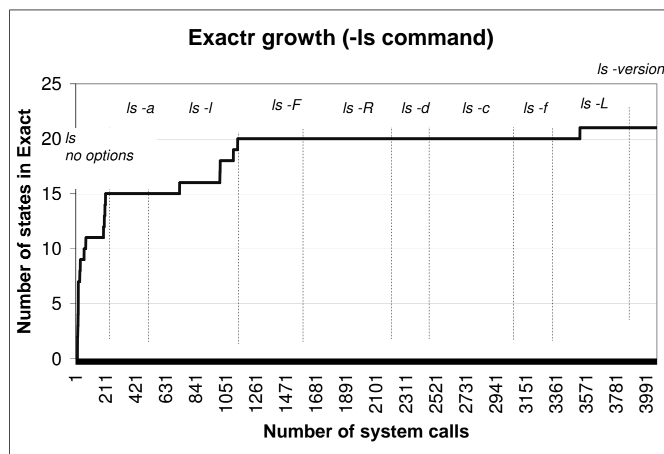
Figure 5.6 SVM algorithm invocation (`cat` utility).

- The `ls` utility lists the files in the current working directory (ls, 2007).
- The `mount` utility requests file system found on a device to be attached to the general file hierarchy (mount, 2007).
- The `whoami` prints an effective user name (whoami, 2007).

Utility name	Version	Length of binary ID (# of distinct `Exact` sequences)	# of distinct `Exact` states
ls	fileutils package, version 4.1	56	21
cat	textutils package, version 2.0.21	26	11
mount	version 2.11n	21	17
whoami	package sh-utils, version 2.0.11-5	32	21

Table 5.1 Normal profiles of the used UNIX utilities.

Figures 5.4, 5.5, 5.7 and 5.8 show the population the `Exact` with normal patterns for `cat` and `ls` utilities. `Exact` with abnormal(anomalous) patterns in these cases remains empty since detection engine only encounters normal behavior of these utilities.



Other options of *ls* command (such as *ls -A, ls -b, ls -B, ls -c -l, ls -C, ls -D, ls -G, ls -h, ls -i, ls -I, ls -k, ls -m, ls -n, ls -N, ls -o, ls -p, ls -q, ls -Q, ls -r, ls -s, ls -S, ls -t, ls -T, ls -u, ls -U, ls -v, ls -w, ls -x, ls -X, ls -1, ls –help*) have no impact on its normal profile.

Figure 5.7 Growth of the number of states in `Exact` graph with legal patterns.

Figures 5.4 and 5.7 show the growth of the `Exact` graph in terms of system calls generated by these utilities. These figures correspond to the number of the states in the `Exact` graph. Since the behavior of `ls` is more variable than the behavior, of `cat`, the number of states in the final normal profile is much higher (21 state) compared to the size of `Exact` for `cat` utility (11 states).

Figures 5.5 and 5.8 present the growth of the length of binary vector associated with the `Exact` graph with normal patterns. In other words, this shows the number of distinct variable-length sequences in `Exact`. Similarly, the size of the binary id of the normal profile for `ls` utility is much larger (56 digits) than the one for `cat` (26 digits). The summary of the normal profiles of these utilities is provided in Table 5.1.

Figures 5.6 and 5.9 demonstrate the classification of system call patterns produced by `cat` and `ls` utilities through the anomaly-based algorithm. Specifically, the graphs show how often the new sequence needs to be classified through the SVM algorithm. In other words, these graphs show how often the underlying SVM algorithm is invoked. Both graphs show two perspectives: the view of the classification from the `Exact` side and the view from the SVM algorithm side. Recall that one unknown variable-length sequence needs to be broken into

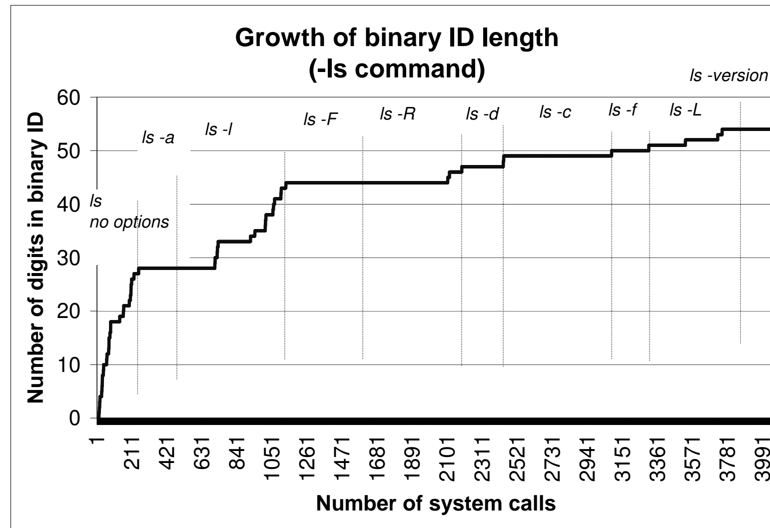


Figure 5.8 Growth of the length of binary ID associated with legal patterns.

one or more fixed-length sequences to be processed by SVM algorithm. Thus, classification of a new `Exact` pattern might result in a number of invocations of SVM. This can be clearly seen on the graphs: the number of fixed-length sequences is much higher than a number of variable-length patterns. This difference also impacts the performance of our system during the monitoring of program behavior as shown in Table 5.10.

The performance experiments were run on VMware Server 1.0.3 running Red Hat Linux 7.3 (Valhalla) Linux distribution with kernel 2.4.18-3 running on IBM Lenovo T61 (Intel Core 2 Duo CPU T7300@2GHz, 984 RAM). Tests were run for 20 iterations. Each command was run without options. *Partially populated* `Exact` refers to the `Exact` graph containing normal patterns for Unix utilities without options. *Backend SVM* denotes an anomaly-based algorithm triggered by our system for classifying unknown patterns.

Table 5.10 shows the execution time of our system without and with the existing normal profile. It is clear that the lack of any knowledge on the normal behavior of the program impacts the performance of our system. Specifically, the absence of the normal profile increases the

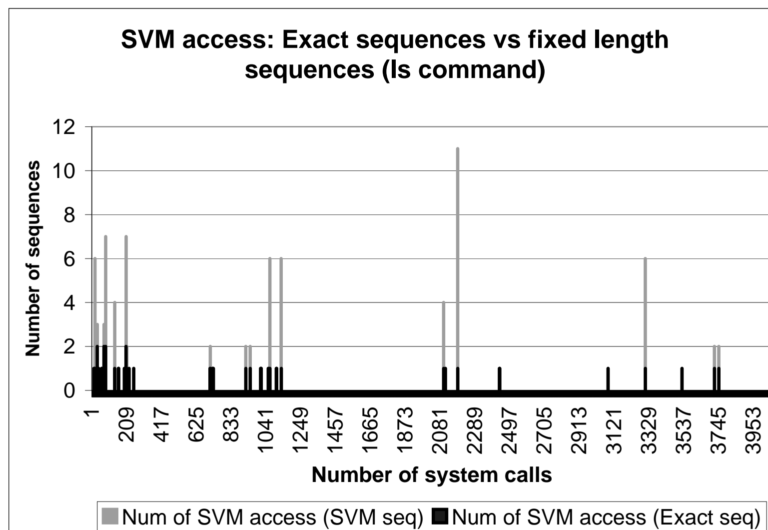


Figure 5.9 SVM algorithm invocation (`ls` utility).

running time on average by 30%. At the same time, depending on utility the running time may be significantly higher. For example, in case of `cat` utility the running time (73 milliseconds) almost doubled, compared to the case with partially populated `Exact` (46 milliseconds).

