CUPIDS: INCREASING INFORMATION SYSTEM SECURITY THROUGH THE USE OF DEDICATED CO-PROCESSING

A Thesis

Submitted to the Faculty

of

Purdue University

by

Paul D. Williams

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2005

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
00 AUG 2005	**N/A**	

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CUPIDS: Increasing Information System Security through the Use of Dedicated Co-processing	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Purdue University**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited.

13. SUPPLEMENTARY NOTES
The original document contains color images.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**UU**	**125**	

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std Z39-18

The opinions expressed in this dissertation are my own, and do not necessarily reflect the views of the US Air Force, or the US Government.

ACKNOWLEDGMENTS

This work would not have been possible without my family and friends who have supported me in myriad ways throughout the years, the excellent teachers who provided a solid foundation upon which to build and then boosted me up on top of it, the support personnel who keep the schools running and my fellow students who always kept things interesting. To each of you[1]: I am immensely grateful for your support!

I would not be here without the support of the United States Air Force. The USAF provides a terrific place to work and endeavor to make the world a better place for everyone's children (and also pays the bills). There has not been a day over the last three years when I have not considered not only how lucky I am to be here working on this degree, but also to be part of such a wonderful organization. I am very much looking forward to getting back into the "Real Air Force" and paying back the investment the USAF has made in my education.

Finally, I would also like to acknowledge my children. Kids—my getting this thing done came with a hefty cost to you in terms of time and attention. Thank you so much for your patience with my weird hours, occasional tired crankyness, and all the rest of the hassle which goes with blending daddyhood and graduate school. I love you both $> (\infty * 2006)$ and am very much looking forward to a more "normal" life with you!

[1]USAF regulations prevent me from using names here.

TABLE OF CONTENTS

Page

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

API	Application Programming Interface
CPP	CuPIDS Production Process
CPU	Central Processing Unit
CSP	CuPIDS Shadow Process
CuPIDS	Co-Processor-Based Intrusion Detection System
HTT	Hyperthreading Processor
ID	Intrusion Detection
IDS	Intrusion Detection System
IPC	Interprocess Communications
MAC	Mandatory Access Control
MC	Multicore Processor
MP	Multi-Processor
NRT	Near Real-Time
O/S	Operating System
SPCM	Security Policy Compliance Monitoring
RT	Real-Time
RVM	Reference Validation Monitor
SMP	Symmetrical Multi-Processing
StUPIDS	Standard Uni-Processor-Based Intrusion Detection System
UP	Uni-Processor

ABSTRACT

Williams, Paul D. Ph.D., Purdue University, August, 2005. CuPIDS: Increasing Information System Security through the Use of Dedicated Co-processing. Major Professor: Eugene H. Spafford.

Most past and present intrusion detection systems architectures assume a uni-processor environment or do not explicitly make use of multiple processors when they exist. Yet, especially in the server world, multiple processor machines are commonplace; and with the advent of technologies such as Intel and AMD's multi-core or Hyperthreading technologies, commodity computers are likely to have multiple processors.

This research explores how explicitly dividing the system into production and security components and running the components in parallel on different processors can improve the effectiveness of the security system. The production component contains all user tasks and most of the operating system while the security component contains security monitoring and validating tasks and the parts of the O/S that pertain to security. We demonstrate that under some circumstances this architecture allows intrusion detection systems to use monitoring models with higher fidelity, particularly with regard to the timeliness of detection, and will also increase system robustness in the face of some types of attacks.

Empirical results with a prototype co-processing intrusion detection system (CuPIDS) architecture support the feasibility of this approach. The construction of the prototype allowed us to demonstrate the implementation costs of the architecture are reasonable. Experimentation using fine-grained protection of real-world applications resulted in about a fifteen percent slowdown while demonstrating CuPIDS' ability to quickly detect and respond to illegitimate behavior.

1 INTRODUCTION

It is feasible to increase information system security by performing system security tasks in parallel with the operational tasks they protect. In this document we examine this claim in detail and describe work performed to demonstrate its validity.

1.1 Background and Problem Statement

Computer-based information processing systems have become a vital element of a broad spectrum of our society's foundation. Our artists use them to define and share their philosophies [1], our farmers depend on them to maximize crop yields [2], our military uses them to fight wars [3], and our scientists use them to improve our future [4]. These information systems, however, have been and are vulnerable to attack [5], both from within and without their host organizations [6]. Our reliance upon these vulnerable information systems necessitates that we attempt to protect them from attack or erroneous operation. Recognizing that we cannot construct provably secure systems [7], we focus instead upon detecting and stopping illegitimate software behavior.

While most of the experimentation described in this dissertation is intrusion detection (ID)-centric, the underlying ideas are much more general in nature. The parallel monitoring architecture described in this dissertation supports security policy compliance monitoring—an overarching concept that encompasses intrusion detection as well as a number of related areas such as error detection and computer forensics.

1.2 Overview of Intrusion Detection

Research into ID is not new [8], nor, based upon the breadth and depth of the field [9, 10], is it simple or solved.

Intrusion detection is defined by Mukherjee, Heberlein, and Levitt as "the problem of identifying individuals who are using a computer system without authorization (i.e., crackers) and those who have legitimate access to the system but are abusing their privileges (i.e., the insider threat)" [11]. This definition needs to be modified to add the identification of attempts to use an information system either without authorization, or in an unauthorized manner [12]. Note that the insider threat is also classified as misuse, and that misuse detection systems have also been proposed and built, but most modern intrusion detection systems (IDS) are designed to perform both activities.

Traditionally there were two main and orthogonal classes of IDS, host and network-based. Recent work has begun to separate and define a third, specification-based detection. Specification-based ID systems search for actions or events that occur outside the specifications of that system (see Bishop Chapter 25 [13] for details).

Host-based IDS, which is typically viewed as deriving from Dorothy Denning's seminal ID model [14], resides on a single machine and monitors the activities of users and processes inside that machine. The range of activities of a host-based IDS start with log monitoring, in which the tool scans through the activity logs created by the host operating system, and may extend all the way through actually monitoring all system calls and activity. A host-based IDS typically has access only to information about activity concerning its host. By itself, it does not have much knowledge of the external world.

A network based IDS is located outside of a host and watches the raw network traffic. It is able to monitor all the traffic between computers in the external world (usually the world is limited to the network segment to which the network IDS is attached), but is not able to determine exactly what is happening inside the

individual machines under its protection. The purpose of both types of system is to detect suspicious behavior. In practice, more and more IDSs are employing both host and network based components [15]. This allows both an internal and external view of all the activity on a network.

A second axis of definition for ID is based upon whether or not existing signatures are used to detect problematic events [9]. Many IDSs are signature-based with operation analogous to that of virus detection systems. They scan network traffic for patterns of activity that have been defined as likely to be part of a scan, probe, or attack.When a pattern is recognized, an alarm is sent to a higher level analyst, which is often a human. The analyst then tries to verify the attack and respond accordingly. As in virus detection systems, this strategy has some strategic weaknesses. The first is that only previously detected attacks have signatures. A novel attack being run manually by a sophisticated attacker is not likely to match a pre-existing signature and therefore may escape detection by a signature-based IDS. Secondly, many attack patterns are similar to the patterns of normal, accepted usage. This leads many security system designers into the quandary of having to choose between a system that alarms often on normal activity (known as the false-positive problem), or one that ignores normal activity, and may not detect an attack that resembles normal activity (known as the false-negative problem). [16, 17]

Mukherjee, Heberlein, and Levitt define two basic intrusion detection models, a Misuse detection model, and an Anomaly detection model [11]. In the misuse detection model, detection is performed primarily by searching for signatures of attacks against known vulnerabilities. In the anomaly detection model, intrusions are detected by the changes they make in the system utilization and behavior patterns. An IDS is a system, with hardware, software, and perhaps human components, that uses the misuse and/or anomaly detection models to identify intrusions.