This increase in the system run time with empty profile is primarily caused by the classification of the unknown patterns using the machine-learning algorithm. The average increase of the time necessary for backend SVM is almost tripled, compared to the SVM use with partially populated `Exact`.

Generally, the overhead that our model brings to the system while monitoring its behavior is 70.25 milliseconds, which is computed as the difference between the time of partially populated `Exact` and running time of the utility. This overhead incorporates time required for several operations: system calls interception by the HISC, system calls processing through detection and response components and finally, the time necessary to wrap the system calls for the classification and retrieve the SVM classification result. Current implementation employs SVM as a stand alone classifier which requires several IO operations. Optimizing classification to avoid costly file operations would lower the overall overhead of our model, thus, is being

Utility	Running time	Partially populated `Exact`		Overhead	Empty `Exact`	
		Total time	Backend SVM running time		Total time	Backend SVM running time
ls	24 (12.7)	93.5 (14.6)	5 (6.1)	69.5	103 (28.5)	35 (18.8)
cat	4.5 (5.1)	46 (15.7)	8.5 (8.7)	41.5	73 (19.2)	19.5 (9.9)
mount	6.5 (6.7)	108.5 (14.6)	13 (10.3)	102	146 (22.3)	31.5 (16.3)
whoami	4.5 (6)	72.5 (13.7)	12.5 (8.5)	68	97.5 (12.9)	25 (8.9)

Figure 5.10 Mean CPU running time of Unix utilities in millisec (Standard deviation is given in parentheses).

considered as one of the future work directions.

5.2.1.1 Abnormal behavior

Response action	RF	SF	Illegal read (`ls`)	Illegal shell access (`whoami`)	Buffer overflow (`ls`)
process termination	0.9	0.8	✓	✓	✓
process delay	0.6	0.7	✓	✓	✓
process restart	0.3	0.5	-	-	✓
memory isolation	0.2	0.6	-	✓	✓
restore backup file	0.25	0.7	-	-	-
backup tampered with files	0.25	0.1	✓	-	-
delete tampered with files	0.5	0.6	✓	-	-
file isolation	0.8	0.8	✓	-	-
detailed logging	0.28	0.1	✓	✓	✓
generate report	0.07	0.15	✓	✓	✓
generate alarm	0.1	0.2	✓	✓	✓

Table 5.2 The response actions suitable to contain considered attacks according to the Appendix B.

Our system relies on the existence of the profiles of normal and abnormal behavior of the programs. The profile of abnormal behavior is not necessary to accurately detect abnormal behavior; however, its existence is crucial for response component of the system.

Once these profiles are built, the system monitors the programs' behavior for anomalous changes. In this subsection we focus on the abnormal behavior of the programs.

What type of abnormal behavior can be detected Our system is pattern-based, i.e it detects the known patterns of normal or abnormal behavior represented by sequences of system calls. Therefore, if the anomalous behavior does not manifest itself through the change in the program's system call behavior, it will not be detected. Naturally, some types of intrusions known for their ability to disguise the behavior as normal remain hidden from our system unless their intrusive behavior will cause program to behave differently.

One of these types of intrusions are *mimicry attacks*, that, in essence, are mimicking the normal behavior of the program, thus making their detection a very challenging task (Wagner and Soto, 2002). The examples of such *mimicry attacks* include exploiting normal although vulnerable application behavior or embedding irrelevant system calls into a malicious code, making it look as a legitimate system call sequence (Wagner and Soto, 2002).

Another type of intrusions that are hard to detect using system-call pattern-based IDS are buffer overflow attacks. Generally, buffer overflow occurs when a program writes data with size larger than a space allocated for this data. Often buffer overflow is a first step of the attack that allows an attacker to overwrite data responsible for program flow control and transfer the control over program to a malicious code (Fayolle and Glaume, 2002; Simon, 2002; Ogorkiewicz and Frej, 2002). Since buffer overflows do not cause change in the programs behavior on the system-call level during their initial step, their detection and prevention usually involves methods such as bound-checking (Jones and Kelly, 1997), static analysis of source code (Wagner et al., 2000; Ganapathy et al., 2003) and dynamic run-time analysis of the program (Lhee and Chapin, 2002). We will explore the behavior of our system for this type of attacks on the example of integer overflow exploit.

Detection of new program behavior: `cat` vs `mount`. In this experiment we explored the behavior of the detection engine of our IDR system when encountering the behavior that differs from the sequences contained in the normal `Exact` graph. Although this new behavior is not necessarily malicious, from the detection engine perspective it is abnormal.

For evaluation purpose we used `mount` utility as a program exhibiting anomalous behavior. Classifier in this case was only trained with the data from normal executions of `cat` utility.

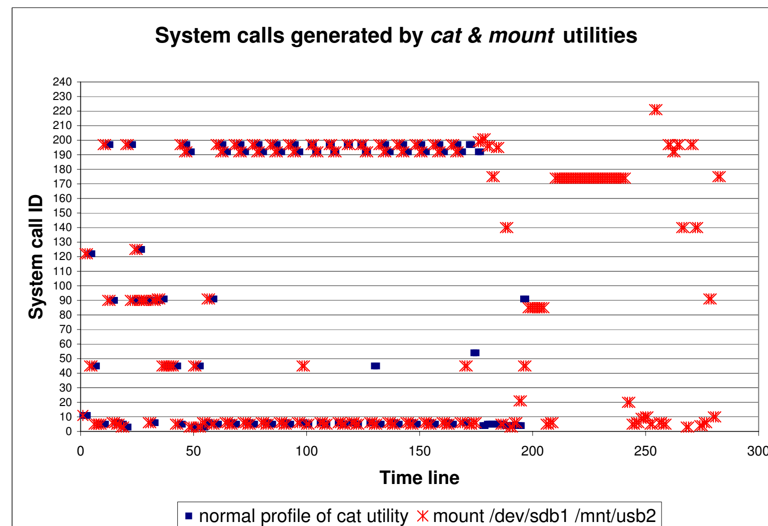


Figure 5.11 System calls generated by *cat* and *mount* utilities.