1.2.1 Desirable Characteristics of Intrusion Detection Systems

Spafford et. al. [12, 18–20] describe ten desirable characteristics of an IDS:

1. The IDS must run continually with minimal human supervision.
2. The IDS must be fault tolerant. This means that the parts of the system should be able to recover their previous state upon startup after a crash.
3. The IDS must be able to resist subversion. It should be able to detect when it has been attacked.
4. It must not impose an unreasonable amount of overhead on the hosts upon which its pieces run.
5. It must be configurable to match an organizations security policies.
6. It must adapt to system changes. These changes may include new applications, users being added or moved, and new resources.
7. The IDS must be scalable. As the number of protected hosts increases, the IDS should still be able to function in a timely and accurate manner.
8. It must provide graceful degradation. If some IDS components fail, the rest should be able to continue to function.
9. It must be able to detect attacks:
   (a) The IDS should not flag legitimate activity as an attack (false positives).
   (b) The IDS should not fail to flag any real attacks as such (false negatives).
   (c) It should be difficult for an attacker to mask his actions to avoid detection.
   (d) The IDS must report intrusions as soon as possible after they occur.
   (e) The IDS must be general enough to detect different types of attacks.

10. Finally, it should provide for dynamic configuration. The pieces of the IDS should be changeable on the fly, without having to restart the entire system after a change.

Our research directly addresses all ten of these characteristics (See Section 6.1.1 for details).

1.2.2 Current Intrusion Detection Challenges

Two (2000) comprehensive reviews of the state of the art in intrusion detection identified these challenges to IDS systems [9, 15]:

1. Increases in the types of intruder goals, intruder abilities, tool sophistication, and diversity, as well as the use of more complex, subtle, and new attack scenarios
2. The use of encrypted messages to transport malicious information
3. The need to interoperate and correlate data across infrastructure environments with diverse technologies and policies
4. Ever-increasing network traffic
5. The lack of widely accepted IDS terminology and conceptual structures
6. Volatility in the IDS marketplace that makes the purchase and maintenance of IDS systems difficult
7. Risks inherent in taking inappropriate automated response actions
8. Attacks on the IDS systems themselves
9. Unacceptably high levels of false positives, making it difficult to determine true positives
10. The lack of objective IDS system evaluation and test information

11. The fact that most computing infrastructures are not designed to operate securely

12. Limited network traffic visibility resulting from switched local area networks

13. Faster networks preclude effective real-time analysis of all traffic on large pipes

This research addresses the first, second, seventh, eighth, ninth, eleventh, twelfth and thirteenth challenges (See Section 6.1.2 for details).

We note that most past and present Intrusion Detection Systems (IDS) architectures assume a simple, uni-processor environment, or do not explicitly make use of multiple processors when they exist. However, especially in the server world, multiple processor machines are commonplace, and with the advent of technologies such as Intel and AMD's multi-core or Hyperthreading technologies [21–23], commodity computers are likely to have multiple processors. This research explores how explicitly dividing the system into parts that are then allocated to particular processors can improve the effectiveness of the security system.

Our primary research thesis is that a tightly focused, parallel monitoring process can detect illegitimate behavior more quickly than can a process operating in either scheduler-interleaved[1] or interposing mode[2]. A cornerstone of the Co-processing Intrusion Detection System (CuPIDS) philosophy is that security is more important than raw performance—particularly with regards to mission critical applications and servers. We also recognize that for this research to be useful it must operate without high performance costs as perceived by system users and developers. For the former group we explore the use of hardware to accelerate security tasks that have primarily been handled by software. The implementation process allowed us to evaluate the likely costs a developer of a CuPIDS application will incur.

A central focus of our research was on the tension between usability and security. We evaluated the psychological acceptability [25] of the CuPIDS architecture;

[1]Monitoring code competes with the monitored process for time on a processor.

[2]Monitoring code is interposed or placed between instructions in the original program allowing it to validate program activity at the points where the monitor is inserted. [24]

exploring how it as a security mechanism makes the system more difficult to utilize or less efficient. Using commodity hardware and lightly customized commodity software we demonstrate that the performance impact of the CuPIDS architecture is modest and tolerable.

1.3 Security Policy Compliance Monitoring

The need for security policy compliance monitoring (SPCM) is not new. Indeed, an ability to detect when a computer system is operating outside its design specifications has existed since the first computers were placed into production use. We define SPCM in two parts: policy generation and policy compliance monitoring. Policy generation is the process of explicitly defining how an information system should operate, its inputs and outputs, resource usage, and execution behavior. Policy compliance monitoring couples the security policy with monitoring of system operation to detect when the system deviates from the stated policy. Historically SPCM has not been generally achievable. The reasons for this are manyfold, but in general it has been too difficult, both in terms of policy generation, and in the computational ability to monitor effectively without an overwhelming runtime cost. The bodies of work in intrusion detection, software engineering and debugging, trusted computing, computer language design, and computer forensics all represent efforts to provide satisficing [26] solutions to specific portions of the general problem. We understood at the outset of our research into parallel security that the concepts with which we would be working crossed over the boundaries of the research areas mentioned above; however it was not until the end of the project that we began to understand how they all fit together under the concept of SPCM. Though this research concentrates primarily on how monitoring in parallel offers advantages over the uni-processor alternatives, the detector sets we have used, as well as advances in all the related research areas, illustrate ways in which SPCM may become feasible.

1.4 Threat Model

A major premise of this research is that the applications in use today and in the reasonably foreseeable future will contain vulnerabilities. Through faults or active exploitation the existence of these vulnerabilities may lead to the application's compromise. If the application is privileged the compromise can effect the operation of the entire system. This research attempts to contain the effects of such a compromise and prevent the compromise of key system components such as the security system even in the face of a successful root-level attack.

We are concerned with a general threat model that assumes:

- Processes running at any privilege level in the production parts of the system may be compromised at any time after boot is complete.
- Attacks may come from local or external users or a combination of both.
- Attacks may succeed without ever causing a context switching event.

Previous work typically assumed a more constrained threat model.

1.5 Thesis Statement

This dissertation describes the work done to show the validity of the following hypothesis:

Under some circumstances, Co-processor-based Intrusion Detection/Intrusion Prevention Systems (CuPIDS) can be more effective than a Standard Uni-processor-based IDS (StUPIDS)[3].

For our purposes more effective is shown by demonstrating that:

1. Running concurrently with attack code affords CuPIDS opportunities that are not available to StUPIDS to detect and respond to attacks.

[3]The name StUPIDS is in tribute to the work done in Purdue's Coast Laboratory on the IDIOT intrusion detection system [27].

2. Because the opportunity exists to detect attacks while they occur without waiting for a context switching event, CuPIDS may be able to respond more quickly and attacks may be detected with higher fidelity.

These are advantages that are difficult or impossible to achieve on a uni-processor system—no matter how powerful.

1.6 Justification for Thesis Statement

Our conjecture is that the use of a dedicated security processor offers two major advantages over uni-processor or non-dedicated systems: concurrency and protection. We believe that these two advantages will allow us to increase the effectiveness of the IDS by allowing the use of higher fidelity security models (particularly with regard to the timeliness of detection). The second advantage derives from the first—by detecting and responding to attacks quickly, compromises of attacker-reachable applications may be halted or prevented, thus making bootstrapping attacks that compromise the entire system more difficult.