We selected `mount` for comparison purposes, mainly due to the nature of system calls generated by these two utilities. As Figure 5.11 shows there is a significant difference in the system calls generated by `cat` and `mount` utilities. However, this difference appears only after approximately 200 system calls. In the beginning of each run, calls produced by these utilities at the same time units are similar, and thus sequences of variable-length produced by `Exact` from this stream of system calls are also similar. Once the behavior of these utilities becomes different, the detection component with `Exact` graphs containing only normal profile for `cat` utility will recognize the execution sequences of `mount` utility as abnormal. This behavior can be clearly seen on the Figures 5.12 and 5.13. Sequences not matching normal profile of `cat` utility are recorded as anomalous in the `Exact` with abnormal patterns.

Detection of known program behavior: illegal read. As the examples of modified normal behavior of the program we experimented with two commonly used programs `ls` and `whoami`.

The motivation behind such experiments is the *trojan horse* scenario in which intelligent

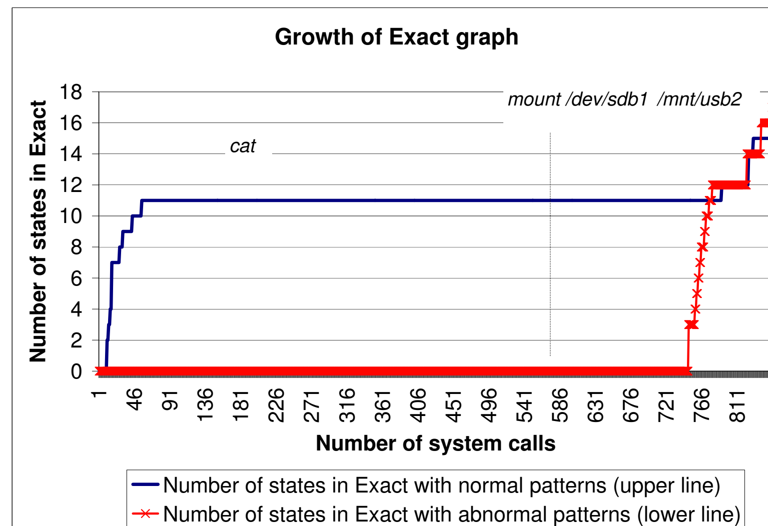


Figure 5.12 Growth of the length of binary ID associated with normal and abnormal patterns.

attacker gains access to the system and modifies or leaves a modified version of the program with some hidden (often malicious) features allowing an intruder to control the system after attacker leaves. Usually *trojan horse* programs are masked under ordinary programs such as `ls, find`, computer clock setting programs, games, etc (Dittrich, 2002). Often programs necessary for break-in and cover-up steps are bundled together for convenience and referred to as *root kits*.

The fundamental problem in detecting the *root kits* is uncertainty about the programs that can be trusted (Wikipedia, 2007). We experimented with detection of the novel patterns in the behavior of known programs, specifically `ls` and `whoami` (whoami, 2007).

As an example, the following fragment has been added to the `ls` *main*() body of the program:

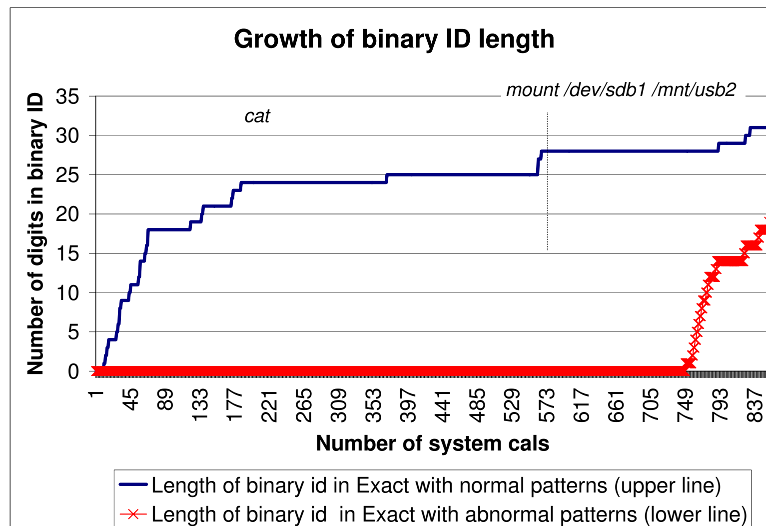


Figure 5.13 Growth of the number of states in Exact graphs with normal and abnormal patterns.

```
#define MODE (S_IRUSR||S_IWUSR||S_IRGRP||S_IROTH)
unsigned int nread, src, dst;
char *mybuff[100];
src = open("/etc/shadow",O_RDONLY);
dst = creat("/home/natalia/myfile", MODE);
while((nread=read(src,mybuff, 100)) < 0){
  write(dst,mybuff,nread);
}
```

The code creates a copy of */etc/shadow* file in the user's directory with read and write permissions for the file owner and read permission for group and others. This code produces the following abnormal sequence of system calls:

```
mmap2, open, open, read, write, read, write,..., read, write, ioctl, ioctl
```

This modified version was tested against the profile gathered with the original version of

the program given in Table 5.1. Our IDR system detected this sequence as anomalous when the system call `write` was produced. The first calls `mmap2, open, open, read` although represent the beginning of the abnormal pattern also present as a pattern in the normal profile. Therefore, the detection engine delayed the alert until the monitored sequence could not be found in the normal profile and matched an abnormal pattern.

Among the responses suitable for this type of violation (Table 5.2), the IDR system selected the response action *delete tampered with files* with risk factor= 0.5 and probability to succeed = 0.6 as the most effective response.

Detection of known program behavior: illegal shell access. As another example of the new pattern detection in the modified programs we experimented with `whoami` utility. The following fragment has been added to the `whoami main()` body of the program:

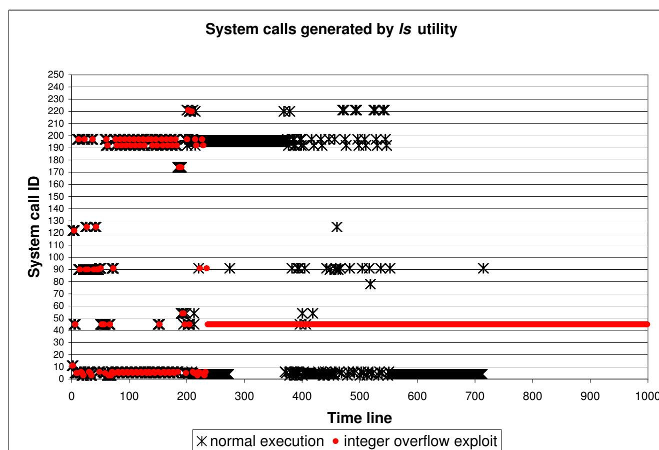
```
if (argc>1 &&(strcmp(argv[1],"--backdoor")==0)){
    char *line[2];
   args[0]= "sh";
   args[1]= NULL;
   setuid(0);setgid(0);
   execve("/bin/sh", line);
}
```

The code allows user to type `whoami --backdoor` in command prompt to gain access to the shell prompt without supplying a password. This shellcode; i.e., code that gives shell to an attacker, is a common code used for the majority of local buffer overflows. It grants the access to the shell `/bin/sh` by calling `execve()` (Younan, 2003). This shellcode produced one abnormal sequence of system calls:

```
write, setuid, setgid, execve
```

The first system call of this sequence *write* is not present in the normal profile of the original version of the program, this led the whole pattern is be classified as anomalous. Our IDR system was able to detect it with the first anomalous call *write*. The response selected

for deployment by IDR system among the applicable responses (Table 5.2) is *memory isolation* action which intuitively seems to be the best choice as it has low risk factor (0.2) with a good probability to succeed (0.6).



Sequence of system calls produced by integer overflow exploit includes over 39000 calls. However, starting at 237 exploit generates only system calls `brk` (ID number 45), thus for clarity of presentation the sequences of exploit calls is limited by 1000.

Figure 5.14 System calls generated by `ls` utility.

Detection limitations. In addition to the above experiments we explored integer overflow vulnerability present in `ls` utility. This problem arises when `ls` is provided with excessively large value for width `-w` option which will cause an incorrect handling of internal integer variable (CVE-2003-0853, 2003).

The exploitable condition can be reproduced by typing the following line in the command prompt:

```
/bin/ls -w 1073741828 -C
```

This command requests a listing of the current directory sorted by columns with column position width 1073741828. The value provided for width variable is used in further calculations of the actual columns' width. However, as the code does not provide necessary bound-checking of the accepted integer value, programs fails during the calculations.

The exploit produces the following behavior:

```
vm73>>/bin/ls -w 1073741828 -C
Out of Memory:Killed process 1603 (ls)
Out of Memory:Killed process 1603 (ls)
Out of Memory:Killed process 1603 (ls)
...
```

Although this vulnerability is non exploitable directly, due to the memory consumption it may cause remote denial of service through `wu-ftpd` (CVE-2003-0853, 2003).

Figure 5.14 presents a stream of systems calls generated by `ls` utility during its normal run and under integer overflow attack. System calls produced in each run are very similar. However, the exploit generates a long sequence of `brk` system call which causes a memory exhaustion.

The exploit was run against the profile of normal behavior generated with the original version of `ls`. However, the IDR system was not able to detect it due to two reasons. Firstly, our IDR system is not monitoring the data (e.g input parameters of the programs). Secondly, although the exploit's behavior is abnormal, it does not produce any changes on the system call level. The abnormally long sequence of `brk` system calls after it is broken down into variable-length `Exact` sequences, constitutes a set of short patterns that can be found in the normal profile.

In the considered scenario the underlying operating system terminated the process. However, if the process continued to execute and proceeded to the next step of attack which is the malicious code insertion and execution, IDR system would be able to detect and respond to such unusual behavior.

In this chapter we presented the experiments with the prototype of our IDR framework. We showed that our prototype is able to successfully detect different variations of abnormal known/unknown program behavior by monitoring the system call patterns. Upon detecting this anomalous behavior our system responds to it by selecting the most effective response.

CHAPTER 6. Conclusion

In this work we presented an integrated approach to intrusion detection and response based on the technique for monitoring abnormal patterns in the program behavior. The proposed model effectively combines the advantages of anomaly-based and specification-based approaches recognizing a known behavior through the specifications of normal and abnormal patterns and classifying unknown patterns using a machine-learning algorithm. Such combination not only allows adaptation of the specification-based detection to the new patterns, but also provides a method for automatic development of specifications.

In addition to detection, our framework also allows for preemptive response. By preemption, we imply deploying response before an monitored pattern is classified completely as an intrusion. Such response deployment is likely to stop an intrusion before it can affect the system. However, preemption also inherently suffers from false positives, i.e., responses are deployed to deter correct execution which may look intrusive in its initial phase. To reduce false positive, we have developed a multi-phase response selection and deployment mechanism. In the first phase, we identify whether or not to preempt depending on a pre-specified threshold. If the probability that the monitored pattern will result in a intrusion exceeds the threshold, our response selection procedure is invoked. The threshold can be managed depending on the tolerance level of system with respect to intrusions, as an example of this; the security-critical systems with low tolerance might set up a low threshold.

In the second phase, response selection proceeds by taking into consideration the damage that may be incurred by a possible intrusion against the damage that may be caused by the response action. This consideration is performed in several steps. In the first step, a set of responses corresponding to the anomalous pattern matched with the monitored sequence

is selected. As a number of intrusive patterns can have the same prefix, multiple candidate response actions are considered for deployment. To reduce the risk of damaging a monitored system by triggering a severe or incorrect response, we identify resources that can be potentially affected by the intrusion and corresponding response actions effective against this intrusion. In the next step, candidate responses evaluated based on the cost information of the system damage caused by intrusion and response. The final response strategy is determined through cost-satisfiability modeling of the candidate responses and the response action with the highest effectiveness is deployed.

We provide a prototype system implementation where we have deployed a host-based intrusion detection and response framework on Red Hat Linux 7.3 (Valhalla) Linux distribution with kernel 2.4.18-3. The implementation includes three components: *Hybrid interposition of system calls component (HISC)* responsible for system calls interception, *Detection module* performing a system call monitoring and processing through Exact with profiles of normal and abnormal behavior, and *Response module* determining the effective response. The experimentation with the system prototype allowed us to draw the following conclusions:

- System-call based detection of abnormal behavior can be done effectively in practice. Since the monitoring for anomalous patterns in programs behavior is based on the profile of normal behavior, the effectiveness of the detection can be characterized by the following factors:
  - *The profiles of normal behavior:* Our system is able to generate specifications of normal behavior automatically. We developed normal profiles for five Unix utilities in a short period of time. Although considered programs have very simple behavior, we believe that the profiles for more complex software applications can be generated automatically in a similar fashion provided additional time.
  - *Detection of unknown anomalous behavior:* Our system correctly recognized the behavior of mount utility as abnormal based on the profile of normal behavior of cat utility. Since the system did not have any information on the mount utility, its behavior was novel to the system.

- *Detection of known anomalous behavior:* We performed several experiments on the detection of programs anomalous behavior based on the normal and abnormal profiles. Our system successfully detected the attacks which behavior caused changes in the normal system-call flow of the programs.

- Cost-sensitive approach to preemptive intrusion response is an effective solution. Since the intrusion response field lacks well developed benchmarks for quantifying the efficiency of the response system, we evaluated our response framework based on the following contributions:

  - *Preemptive deployment of the responses:* Integration of pattern-based intrusion detection and response allows effective early attack identification and preemptive deployment of a response. Based on the profiles of normal and anomalous behavior our system was able to detect malicious behavior on its first stages and immediately select the most effective response action.

  - *Cost-sensitive response selection:* Using the provided cost information of the system resources and responses together with their correspondence matrix our response component selected an optimal response action.