In a uni-processor system only one instruction stream is executed at a given point in time. This results in attack code running in isolation on the machine with opportunities to compromise the system and/or destroy traces of its actions before a StUPIDS has a chance to detect the malicious activity. This problem also exists in multi-processor systems that do not allocate at least one processor exclusively to IDS related tasks. By ensuring that the IDS-related processes are always running and have exclusive access to a processor, CuPIDS reduces this advantage for the attacker.

Many host-based IDS interact with monitored processes in a synchronous and serial manner. For example a system that uses a model of system call activity for detection may interpose itself between the application and the system calls. This allows the IDS to validate system call requests, but it does so while the application is blocked. To reduce slowdowns for the production process the IDS must limit the

amount of work it can do while the production application is blocked. In this example our model allows the IDS to run concurrently with the system call execution thereby increasing the amount of work that can be done by the IDS to validate the system call and its parameters without impacting the performance of the production process. CuPIDS can also interact asynchronously with production processes. The parallel nature of CuPIDS' operation allows CuPIDS to take snapshots of the production process' state or take control of it at any time. These capabilities do not depend on a context switching event taking place before CuPIDS can take action.

1.7 Contribution

Key differences between this research and existing work include:

1. Our security processor is dedicated exclusively to security monitoring and controlling tasks. Most previous co-processor-based security research has focused primarily on secure booting or intellectual property protection.

2. Code running on the security CPU is different than that running on production CPUs.

3. Under many circumstance CuPIDS can guarantee real-time detection and response to illegitimate events—this guarantee cannot be made by a StUPIDS with a comparable detector set (time definition and discussion can be found in Section 2.1.2).

4. Security processes in our system interact with production processes both asynchronously and synchronously. While other IDS can do one or the other, ours it the first to be able to do both at once.

5. CuPIDS can monitor activity during critical code sections [4] as well as operations with real-time [10] constraints which may not be interrupted. This capability cannot be achieved on a uni-processor system.

1.8 Document Organization

This dissertation is organized as follows: Chapter 1 presents the background, thesis statement, justification for the thesis statement, and brief discussion of the research contribution. Chapter 2 presents past work done in the areas of ID, the use of co-processors in related areas, and other fundamental research that is relevant to this work. Chapter 3 describes the abstract model of multi-processor security. Chapter 4 describes an implementation of a multi-processor architecture based on the model in Chapter 3. Chapter 5 describes the experimentation done using that prototype. Finally, Chapter 6 presents the conclusions, summarizes the contributions of this work and discusses directions for future research.

[4] Here we define a critical code section as a code segment in which a process cannot be interrupted for any reason, including context switches or system interrupts. This definition differs slightly from the mutual exclusion critical section as defined in [28], rather it is the concept used to define operations such as operating system kernel operations that may not be interrupted.

2 RELATED WORK

This chapter discusses research related to our thesis statement.

2.1 Intrusion Detection Surveys and Taxonomies

2.1.1 Intrusion Detection Systems: A Survey and Taxonomy (2000)

Axelsson provides an in-depth, thorough taxonomy and survey of the field of intrusion detection [9]. The work is based primarily upon major research efforts that had progressed to the point that prototype systems were available to study. He begins by pointing out some of the major difficulties in ID: the adversary is difficult to pin down; in most cases we do not yet know what to look for; and in many cases even when we do know what to look for, the intrusive activity is sufficiently similar to that of normal activity that separating the two is extremely difficult. He points out that much of the difficulty lies in the information available to IDSs: logging sources that are rich in accounting information, but poor in information that will help catch intruders. See the section on Kuperman's work at 2.1.2 on Page 14 for another look at this topic.

Axelsson begins with a taxonomy of intrusion detection systems. He broke the field up into two main categories: anomaly-based, and signature-based systems. To this taxonomy, we add a third category, specification-based detection. The modified taxonomy better fits the current state of the art in this field.

- Anomaly-based Detection: Anomaly detection searches for events that differ from a norm. This consists of first constructing a sense of what behavior is normal, and then setting thresholds for flagging events as abnormal or anomalous.

- Signature-based Detection: In signature-based detection known intrusive events are analyzed for elements that can be used to uniquely identify them. These elements are formed into a signature against which system events are compared. If a set of system events matches a signature of a known attack, an alarm is raised. This model is both powerful and fragile. The power derives from the ability to accurately detect malignant events from within a huge body of non-malignant events. Signature-based detection typically has a low false negative rate (for events with signatures). The fragility results from the need to keep the false positive rate low. Because these signatures are often compared to huge bodies of data they must be precise; therefore, it is often possible to evade signature-based detectors by only slightly modifying an existing attack [29,30].

- Specification-based Detection: Specification-based detection is defined in [13]. In systems with explicitly defined security specifications, models can be built that will detect any behavior that deviates from those specifications. This has not been a widely used model in the past, primarily because security specifications have not been explicitly defined. The software development community is reaching the point where a high enough level of awareness of the need for such specifications exists; therefore, we will see more and more systems with such designs. As that occurs, this type of detection should become more common. Specification-based detection takes two main forms: default-deny and default-permit.

  With default-deny the system has an explicit and detailed model of the circumstances in which the system is secure. Any deviations from this model are then flagged as alarms. This idea is closely related to default deny security policies [13]. In systems designed to be secure and built around models such as Bell-Lapadula [31] this type of detection system offers great promise. Axelsson placed this model in the anomaly detection category, but it really

straddles both anomaly and signature-based detection, yet does not fit cleanly into either.

With default-permit the system is designed around a specification of explicitly what traces of intrusion are thought to occur uniquely in the search space of the system. Any activity matching an item in the specification is flagged as an alarm. By analogy, these systems explicitly define illegitmate behavior, and all other behavior is considered legitimate. Axelsson describes this as having an explicit correspondence with the default behavior in a default-permit security policy.

Axelsson points out that most of the research systems he surveyed employed techniques from multiple categories. These systems typically try to balance known normal and known abnormal behavior detection techniques. His work classifies the categories into two sets of orthogonal concepts: anomaly versus signature and self-learning versus programmed. We have added specification to the former, so we have anomaly versus signature versus specification. Systems that will be effective in the general case will have to cover all three dimensions.

The remainder of Axelsson's paper classifies intrusive events into three classes: well-known intrusions, generalizable intrusions, and unknown intrusions. It then discusses how these events are related to the taxonomy. The discussion part of the paper ends with a taxonomy of system characteristics similar to that done by Kuperman in 2003 [32].

2.1.2 Categorizing CSM Systems to Derive Audit Source Specifications

Kuperman's Ph.D. dissertation focuses on improvements to event generation and storage systems for information system security [10]. Three areas in particular are relevant to CuPIDS: detection focus area or domain, timeliness of detection, and the categorization of audit source information.

CSM Focus Areas

When categorizing computer security systems Kuperman describes four focus areas into which a system may fall:

- Detection of Attacks: The security system is designed to detect attempts to exploit a vulnerability in the system.
- Detection of Intrusions: The security system is designed to detect the unauthorized use of system resources by outsiders. This category differs from attacks in that it is dependant upon a particular set of users, resources, and authorizations whereas attacks are independent of any particular system or policy.
- Detection of Misuse: The security system is designed to examine the behavior of authorized users in an attempt to detect the unauthorized use of system resources by these users.
- Computer Forensics: In the context of system security, forensics is an attempt to ascertain what actions occurred causing the system to leave a secure state and enter an insecure state.

CuPIDS is intended to be policy agnostic in that it supports all four of the above focus areas.