6.0.2 Future work

We see the following directions of future research:

- *Development of resource list:* We have presented a classification of system resources that can be affected by a malicious process. One of the future avenues of research will be to develop a comprehensive list of the resources for various types of systems.

- *Development of domain-specific response mechanisms:* We have provided an overview of the response mechanisms mainly for a host-based IDR system. There is a need to develop a comprehensive list of response mechanisms for various types of intrusive behavior, e.g., OS attacks, network attacks and for various types of systems.

- *Completion and Optimization of IDR system:* The current prototype of the system is still in its infancy state. The detection process performs the patterns matching twice with the normal profile and similarly with the abnormal profile causing an additional overhead. The repeated matching can be avoided if Exact containing normal and abnormal sequences are integrated together.

APPENDIX A. Resource/Response Correspondence Matrices

	MEMORY RESOURCE	RESPONSE LATTICE
A: DATA SEGMENT	i. Illegal write to a process data segment	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ Memory isolation: ↙ ↘ Create a separate memory for the process to make updates / Disallow write to certain portions of memory for certain processes ↓ protection: Delay suspicious process ↓ passive responses: notification, logging
	ii. Wrong format/values of the used variables	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ protection: Delay suspicious process ↓ passive responses: notification, logging

Table A.1 Mapping between data segment resource and response actions

MEMORY RESOURCE		RESPONSE LATTICE
B: STACK SEGMENT	i. Illegal write to the stack memory	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ Memory isolation: ↙ ↘ Create a separate memory for the process to make updates / Disallow write to certain portions of memory for certain processes ↓ protection: Delay suspicious process ↓ passive responses: notification, logging
	ii. Illegal read of the stack memory	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ Memory isolation: ↙ ↘ Disallow read of certain portions of memory / Disallow read of certain portions of memory during certain times of its execution ↓ protection: Delay suspicious process ↓ passive responses: notification, logging

Table A.2 Mapping between stack resource and response actions

	INPUT/OUTPUT RESOURCE	RESPONSE LATTICE
C: FILE SYSTEM	i. Illegal creation/deletion of file/file links	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ File system isolation: ↓ Disallow full/selective access to file ↓ protection: ↙ ↘ Delete/restore created/deleted file/link Delay suspicious process ↓ passive responses: notification, logging
	ii.Illegal read/write of file content	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ File system isolation: ↙ ↓ ↘ File writes are made in a chroot environment Disallow full/selective access to file Allow read/write on sanitized/empty/dummy version of (privileged) file ↓ protection: ↙ ↘ Restore original/backed version of tampered with file Delay suspicious process ↓ passive responses: notification, logging
	iii.Incorrect file permissions	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ protection: Delay suspicious process ↓ passive responses: notification, logging

Table A.3 Mapping between file system resource and response actions

INPUT/OUTPUT RESOURCE		RESPONSE LATTICE
D: NETWORK INTERFACE	i. Illegal access of the data	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ Network isolation ↓ 1. Isolate process by capturing at the packet filter level the data sent to certain IP addresses/subnets 2. Prevent the process from sending data a) to certain IP addresses/subnets b) to any server other than the one from which a connection was made to the process 3. Prevent process access to all/some of the packet traffic ↓ protection: Delay suspicious process ↓ passive responses: notification, logging
	ii.Wrong format/values of the data	Process termination ↓ complete isolation : migration or emulation ↓ alternative methods to complete isolation ↓ protection: Delay suspicious process ↓ passive responses: notification, logging

Table A.4 Mapping between network interface resource and response actions

APPENDIX B. Response Factor Values Tables

Response action	Availability	Confidentiality	Integrity	Man-hours	Performance	Storage
Backup tampered with files		0.25			0.14	1
Delay suspicious process					0.86	
Delete tampered with file			1			
Detailed logging				0.33	0.29	0.5
File system isolation	0.8					
Generate alarm				1		
Generate report				0.67		
Global offset table isolation	0.1					
Intrusion analysis tools					0.71	
Memory isolation	0.2					
complete network isolation	1					
Network isolation: prevent process access to packet traffic	0.7					
Network isolation: block specific subnets	0.5					
Network isolation: packet inspection	0.6					
Process isolation: different environment	0.4	0.5				
Process isolation: virtual environment		1				
Process restart	0.3					
Process termination	0.9					
Remote code execution: full code		1			0.57	
Remote code execution: partial code		0.75			0.43	
Restore original backup version of tampered with file			0.5			

Table B.1 Impact Values by category for Potential Responses

Policy Goal	Weight
Availability	1
Integrity	0.5
Confidentiality	0.1
Performance	0.7
Man-hours	0.1
Storage	0.1

Table B.2 Assignment of Security Policy Priorities: Public Web Server

Policy Goal	Weight
Availability	0.2
Integrity	0.7
Confidentiality	1
Performance	0.2
Man-hours	0.01
Storage	0.1

Table B.3 Assignment of Security Policy Priorities: Classified

Policy Goal	Weight
Availability	0.7
Integrity	0.9
Confidentiality	0.5
Performance	0.3
Man-hours	0.01
Storage	0.1

Table B.4 Assignment of Security Policy Priorities: Business Critical

Response	RF Value
Generate-report	0.067
Global offset table isolation	0.100
Generate alarm	0.100
Memory isolation	0.200
Backup tampered with files	0.225
Restore original backup version of tampered with file	0.250
Detailed logging	0.283
Process restart	0.300
Remote code execution: partial code	0.375
Process isolation: different environment	0.450
Remote code execution: full code	0.500
Intrusion analysis tools	0.500
Delete tampered with file	0.500
Network isolation: block specific subnets	0.500
Delay suspicious process	0.600
Network isolation: packet inspection	0.600
Process isolation: virtual environment	0.700
Network isolation: prevent process access to packet traffic	0.700
File system isolation	0.800
Process termination	0.900
Network isolation: complete network isolation	1.00

Table B.5 Computed RF values: Public Web Server

Response	RF Value
Generate report	0.006
Generate alarm	0.009
Global offset table isolation	0.018
Memory isolation	0.036
Process restart	0.054
Network isolation: block specific subnets	0.090
Detailed logging	0.099
Network isolation: packet inspection	0.108
Network isolation: prevent process access to packet traffic	0.126
Intrusion analysis tools	0.128
File system isolation	0.144
Delay suspicious process	0.154
Process termination	0.162
Process isolation: virtual environment	0.179
Network isolation: complete network isolation	0.179
Restore original backup version of tampered with file	0.314
Backup tampered with files	0.340
Process isolation: different environment	0.521
Delete tampered with file	0.628
Remote code execution: partial code	0.750
Remote code execution: full code	1.00

Table B.6 Computed RF values: Classified

Response	RF Value
Generate report	0.007
Generate alarm	0.011
Global offset table isolation	0.078
Detailed logging	0.154
Memory isolation	0.156
Process restart	0.233
Intrusion analysis tools	0.238
Delay suspicious process	0.286
Backup tampered with files	0.298
Process isolation: virtual environment	0.333
Network isolation: block specific subnets	0.389
Network isolation: packet inspection	0.467
Restore original backup version of tampered with file	0.500
Network isolation: prevent process access to packet traffic	0.544
Remote code execution: partial code	0.560
Process isolation different environment	0.589
File system isolation	0.622
Process termination	0.700
Remote code execution: full code	0.746
Network isolation: complete network isolation	0.778
Delete tampered with file	1.00

Table B.7 Computed RF values: Business Critical

BIBLIOGRAPHY

AlephOne (1996). Smashing the stack for fun and profit. *Phrack Magazine*, 7(49).

AsmL (2007). Microsoft Research. Abstract State Machine Language. "http://research.microsoft.com/fse/asml/". Accessed May 15, 2007.

Axelsson, S. (2000). Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers Univ.

Balepin, I., Maltsev, S., Rowe, J., and Levitt, K. (2003). Using specification-based intrusion detection for automated response. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*.

Bazaz, A. and Arthur, J. D. (2005). On vulnerabilities, constraints and assumptions. "http://www.citebase.org/abstract?id=oai:arXiv.org:cs/0509076". Accessed May 15, 2007.

Bishop, M. (2003). *Computer Security: Art and Science*. Addison-Wesley Publishing Co.

Bishop, M. and Dilger, M. (1996). Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152.

Bowen, T., Chee, D., Segal, M., Sekar, R., Shanbhag, T., and Uppuluri, P. (2000). Building survivable systems: An integrated approach based on intrusion detection and damage containment. In *Proceedings, IEEE DARPA Information Survivability Conference and Exposition*.

C0ntex (2005). How to hijack the Global Offset Table with pointers for root shells. Available from "http://packetstormsecurity.org/papers/bypass/GOT_Hijack.txt". Accessed May 15, 2007.

Carver, C., Hill, J. M., and Surdu, J. R. (2000). A methodology for using intelligent agents to provide automated intrusion response. In *Proceedings of the IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop*, pages 110–116.

Carver, C. and Pooch, U. (2000). An intrusion response taxonomy and its role in automatic intrusion response. In *Proceedings of the IEEE Workshop on Information Assurance and Security*.

cat (2007). Linux man page. Available from "http://linux.die.net/man/1/cat". Accessed May 15, 2007.

Chang, C.-C. and Lin, C.-J. (2001). *LIBSVM: a library for support vector machines*. Software available at http://www.csie.ntu.edu.tw/ cjlin/libsvm.

Coyle, R. G. (1972). *Decision Analysis*. The Camelot Press Ltd., London, GB.

CVE-2003-0853 (2003). An integer overflow in ls in the fileutils or coreutils packages. Available from "http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2003-0853". Accessed May 15, 2007.

Dalvi, N., Domingos, P., Mausam, Sanghai, S., and Verma, D. (2004). Adversarial classification. In *Proceedings of the ACM SIGKDD International Conference on Knowledge discovery and data mining*.

d'Auriol, B. J. and Surapaneni, K. (2004). A state transition model case study for intrusion detection systems. In *Proceedings of the International Conference on Security and Management*, pages 186–192.

Debar, H., Becker, M., and Siboni, D. (1992). A neural network component for an intrusion detection system. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 240.

Debar, H., Dacier, M., Nassehi, M., and Wespi, A. (1998). Fixed vs. variable-length patterns for detecting suspicious process behavior. In *Proceedings of the European Symposium on Research in Computer Security*.

Dittrich, D. (2002). "Root Kits" and hiding files/directories/processes after a break-in. Available from "http://staff.washington.edu/dittrich/misc/faqs/rootkits.faq". Accessed May 15, 2007.

Enterasys (2007). Dynamic intrusion response. Available from "http://www.enterasys.com/". Accessed May 15, 2007.

Eskin, E., Lee, W., and Stolfo, S. J. (2001). Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings of DISCEX II*.

Eskin, E., Miller, M., Zhong, Z.-D., Yi, G., Lee, W.-A., and Stolfo, S. J. (2000). Adaptive model generation for intrusion detection. In *Proceedings of the ACMCCS Workshop on Intrusion Detection and Prevention*.

Fayolle, P.-A. and Glaume, V. (2002). A buffer overflow study, attacks & defenses. Available from "http://www.enseirb.fr/ glaume/indexen.html". Accessed May 15, 2007.

Fisch, E. (1996). *A Taxonomy and implementation of automated responses to intrusive behavior*. PhD dissertation, Texas A&M University.

Foo, B., Wu, Y.-S., Mao, Y.-C., Bagchi, S., and Spafford, E. H. (2005). ADEPTS: Adaptive intrusion response using attack graphs in an e-commerce environment. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 508–517.

Forrest, S. (2005). Computer immune systems, data sets. Available from "http://www.cs.unm.edu/$\tilde{\imath}$mmsec/data/synth-sm.html". Accessed November 15, 2005.

Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. (1996). A sense of self for Unix processes. In *Proceedinges of the IEEE Symposium on Security and Privacy*.

Ganapathy, V., Jha, S., Chandler, D., Melski, D., and Vitek, D. (2003). Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 345–354, New York, NY, USA. ACM Press.

Garvey, T. and Lunt, T. (1991). Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference*.

Gopalakrishna, R., Spafford, E. H., and Vitek, J. (2005). Efficient intrusion detection using automaton inlining. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 18–31, Washington, DC, USA.

Habra, N., Charlier, B. L., Mounji, A., and Mathieu, I. (1992). ASAX : Software architecture and rule- based language for universal audit trail analysis. In *European Symposium on Research in Computer Security)*, pages 435–450.

Hinton, H. M., Cowan, C., Delcambre, L. M. L., and Bowers, S. (1999). SAM: Security adaptation manager. In *Proceedings of Annual Computer Security Applications Conference*, pages 361–370.

Hofmeyr, S. A., Forrest, S., and Somayaji, A. (1998). Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180.

Huang, Y., Fan, W., Lee, W., and Yu, P. S. (2003). Cross-feature analysis for detecting ad-hoc routing anomalies. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 478.

Ilgun, K. (1993). Ustat: A real-time intrusion detection system for unix. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 16.

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

Innella, P. (2001). The evolution of intrusion detection systems. Available from "http://www.securityfocus.com/infocus/1514". Accessed May 15, 2007.

Jean-Philippe Pouzol, M. D. (2002). Formal specification of intrusion signatures and detection rules. In *Proceedings of the IEEE Computer Security Foundations Workshop*.

Joglekar, S. P. and Tate, S. R. (2005). Protomon: Embedded monitors for cryptographic protocol intrusion detection and prevention. *J. UCS*, 11(1):83–103.

Jones, R. W. M. and Kelly, P. H. J. (1997). Backwards-compatible bounds checking for arrays and pointers in c programs. In *Automated and Algorithmic Debugging*, pages 13–26.