CSM Time-to-detect

Kuperman also categorized systems by the timeliness of detection. Using his terminology, the operation of a system can be viewed as an ordered sequence of events, where events are actions leading to state transitions. The granularity of detectable events can range from those as fine as program counter changes to those as coarse as power cycling events. Borrowing Kuperman's notation we represent the

set of events taking place in a computer system by the set E. This set contains a subset of suspect events B such that

$$B \subseteq E$$

and there exist events a, b, and c such that

$$a, b, c \in E$$

$$b \in B$$

The notation t_x represents the time of occurrence of event x. For the purposes of this research time measurements are based upon the local system clock as seen by all processes in a SMP system. The notation $x \rightarrow y$ indicates that y is causally dependant upon x and unless noted otherwise we assume the dependance of events under discussion occur in alphabetical order, namely

$$a \rightarrow b \rightarrow c$$

and that

$$t_a < t_b < t_c$$

with a necessarily occurring before b and b before c noting, however, that a is not necessarily the cause of b nor are a or b necessarily the cause of c. We also need a detection function $D(x)$ that determines the truth of the statement $x \in B$. This function is complex and difficult to define correctly—indeed a complete detection function is the goal of most security research today.

Kuperman's categorization describes four major timeliness categories in which detection can be accomplished: real-time, near real-time, periodic and retrospective. It is in the category of real-time and near real-time that CuPIDS offers significant gains over StUPIDS.

- Real-time: Detection of a bad event b takes place while the system is operating and is further restricted to mean that detection of b occurs before any events

dependant upon b take place. Given the stream of system events described above, real-time detection requires the ordering

$$t_a < t_{D(b)} < t_c$$

or alternatively

$$t_{D(b)} \in (t_a, t_c)$$

- Near real-time: Detection of a bad event b occurs within some finite time δ of the occurrence of b. This requires the ordering

$$|t_b - t_{D(b)}| \leq \delta$$

or alternatively

$$t_{D(b)} \in [t_b - \delta, t_b + \delta]$$

- Periodic: Here blocks of event records are analyzed by the security system once every time interval p where p is ordinarily on the order of minutes or hours. The ordering of events is then

$$t_{D(b)} \leq t_b + 2 * p$$

The requirement that detection takes place before the next set of event records is analyzed is needed to ensure that an ever-increasing backlog of security events does not cause the system to fail.

- Retrospective: Detection of bad events takes place outside of any particular time bounds. Analysis operations typically take place using archived events. Many types of security systems that normally operate in the other three time categories can operate in retrospective mode, possibly by using some sort of event replay mechanism.

While no complete detection function $D(x)$ exists, there are a great number of bad events, $B_D = \{b_0, b_1, \ldots, b_n\} \in B$ for which we do have effective detection

functions. Assuming the existence of identical CuPIDS and StUPIDS detection functions, $D_{CuPIDS}(B_D)$ and $D_{StUPIDS}(B_D)$ CuPIDS offers improvements in guaranteed detection time. On a uni-processor system in which the StUPIDS runs as a normal task the soonest it can possibly detect a bad event, b_i, is when a context switching event occurs after t_{b_i} but before t_{c_i} and the scheduler chooses the StUPIDS to run. In the best case b_i involves the execution of a system call or some other blocking event, the scheduler picks the appropriate StUPIDS process to run next, and b_i is detected before c_i can occur. In the worst case the system is compromised before the StUPIDS has an opportunity to run and detect b_i. Other complications include the relative priority of StUPIDS processes to other processes in the system, and even if a StUPIDS process is chosen to run, its portion of $D_{StUPIDS}(B_D)$ may not include b_i. Therefore even though the StUPIDS is capable of detecting b_i it may not do so before the production process is made active again and t_{c_i} occurs. This means that even though $D_{StUPIDS}(b_i)$ exists a StUPIDS can at best claim near-realtime detection with $\delta = CPUQuantum$. In the case of a StUPIDS running on a MP machine, the appropriate monitoring process may be executing at the right time; however, there is no guarantee that this is the case. CuPIDS reduces the uncertainties described above by ensuring, whenever possible, the appropriate monitor is executing, thus offering real-time detection capability.

2.2 Trusted Computing and Related Work

A significant amount of research has been done in using hardware to support and harden a system against attacks or faults. This section reviews a subset of this field.

2.2.1 Dyad: A System for Using Physically Secure Coprocessors

Tygar and Yee were among the first in the field to use secure coprocessors to address a number of security related problems. They present a set of problems and related solutions that take advantage of physically secure coprocessors in [33].

The Dyad project was an exploration of the use of physically secure coprocessors to address a number of security problems. Examples include protecting the security of physically accessible workstations and ensuring tamper-proof accounting and audit trails.

The authors define a secure coprocessor as a hardware module containing a CPU, ROM, and NVRAM. This module should be physically shielded from penetration. While interesting, these anti-tamper mechanisms are outside the scope of our research. The paper describes a number of physical protection measures and attacks against them.

The NVRAM on the module can contain many types of information. Some of the most likely candidates include keys for public-key cryptography systems and trusted bootstrap code. The research describes how system integrity might be ensured throughout the system boot process. In this architecture, the security module gets control of the system before the system bios begins the bootstrap process. It then verifies the integrity of the system bootloader code, the operating system kernel, and all system utilities. As the authors point out, there is no way to prevent a specially designed processor replacement from being used that does not correctly process specific secure OS code.

Another interesting application of this system is the use of cryptographic techniques to authenticate and encrypt system logs, thus helping to secure their integrity. Both [33] and [34] discuss the use of public key encryption schemes as enablers of integrity checking systems that do not rely upon a stored secret.

While systems such as this may help boot the system to a known, secure state, there remain the problems that occur after system startup. Systems such as these only seem to protect against the effects of viruses, or possible system corruption.

2.2.2 A Secure and Reliable Bootstrap Architecture

Arbaugh et. al. describe a means of securely bootstrapping a computer system into a fully functional and trusted state [34]. The authors point out that without some sort of trusted boot process, no guarantees can be made about the security of the system. This applies to operating systems and applications, but is also true for security mechanisms such as intrusion detections systems.

The paper describes a process of initializing a computer system by breaking the system into layers, and then validating the integrity of software running at each layer. The research described also includes a recovery process. This process is invoked when an integrity check on software at some layer fails, and the system needs to restore itself to a trusted state.

In this work, systems are organized into layers to reduce complexity. The bottom layer, which can be quite small, consists of a verified and trusted boot manager. This boot manager is stored in some sort of non-editable storage, and is the first part of the system that is invoked after power-up. This layer loads the next layer from some protected source, verifies its integrity through computing a hash, installs it on the system, and then passes control to that layer. This process continues until the system is fully loaded with a verifiably secure operating system. Finally, control is passed to the operating system and the boot process completes. A characteristic of this process is that the integrity of an upper levels is guaranteed if and only if:

1. integrity of lower levels is checked, and

2. transitions to higher levels occur only after integrity checks on them are complete.

The resulting "integrity chain" can be used to inductively verify system integrity. As mentioned earlier, the boot process may include recovery techniques if a component fails integrity checks. Some of the options available to the system if an error is detected include:

1. Issue warning and continue, or

2. not use component, or

3. recover compromised code from trusted source.

As the authors point out, the last option is the only real possibility for important elements such as IDS.

The research relies upon the assumption that the hardware supporting the secure bootstrapping process is not compromised. In later research this is handled in part by the creation of co-processing modules in tamper-resistant housings. The authors also assume the existence of some public key cryptographic infrastructure. Finally they assume that some trusted source exists from which a corrupted system might recover. The first two assumptions seem valid, but the third may be difficult. In particular, if an attacker is able to compromise a host in a system such that the system needs to recover, it is possible that he can compromise the recovery host as well.