Kemmerer, R. A. and Vigna, G. (2002). Intrusion detection: A brief history and overview. 35:27 – 30.

Ko, C. (2000). Logic induction of valid behavior specifications for intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Ko, C., Brutch, P., Rowe, J., Tsafnat, G., and Levitt, K. N. (2001). System health and intrusion monitoring using a hierarchy of constraints. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 190–204.

Ko, C., Fink, G., and Levitt, K. (1994). Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the Annual Computer Security Applications Conference*, pages 134–144.

Ko, C., Ruschitzka, M., and Levitt, K. N. (1997). Execution monitoring of security-critical programs in distributed systems: A specification-based approach. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 175–187.

Kosoresow, A. P. and Hofmeyr, S. A. (1997). Intrusion detection via system call traces. 14:24–42.

Krishnakumar, R. (2005). Experiments with the linux kernel: Process segments. *Linux Gazette*, (112).

Kumar, S. (1995). *Classification and Detection of Computer Intrusions.* PhD dissertation, Purdue University, West Lafayette, IN, USA.

Kumar, S. and Spafford, E. H. (1994). A pattern matching model for misuse intrusion detection. In *Proceedings of the National Computer Security Conference*, pages 11–21, Baltimore, MD, USA.

Lane, T. and Brodley, C. (1999). Temporal sequence learning and data reduction for anamoly detection. *ACM Transactions on Information and System Security*, 2(3):295–331.

Lazarevich, A., Ertoz, L., Ozgur, A., Srivastava, J., and Kumar, V. (2003). A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of SIAM Conference on Data Mining.*

Lee, W., Fan, W., Millerand, M., Stolfo, S., and Zadok, E. (2000). Toward cost-sensitive modeling for intrusion detection and response. In *Journal of Computer Security*, volume 10.

Lewandowski, S. M., Hook, D. J. V., O'Leary, G. C., Haines, J. W., and M.Rossey, L. (2001). SARA: Survivable autonomic response architecture. In *DARPA Information Survivability Conference and Exposition II.*

Lhee, K. and Chapin, S. (2002). Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90.

Lin, J.-L., Wang, X. S., and Jajodia, S. (1998). Abstraction-based misuse detection: High-level specifications and adaptable strategies. In *Proceedings of the IEEE Computer Security Foundations Workshop*, pages 190–201.

Linn, C. M., Rajagopalan, M., Baker, S., Collberg, C., Debray, S. K., and Hartman, J. H. (2005). Protecting against unexpected system calls. In *Proceedings of the 14th conference on USENIX Security Symposium*, pages 16–16. USENIX Association.

Locasto, M. E., Wang, K., Keromytis, A. D., and Stolfo, S. J. (2005). FLIPS: Hybrid adaptive intrustion prevention. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 82–101.

ls (2007). Linux man page. Available from "http://linux.die.net/man/1/ls". Accessed May 15, 2007.

Lunt, T. F., Tamaru, A., Gilham, F., Jagannathan, R., Jalali, C., Neumann, P. G., Javitz, H. S., Valdes, A., and Garvey, T. (1992). A real-time intrusion-detection expert system (ides). Technical report, SRI International.

Marceau, C. (2000). Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the workshop on New security paradigms*, pages 101–110.

MIT (1998). Intrusion detection attacks database. Available from "http://www.ll.mit.edu/IST/ideval/docs/1998/attackDB.html". Accessed May 15, 2007.

mount (2007). Linux man page. Available from "http://linux.die.net/man/8/mount". Accessed May 15, 2007.

Musman, S. and Flesher, P. (2000). System or security managers adaptive response tool. In *DARPA Information Survivability Conference and Exposition II*.

Ning, P., Xu, D., Healey, C. G., and Amant, R. A. S. (2004). Building attack scenarios through integration of complementary alert correlation methods. In *Proceedings of the Annual Network and Distributed System Security Symposium*, pages 97–111.

Noel, S., Jacobs, M., Kalapa, P., and Jajodia, S. (2005). Multiple coordinated views for network attack graphs. In *Proceedings of the Workshop on Visualization for Computer Security*.

Noel, S. and Jajodia, S. (2004). Managing attack graph complexity through visual hierarchical aggregation. In *Proceedings of the ACM workshop on Visualization and data mining for computer security*, pages 109–118.

Noel, S., Robertson, E., and Jajodia, S. (2004). Correlating intrusion events and building attack scenarios through attack graph distances. In *Proceedings of the Annual Computer Security Applications Conference*.

Ogorkiewicz, M. and Frej, P. (2002). Analysis of buffer overflow attacks. Available from "http://www.windowsecurity.com/articles/windows_os_security". Accessed May 15, 2007.

Peltier, T. R. (2001). *Information Security Risk Analysis*. Auerbach Publications.

Porras, P. and Neumann, P. (1997). EMERALD: event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the National Information Systems Security Conference*.

Ragsdale, D., Carver, C., Humphries, J., and Pooch, U. (2000). Adaptation techniques for intrusion detection and intrusion response system. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics at Nashville, Tennessee*, pages 2344–2349.

Raihan, M. F. and Zulkernine, M. (2005). Detecting intrusions specified in a software specification language. In *Proceedings of the Annual International Computer Software and Applications Conference*, pages 143–148, Washington, DC, USA. IEEE Computer Society.

Schiffman, M., Eschelbeck, G., Ahmad, D., Wright, A., and Romanosky, S. (2004). CVSS: A Common Vulnerability Scoring System. In *National Infrastructure Advisory Council (NIAC)*.

Schnackenberg, D., Djahandari, K., and Sterne, D. (2000). Infrastructure for intrusion detection and response.

Schnackenberg, D., Holliday, H., Smith, R., et al. (2001). Cooperative intrusion traceback and response architecture CITRA. In *Proceedings of the IEEE DARPA Information Survivability Conference and Exposition (DISCEX I)*.

Scholkopf, B., Platt, J., Shawe-Taylor, J., Smola, A., and Williamson, R. (1999). Estimating the support of a high-dimensional distribution. Technical Report 99-87, Microsoft Research.

Sebring, M., Shellhouse, E., Hanna, M., and Whitehurst, R. (1988). Expert systems in intrusion detection: A case study. In *Proceedings of the National Computer Security Conference*.

Secunia (2004). haserl manipulation of critical environment variables vulnerability. Available from "http://secunia.com/advisories/13031". Accessed May 15, 2007.

Sekar, R., Cai, Y., and Segal, M. (1998). A specification-based approach for building survivable systems. In *Proceedings of the NIST-NCSC National Information Systems Security Conference*, pages 338–347.

Sekar, R., Gupta, A., Frullo, J., Shanbhag, T., Tiwari, A., Yang, H., and Zhou, S. (2002). Specification-based anomaly detection: a new approach for detecting network intrusions. In *Proceedings of the ACM conference on Computer and communications security*.