This work relates to CuPIDS in that a mechanism such as this may be useful in ensuring the integrity of the system before any CuPIDS applications are loaded. The use of a co-processor in this fashion is orthogonal to our intended use of co-processors.

2.3 Coprocessor-based Intrusion Detection

There have been a few direct applications of co-processors to intrusion detection. This section summarizes those results.

2.3.1 Secure Coprocessor-based Intrusion Detection

Zhang, et.al. describe how a secure co-processor, similar to that used in the papers reviewed in sections 2.2.1 and 2.2.2 is used to perform some host-based intrusion detection tasks [35]. Similar work is described by Petroni in [36]. This discussion focuses on Zhang's work in which they examine the possible effectiveness of using

hardware designed for securely booting the system to run an intrusion detection system. The benefits from doing so include protecting the IDS processor from the production processor, and offloading IDS work from the main processor onto one dedicated for that task. Possible drawbacks of the approach described include the lack of complete visibility into the actions of the main processor and operating system.

The authors describe coprocessor-based intrusion detection as collecting host data and processing it on software running on the coprocessor rather than on the host itself. A coprocessor will share a set of interfaces with the host that allow it to examine, and possibly modify the state of the host. The specific type of coprocessor will determine which interfaces are available, and therefore what types of intrusion detection are possible. In this research the authors used a PCI bus-based Cryptographic Coprocessor which has a CPU, secure and non-secure memory, and crypto accelerators contained in a tamper-resistant housing. The coprocessor module is designed to support generic security applications, including secure booting—the application most commonly discussed in the literature.

Some of the main advantages of using a separate hardware-based coprocessor for ID are:

- Independence from the host OS: Because the coprocessor is autonomous it is not easily affected by processes running on the host OS. Additionally, the tamper-resistant design used in this research offers integrity protection for the IDS.

- Narrow Interface: The simple and well-defined interface between the host and coprocessor is easier to protect.

- Secure booting: Because the boot processes for both the IDS and host OS can be verifiably secure, the initial state of the host OS is known to the IDS which should simplify its task tremendously.

- Trusted observer: Because the coprocessor maintains its own authentication keys, any authenticated communications made by it can be fully trusted. This may be useful in preventing attacks which attempt to spoof the acts of an IDS.

The authors acknowledge that with their design it is not possible to monitor system calls and key kernel operations with the same level of detail as a native IDS. The problem lies in the lack of interposition with the IDS tasks and host operations such as system calls and kernel ops. With the IDS running on a separate host a different paradigm is needed. Here they propose that the IDS use system invariants as entities upon which to base its analysis. This idea has considerable merit in that the host OS can be booted into a known good state, and the invariant can be monitored from that point forward. The technique used by Zhang and Petroni is to sample the host OS periodically for changes to the invariants.

While CuPIDS does not use a separate co-processor, it may make use of the same system-wide invariant tests as Zhang and Petroni's work. The invariants discussed in the paper are primarily kernel data structures. Examples include *task_struct*, the structure holding information about current tasks in the system, and *inodes*, structures holding information about the file system. The system samples these structures at system startup and task creation, stores a copy in the protected memory space, and then periodically polls the OS for changes. If changes occur in ways that violate an invariant, then an alarm is raised. Other possibilities discussed include off-host virus scanning, and Tripwire-like monitoring [37].

The system as described has a relatively narrow interface between the host and coprocessors. This is both a boon (for the reasons stated above) and a hindrance. By not being an integral part of the OS running on the host, the IDS works by polling, and possibly misses events. For example, it is possible to imagine an attack that does its damage quickly and then restores the system to a state that does not violate the invariants before the next coprocessor polling period. The authors do acknowledge this weakness, and discuss some reasonable methods for reducing these vulnerabilities; however, they do not completely eliminate the threat. This

ties to the limitations of the PCI-based coprocessor and the narrow window between the two processors. Because CuPIDS resides inside the host operating system and shares most system resources it has much simpler access to internal data structures; however, it does not share the protections listed above.

2.3.2 Using Independent Auditors as Intrusion Detection Systems

Molina and Arbaugh describe work on using a secure coprocessor component similar to that described in 2.2.1, 2.2.2 and 2.3.1 [38]. This coprocessor is used to implement a trustable Integrity Verification Tool that can be used to implement functionality such as that in Tripwire [37], but which can not be subverted by compromising the system kernel. The paper describes how kernel attacks can be used to subvert the functionality of such integrity checkers. Examples include using loadable kernel modules that replace lists of items with modified lists that do not include evidence of malicious items. This discussion relates to CuPIDS for two reasons: the methods by which the kernel is compromised provided a fertile starting point for our own investigations, and it does provide a concrete example of where using a trusted coprocessor for IDS adds significant capabilities that cannot be achieved without using a trustable agent.

The authors describe an independent auditing mechanism as having the following properties:

- Unrestricted access: The coprocessor must have access to the devices or systems being audited. This includes peripherals, hard disks and interrupts. The system may need to utilize mutual exclusion mechanisms to prevent race conditions.

- Secure transactions: The means by which the coprocessor gets information should not be monitorable or modifiable by the host processor. This prevents the host from detecting or affecting the activities of the auditor.

- Inaccessibility: The host processor should not have access to the internal state of the coprocessor, nor should it have any control over its execution.

- Continuity: The coprocessor must run from the time the host starts until the auditing tasks are complete. There should be no interruptions in the runnability of auditing tasks.

- Transparency: The activities of the auditing processor should not be visible to the host processor. A possible exception to this might be mutual exclusion issues involving shared items of interest.

- Verifiable software: The entire auditing process should be trusted. This requirement is made possible by using the type of secure processor selected for this research. This requirement does not apply to CuPIDS using commodity SMP hardware; however, it may be feasible in the future if trusted computing architectures become standard.

- Non-volatile memory: The auditing must have some mechanism for recording events that occur even if the entire machine is halted or crashed. This allows for useful forensics, and detection of the cause of crash events.

- Physically secure: This assumes tamper resistance or storage of the hardware in a protective environment. Again, this is handled by the tamper-resistant hardware, but is also a requirement that may be relaxable in our research.

The rest of the paper is a description of research into using a specific coprocessor architecture to implement Tripwire-like [37] functionality in a secure and trusted auditing system.

The authors make some explicit assumptions that certainly apply to the system implemented, but which do not apply to our research. For example, the assumption that the coprocessor hardware must be tamper-resistant is not necessary if the hardware is locked in a trusted environment (as may be the case with high-traffic web

servers). This may open some doors for research as the trusted computing base hardware seems somewhat restrictive. Another possibility is the use of such a mechanism as an auditing platform/trusted base that is used by other forms of IDS running on the machine. It may be possible to bootstrap and then monitor the integrity of other IDS mechanisms not running in the tamper-resistant hardware.

The concepts described are intriguing. The paper outlines concrete vulnerabilities to existing logging mechanisms, and then provides a reasonable and secure mechanism for countering those vulnerabilities. While these concepts are not part of the CuPIDS architecture defined in this document they may be valid components of a future architecture.

2.4 Reference Monitors

2.4.1 Computer Security Technology Planning Study

Anderson's paper is an older study of security needs applied to the AF in 1972 [39]. The focus of the paper is on the development of a secure, open-use multilevel system supporting general applications and development. In this context, he describes the threat posed by a program run by a malicious user, or a supervisory control program [Operating System] acting on the user's behalf, as making a reference to a program or data not authorized for that user.

In an effort to define the requirements for the defense against malicious users, Anderson studied the mechanisms used by such a user to achieve penetration of the system. In an attack, the attacker searches for and exploits flaws in the system that will give her access for which she does not have authority. With this authority, the attacker often exploits other vulnerabilities, escalating his authority until he has supervisory control over the system. With this access, he is then able to access any data on the system to read or modify—including the system code itself. Anderson points out that in contemporary systems (circa 1972), the operating systems run most of their code in supervisory mode with virtually unlimited access to every part

of the system. This code is all of potential interest to an attacker as finding an exploit in it will automatically give supervisory access to the system. As Anderson points out, large bodies of complex code such as an operating system are virtually impossible to validate statically, much less when the dynamic behavior of the system must be considered. There is no practical way to validate that all of the possible control paths in an O/S produce correct results. It is interesting that this problem really has not changed since 1972. Our contemporary O/Ss, while much larger and more complex, still suffer from the same problem [28]. Then and now, many, perhaps most, attacks are feasible because so much of the operating system has unlimited access to the machine—all this code must be trusted. This trust is often misplaced, not only because of flaws in design and implementation, but because most operating systems were not designed to be secure.

Given the situation described above, Anderson defined the following requirements to defend against a malicious user:

- A system designed from the beginning to be secure;
- Adequate access control mechanisms;
- An authorization mechanism;
- Control over all program execution—specifically including operating system functions.

These requirement have the usual meanings. The authorization mechanisms must be able to verify a user's identity. This identity is used to compare attempted accesses to a list of authorizations or privileges granted to that identity. The combination of a mechanism that controls execution of code running on a user's behalf is sufficient to protect the system. Unfortunately, all three of the tasks are difficult enough that they have not been solved adequately in real, common use operating systems.

He hypothesized it is possible to make a system secure by employing a reference monitor that monitors all references to programs or data made by all users. The

monitor is able to compare these references to some sort of access control matrix, and is therefore able to mediate each access. The reference validation mechanism has the following principles:

- it must be tamperproof;
- it must always be involved;
- it must be small enough to be tested (exhaustively if necessary).

For this idea to be viable, the monitor must be implemented correctly, cannot be altered by any other part of the system, and because the monitor must interpret many if not all references the system makes, it must be efficient. This monitor is used to secure sensitive parts of the O/S, and is the basis for evaluating the security of the system as a whole. This idea of a reference monitor is not new; Anderson cites related work by Lampson, Graham and Denning, and others. It is still current however. It appears in the Common Criteria (CC) evaluation criteria in almost exactly this form [40].

This is directly applicable to our work—a central requirement of the CuPIDS architecture is validating system events against a set of reference criteria. Also, this monitor is likely part of the mediation component of a mandatory access control (MAC) system. Future work on parallel architectures such as CuPIDS will implement or extend reference validation monitors in a MAC implementation.

2.4.2 Spy: A Method to Secure Clients for Network Services

> "A fundamental problem in security is to guarantee correct program behavior on an un-trusted computer regardless of a users actions ... dependable security is necessary not only for e-commerce, but also to ensure that, under critical conditions of information warfare, remote clients behave predictably and securely, and cannot compromise the infrastructure." [41]

Lipton researches the above problem by focusing on guaranteeing correct behavior using low-level mechanisms [41]. This differs from our interest in detecting incorrect behavior; however, the two problems dovetail nicely in that guaranteeing correct behavior is unsolvable in the general case (Lipton's proof of this is discussed briefly below), but also in that mechanisms designed to guarantee behavior may prove useful in detecting aberrant system events and actions.

This paper concentrates on intellectual property (IP) rights management, and is therefore focused primarily on securing client applications against misuse of sensitive information—integrity and confidentiality. The authors argue this problem is still quite general. Specifically, integrity in the IP context means ensuring that applications accessing IP content execute only as intended by the IP owner. In this context, there are a number of high level operations that must complete correctly (e.g. counter increments, heap operations etc.). They do not specify what operations are necessary, but rather abstract the high level operation and specify the need that it is always performed correctly.

The subject domain is a client-server environment with a trusted server providing protected content to the client. The client is secured through an "inverse pyramid" built starting with a small, trusted "spy" secure, tamper-proof module. This is a minimalistic approach, one the authors feel is necessary in that if it is to be adopted by business, cost is a significant factor.

The authors define a client as being completely untrustworthy in software—that is that the client (to include the controlling OS) can stop or start at any time, and can copy or modify the contents of memory or registers at will. They use this definition to prove the need for trusted hardware support. A sketch of the proof is that a program (perhaps the O/S) can interrupt the client software after the information has been decrypted. It can then copy the information, and return control to the software IP client. The software client has no way to detect that this has occurred, much less do anything to prevent it from occurring.

Solutions to this problem are reduced to two key sub-problems: maintaining the secrecy of cryptographic keys, and providing atomicity to the cryptographic operations used in the protection scheme. The way they handle these two problems are of direct interest to our research. The first hardware assumption is that there exists a "spy." This spy is essentially a CPU or co-processor that exists in a tamper-proof environment, but which can interact with the external, un-trusted environment. This spy must be the master—not slave—of the production CPU. This requirement stems from the need to control the execution of the main CPU, while not being vulnerable to being controlled by the main CPU. In the IP domain this prevents the main CPU from suspending the spy while decrypted data is copied. In the ID domain, this will prevent an attacker from stopping or modifying the behavior of the ID processor.

In Lipton's work, the spy by itself is not sufficient. An example given is a client running in a compromised OS. The spy decrypts the data and places it into memory. The OS then can pause the client (perhaps during a normal time-slice interrupt), copy the decrypted data from memory, and then allow the client to resume execution. The solution to this is what the authors call a "Two Minute Warning." This is a special, highest priority hardware interrupt that occurs a fixed amount of time before any other interrupt occurs, including normal time-slice expiration. The intent is to give the client time to move at least small amounts (a few pages) of decrypted content into an encrypted and hashed buffer before the OS regains control of the system (the encryption prevents the OS from using the data and the hash allows detection of tampering). An alternative is to have the spy put the main CPU to sleep during all encryption/decryption actions. This assumes that the spy has sufficient computing power to perform these actions in the microseconds or milliseconds it is able to stop the main CPU.

This work has relevance to CuPIDS in the case where the ID coprocessor is less powerful than the production processor, or if there is a computationally expensive task that needs to be run to determine if it is safe to let the production processor continue execution.

2.5 Non-coprocessor Hardware Security Mechanisms

2.5.1 Enlisting Hardware Architecture to Thwart Malicious Code Injection

Lee et. al. discuss buffer overflows, and describe a hardware mechanism for detecting and thwarting malicious code injection involving procedure return address corruption (stack smashing) [42]. This paper was particularly interesting in that buffer overflows are one of our primary targets for low level IDS. The authors point out that buffer overflows really should not be a threat—we know as a science how to prevent the majority of them from being created. However, they do remain one of the most significant sources of vulnerabilities in systems comprising about 50 percent of the vulnerabilities reported by CMU's CERT over the last eight years [5]. Multiple sources report that attacks against these vulnerabilities cost society billions of dollars every year. Lee proposes that this cost more than justifies the slight extra expense her scheme would add to future processors.

Most buffer overflow exploits involve an attacker corrupting the procedure return address by passing too much information into a program's input. The overlarge data includes either executable attack code for the target system, or perhaps the address of code already on the system, and a landing pad that replaces a procedure return address on the stack (see Figures 2.1 and 2.2, both taken from [42]). When the processor finishes a procedure, it pops the corrupted return address off of the stack and resumes execution at that address. If the attacker included code in the exploit, that code is then run, possibly compromising the security of the system. If the attacker uses pre-existing code, then this code is executed, also putting the system at risk. The authors failed to note that the exploit code runs with the privilege of the vulnerable program, which in many cases is supervisory, or root. This automatically gives the attacker root access to the compromised system.