Sekar, R. and Uppuluri, P. (1999). Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the Usenix Security Symposium*.

Sheyner, O., Haines, J., Jha, S., Lippmann, R., and Wing, J. M. (2002). Automated generation and analysis of attack graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Sidiroglou, S., Locasto, M. E., Boyd, S. W., and Keromytis, A. D. (2005). Building a reactive immune system for software services. In *Proceedings of the USENIX Annual Technical Conference*.

Simon, I. (2002). A comparative analysis of methods of defense against buffer overflow attacks. Available from "http://www.mcs.csuhayward.edu/ simon/security/boflo.html". Accessed May 15, 2007.

Smaha, S. E. (1988). Haystack: An intrusion detection system. In *Proceedings of the IEEE Aerospace Computer Security Applications Conference*.

Somayaji, A. and Forrest, S. (2000). Automated response using system-call delay. In *Proceedings of the USENIX Security Symposium*.

Song, T., Alves-Foss, J., Ko, C., Zhang, C., and Levitt, K. (2003). Using ACL2 to Verify Security Properties of Specification-based Intrusion Detection Systems. In *Proceedings of the International Workshop on the ACL2 Theorem Prover and Its Applications*.

Stakhanova, N., Basu, S., Lutz, R., and Wong, J. (2006). Automated caching of behavioral patterns for efficient run-time monitoring. In *Proceedings of the IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 333–340.

Stakhanova, N., Basu, S., and Wong, J. (2007a). A cost-sensitive model for preemptive intrusion response systems. In *Proceedings of the IEEE International Conference on Advanced Information Networking and Applications*, pages 428–435.

Stakhanova, N., Basu, S., and Wong, J. (2007b). A taxonomy of intrusion response systems. In *International Journal of Information and Computer Security*, volume 1, pages 169–184.

Staniford-Chen, S., Cheung, S., Crawford, R., Dilger, M., Frank, J., Hoagland, J., Levitt, K., Wee, C., Yip, R., and Zerkle, D. (1996). GrIDS-a graph based intrusion detection system for large networks. In *Proceedings of the National Information Systems Security Conference*, pages 361–370.

Swiler, L. P., Phillips, C., Ellis, D., and Chakerian, S. (2001). Computer-attack graph generation tool. In *DISCEXII Proceedings, DARPA's Information Survivability Conference and Exposition*.

Tan, K. and Maxion, R. (2002). Why 6? Defining the Operational Limits of Stide, an Anomaly-Based Intrusion Detector. In *Proceedings of the IEEE Symposium on Security and Privacy*.

TippingPoint (2007). Intrusion prevention systems. Available from "http://www.tippingpoint.com". Accessed May 15, 2007.

Toth, T. (2003). *Improving Intrusion Detection Systems*. PhD dissertation, Technical University of Vienna.

Toth, T. and Kruegel, C. (2002). Evaluating the impact of automated intrusion response mechanisms. In *Proceedings of the Annual Computer Security Applications Conference*.

Tseng, C.-Y., Balasubramanyam, P., Ko, C., Limprasittiporn, R., Rowe, J., and Levitt, K. (2003). A specification-based intrusion detection system for AODV. In *Proceedings of the 1st ACM workshop on Security of ad hoc and sensor networks*, pages 125–134.

Tseng, C.-Y., Song, T., Balasubramanyam, P., Ko, C., and Levitt, K. N. (2005). A specification-based intrusion detection model for OLSR. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 330–350.

Tsyrklevich, E. and Yee, B. (2003). Dynamic detection and prevention of race conditions in file accesses. In *Proceedings of the 12th conference on USENIX Security Symposium*, pages 17–17, Berkeley, CA, USA. USENIX Association.

Uppuluri, P. (2003). *Intrusion Detection/Prevention Using Behavior Specifications*. PhD dissertation, State University of New York at Stony Brook.

Uppuluri, P., Joshi, U., and Ray, A. (2005). Preventing race condition attacks on file-systems. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 346–353, New York, NY, USA. ACM Press.

Uppuluri, P. and Sekar, R. (2000). Experiences with specification-based intrusion detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*.

Uppuluri, P., Xu, W., Sun, W., Bhatkar, S., and Sekar, R. (2006). Hybrid system call interposition. Technical report, University of New York at Stony Brook.

Vapnik, V. (1998). *Statistical Learning Theory*. Wiley, New York.

Wagner, D. and Dean, D. (2001). Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A. (2000). A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA.

Wagner, D. and Soto, P. (2002). Mimicry attacks on host based intrusion detection systems. In *The Proceedings of the ACM Conference on Computer and Communications Security*.

Wang, X., Reeves, D. S., and Wu, S. F. (2001a). Tracing based active intrusion response. In *Journal of Information Warefare*, volume 1.

Wang, X., Reeves, D. S., Wu, S. F., and Yuill, J. (2001b). Sleepy watermark tracing: an active network-based intrusion response framework. In *Proceedings of the 16th international conference on Information security: Trusted information*, pages 369–384.

Wang, Y., Wong, J., and Miner, A. (2004). Anomaly intrusion detection using one class svm. In *Proceedings of the IEEE Information Assurance Workshop*.

Warrender, C., Forrest, S., and Pearlmutter, B. (1999). Detecting intrusions using system calls: Alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*.

Wheeler, D. A. (2003). *Secure Programming for Linux and Unix HOWTO*. "http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO.html". Accessed May 15, 2007.

White, G., Fisch, E., and Pooch, U. (1996). Cooperating security managers: A peer-based intrusion detection system. In *IEEE Network*, volume 10, pages 20–23.

whoami (2007). Linux man page. Available from "http://linux.die.net/man/1/whoami". Accessed May 15, 2007.

Wikipedia (2007). Rootkit — wikipedia, the free encyclopedia. Available from "http://en.wikipedia.org/w/index.php?title=Rootkit&oldid=153619249". Accessed May 15, 2007.

Wu, Y.-S., Foo, B., Mao, Y.-C., Bagchi, S., and Spafford, E. H. (2007). Automated adaptive intrusion containment in systems of interacting services. *Comput. Networks*, 51(5):1334–1360.

Yong, S. H. (2004). *Runtime Monitoring of C Programs for Security and Correctness*. PhD dissertation, University of Wisconsin-Madison, Madison, USA.

Younan, Y. (2003). An overview of common programming security vulnerabilities and possible solutions. Master's thesis, Vrije Universiteit, Brussel, Belgium.

Younan, Y., Joosen, W., and Piessens, F. (2005). A Methodology for Designing Countermeasures against Current and Future Code Injection Attacks. In *Proceedings of the Third IEEE International Workshop on Information Assurance (IWIA'05)*, pages 3–20, Washington, DC, USA. IEEE Computer Society.

Younan, Y., Piessens, F., and Joosen, W. (2006). Protecting global and static variables from buffer overflow attacks without overhead. Technical Report CW463, Departement Computerwetenschappen, Katholieke Universiteit Leuven.