Lee describes a fair amount of past research in protecting the stack. None of the solutions are practical in all cases. The techniques range from storing the stack in non-executable pages to the use of safe programming languages. Non-executable

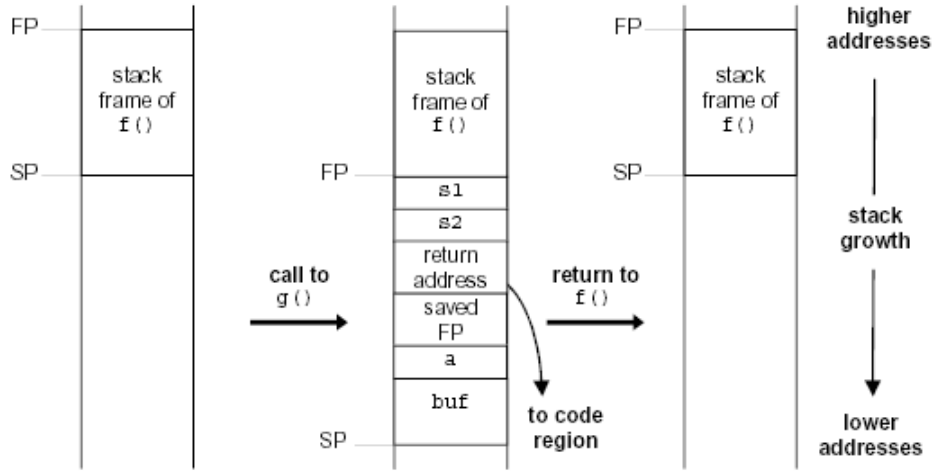

Figure 2.1. Example Stack Operation Before Corruption

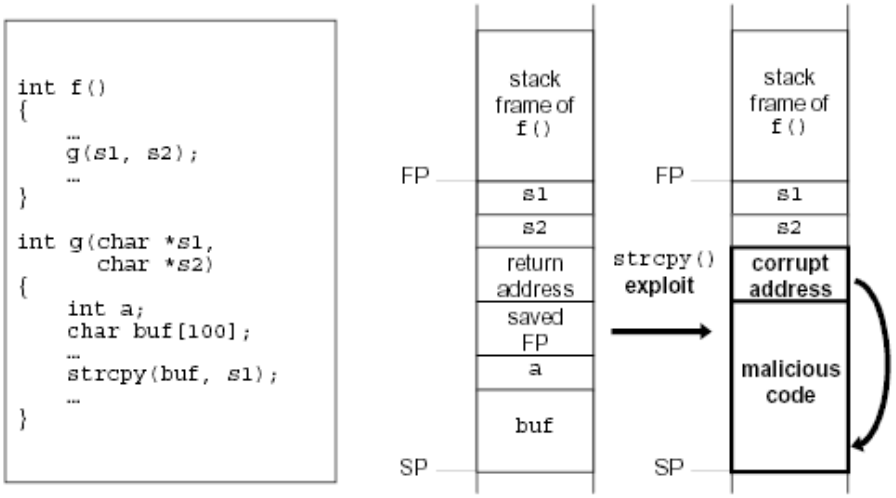

Figure 2.2. Example Stack Operation After Corruption

stack pages prevent attackers from injecting their own malicious code into the stack, but do not prevent them from executing arbitrary existing code on the system. Additionally, this technique does not preserve compatibility with operating systems and applications that employ executable stacks. Another choice is the use of safer dialects of C and C++. An advantage of this method is that most buffer overflows result from the use of dangerous constructs in the C programming languages. It is possible to write safe programs in C, and the community as a whole has not shown much inclination to engage in safe coding practices. In addition to the difficulty involved in getting people to use new languages, there remain the existing buffer problems in legacy code. Therefore it seems unlikely that alternative languages are a good short or medium term solution to this problem. A third choice is the use of static source code checkers. While these do not require language changes, they do have to be used to detect problems, and the techniques used in static checkers are not perfect. Lee reports that both the false positive and false negative error rates are high with static checkers, which likely leads to under-use of these tools in the community.

A fourth technique involves modifying the compiler so that it places a randomly generated "canary" value next to the stack pointer in memory. This same value is stored in a general-purpose register. When the procedure completes, the canary is compared to the register value to determine if a buffer overflow has occurred. While this does make the corrupting problem more difficult, it is possible to overflow a buffer and corrupt other variables—including pointers to functions.

SPARC architectures offer a fifth form of protection in the form of register windows. By placing stack space into a register window, limited protection is afforded a return address; however, once the register window is spilled to memory the data stored there is vulnerable to attack. Both of these last two techniques do raise the bar considerably in terms of making an attack more difficult; however, neither of them is completely effective, and the SPARC version is only available on SPARC-like architectures. Finally, there are run-time defenses that intercept unsafe library

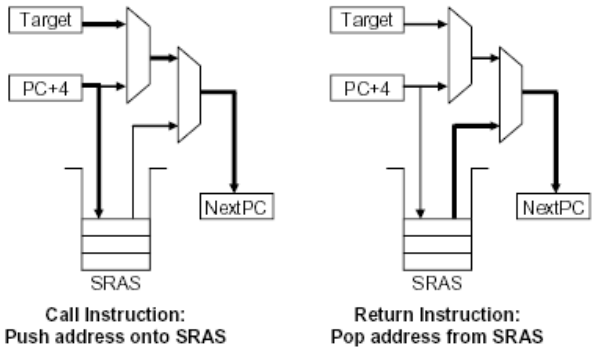

Figure 2.3. Hardware SRAS Operations

calls and automatically perform bounds checking, or that save a copy of every function and return address on the heap, then insert instructions that operate out of the heap instead of on the stack. In the former case, only certain C library functions are protected, and then only if applications are dynamically linked. In the latter case, there is a substantial performance degradation, both in terms of memory use and in extra code execution—a real problem for embedded applications.

As Lee points out, none of these techniques are perfect. What she proposes is a multi-layered approach that combines hardware changes with static and dynamic software defenses. Static defenses include using safe programming languages, statically analyzing code, and using security code inserts (canaries) at compile time. Dynamic software defense include library wrappers like those described above. These measures do make sense, but do not seem to really solve the problem (for the reasons she describes). The value of this research lies in the hardware solution.

In all instruction set based computing architectures, procedure call and return instructions are clearly identifiable. Because the processor can identify these instructions, it can operate on them at runtime. Lee suggests augmenting processors with a LIFO secure return address stack (SRAS). As the stack is in hardware, the processor does not have to trust external memory such as the stack to protect its contents from buffer overflows and related attacks. As execution encounters a subroutine call, the stack frame is constructed normally, but when the return address is placed on the

stack, it is also pushed onto the SRAS. When a return instruction is encountered, the value on top of the stack is popped and compared with the value stored on the stack as shown in Figure 2.3. If the values are the same, then no stack corruption has occurred and execution can continue normally. If the values are different, then a buffer overflow has occurred, and was caught by the hardware mechanism. In this case, the processor can either replace the corrupted return address with the correct one, or signal an error identifying the corrupted process. The operating system can then shut down the offending process, log errors etc. The former solution seems particularly risky in that the buffer attack has likely already affected execution of the called subroutine. A third alternative would be for the processor to halt, but as this would lead to easy denial of service, the second alternative is the best choice.

The hardware support for this scheme is minimal: the SRAS stack, and instruction support for its use. As the stack is finite in size, the processor will also need to raise a signal that will inform the OS that some of the stack contents need to be spilled to a protected memory segment. Additionally, the OS will need to be able to respond to buffer overflow errors as caught by the processor. Finally, because each thread in the system must have its own stack, the context switch routines must be able to save and restore the SRAS.

Lee points out that not all programs use LIFO control flow. For example, C++ exception handling allows jumps out of arbitrarily deep stacks. This is a significant problem for the SRAS. The authors propose four solutions ranging from not allowing non-LIFO code to execute (which eliminates many legacy programs), to recompiling programs to not contain non-LIFO behavior (not useful if you do not have source), to dynamically modifying existing binaries to correct non-LIFO behavior (probably not an easy task), to finally disabling the SRAS for non-LIFO code (which requires determining what code is non-LIFO).

The performance impact of the SRAS technique should be negligible. The two instructions that modify the stack should not have a performance hit, so the only

slowdowns will occur when the stack overflows or is reloaded from memory, and during context switches.

CuPIDS addresses the buffer overflow problem by using parallel monitoring of buffer accesses. It does not require hardware changes, but does, at least in the current architecture, require programmer involvement in determining when a buffer is being accessed and its parameters. We found this paper to be interesting in that we explored how to accomplish the author's goals using software-only solutions based upon parallel monitoring.

2.5.2 Buffer Overflow Taxonomy

Wilander discusses a taxonomy of buffer overflow types [43]. Because the goal of the attack is to change flow of control for the currently executing process a code pointer is changed. There are four types of code pointers that are abused by current buffer overflow attacks: the return instruction pointer on the stack, the old base pointer on the stack, function pointers on the heap, in the BSS, in the data segment, or on the stack, and long jump buffers allocated on the heap (setjmp/longjmp pairs). CuPIDS is designed to address all of the classes of attacks described by Wilander.

2.6 Virtual Machines

Garfinkel and Rosenblum discuss a novel approach to protecting IDS components [44]. They use a virtual machine monitor (VMM) to separate the IDS from the monitored host for greater attack resistance. This approach has the benefit of largely isolating the IDS from code running in the virtual host. The authors state that they still have excellent visibility into the host's state; however, no evidence is presented that supports this statement.

The VMM approach has much in common with the reference monitor work discussed in Section 2.4 in that it provides a means by which the IDS can mediate access between software running in the virtual host and the hardware. It can also

interpose at the architecture interface, which yields a better view into the system operation by providing visibility into both software and hardware events. A traditional software-only IDS does not have this advantage.

Of course, the IDS running in the VMM only has visibility into hardware-level state. This means that the IDS can see physical pages and hardware registers, but must be able to determine what meaning the host OS is placing on those hardware items. The relationship between CuPIDS and this work is similar to the cryptographic co-processor-based work in Section 2.3.

2.7 Debugging

An example from the separate runtime error checking body of research is that done by Patil and Fischer [45] on detecting runtime errors in array and pointer accesses. They point out that including runtime error checking may slow applications by as much as 1000%, which is very expensive given that most runs of a well-tested program are error free. Therefore once debugging and testing is complete, runtime error checks are disabled before the code is placed into production use. While this makes sense from a performance perspective, it is dangerous because errors that may have been caught by those runtime checks go undetected, potentially causing severe damage. The authors responded by creating guard programs that model the execution of the production program, but only at the pointer and array access level. The guards include all runtime checks on pointer and array bounds and were capable of detecting many runtime errors that evaded the software testers during development. These guards were run as batch processes using trace information stored by the production process. The paper also discussed having the guard run on a separate processor or as a normal process, interleaving execution with the production process. The runtime penalty perceived by the user was typically less than 10%. We use the idea of exporting runtime checks to a shadow process; however, our work differs from theirs in that we focus on real-time monitoring of the actual

memory locations in use by the production process as well as a much larger set of monitoring capabilities.

2.8 External Modeling

Research into performing intrusion detection via external modeling of application behavior is active as of this writing. The recent work done by Haizhi Xu et al. in using context-sensitive monitoring of process control flows to detect errors is an example of external modeling [46]. They define a series of "waypoints" as points along a normal flow of execution that a process must take. They focused their efforts on the system call interface and demonstrated results in detecting attempts to access system resources by a subverted process. CuPIDS makes use of a similar idea to their waypoints in its checkpoints: those points in both the interactive and passive systems where CuPIDS is notified of events in which it is interested; however, CuPIDS checkpoints are much finer-grained and are generated within the production process as well as its interaction with the external environment. As an example, CuPIDS uses function call entry and exit information to perform rough granularity program counter tracking and validation as well as model a program stack for use in detecting illegitimate control flows within a process code segment.

Related work by Feng et al. [47] describes novel work in extracting return addresses from the call stack and using abstract execution path checking between pairs of points to detect attacks.

Finally, Gopalakrishna et al. [48] present good results in performing online flow- and context-sensitive modeling of program behavior. Gopalakrishna's Inlined Automaton Model (IAM) addresses inefficiencies in earlier context-sensitive models [49, 50] by using inlined function call nodes to dramatically reduce the non-determinism in their model while applying compaction techniques to reduce the model's memory usage. Using an event stream generated by library call interpositioning IAM is shown to be efficient and scalable even in a StUPIDS architecture. The techniques

used by IAM fit naturally into the CuPIDS architecture. The model simulation can be run as a CSP, getting its inputs from the CuPIDS event streams.

2.9 Focused Intrusion Detection

Zamboni's doctoral dissertation describes an ID architecture based upon embedding focused detectors in vulnerable components of the operating system and applications [51]. He notes that host-based IDS has usually been developed using large, broadly focused components. These components impose a significant load on the system upon which execute. Furthermore, because they run as applications they may be subject to tampering or disabling by an attacker. Also, as noted by Axelsson and Kuperman [9, 10], IDS have typically obtained information about host behavior through indirect means, such as audit trails or network packet traces. This potentially allows intruders to modify the information before the intrusion detection system obtains it, making it possible for an intruder to hide his activities.

Zamboni's work, the Embedded Sensor Project (ESP), demonstrates it is possible to perform intrusion detection using small sensors embedded in a computer system. ESP sensors are embedded in the operating system code They look for signs of specific intrusions, performing target monitoring by observing the behavior of the system directly, instead of through an audit trail or other indirect means. Because they are built into the code of the operating system and its programs, they do not impose an extra load on the host they monitor except when the vulnerability they protect is accessed. The focused monitors in CuPIDS are similar in that they target specific vulnerabilities; however we place our sensors in a separate application, give them access to the memory of the protected application, and execute them in parallel with the monitored code.

2.10 Conclusions

In this chapter we briefly discussed research related to ours and discussed the role of our research in solving open problems in this field.

3 AN ARCHITECTURE FOR INTRUSION DETECTION BASED ON PARALLEL COMPUTATION

In this chapter we briefly describe the CuPIDS architecture, discuss its defining characteristics, the hardware and software architectures, the capabilities required by the architecture, a set of protective activities, and finish with discussion of its strengths and weaknesses.

3.1 Architecture Overview

The basis of the CuPIDS architecture is parallel monitoring of the activities of a process by another process as the first process executes. Figure 3.1 graphically depicts a high-level overview of the architecture. CuPIDS is designed around a shared-resource, symmetrical multi-processing (SMP) hardware foundation. As seen in the figure, programs using the architecture are divided into two components, the protected application and a shadowing application. The CuPIDS architecture is event-driven. As the protected process executes it generates a stream of events based upon its activities. These events are used by the shadowing process to choose specific monitoring actions. The overlapping use of memory depicted in the figure is used to illustrate that nearly all the protected process state is available to the shadow process. This enables non-intrusive security monitoring while the protected task executes. Finally, the shadowing process is able to control the activities of the protected application. This control capability allows CuPIDS to protect the application and system when illegitimate behavior is detected.

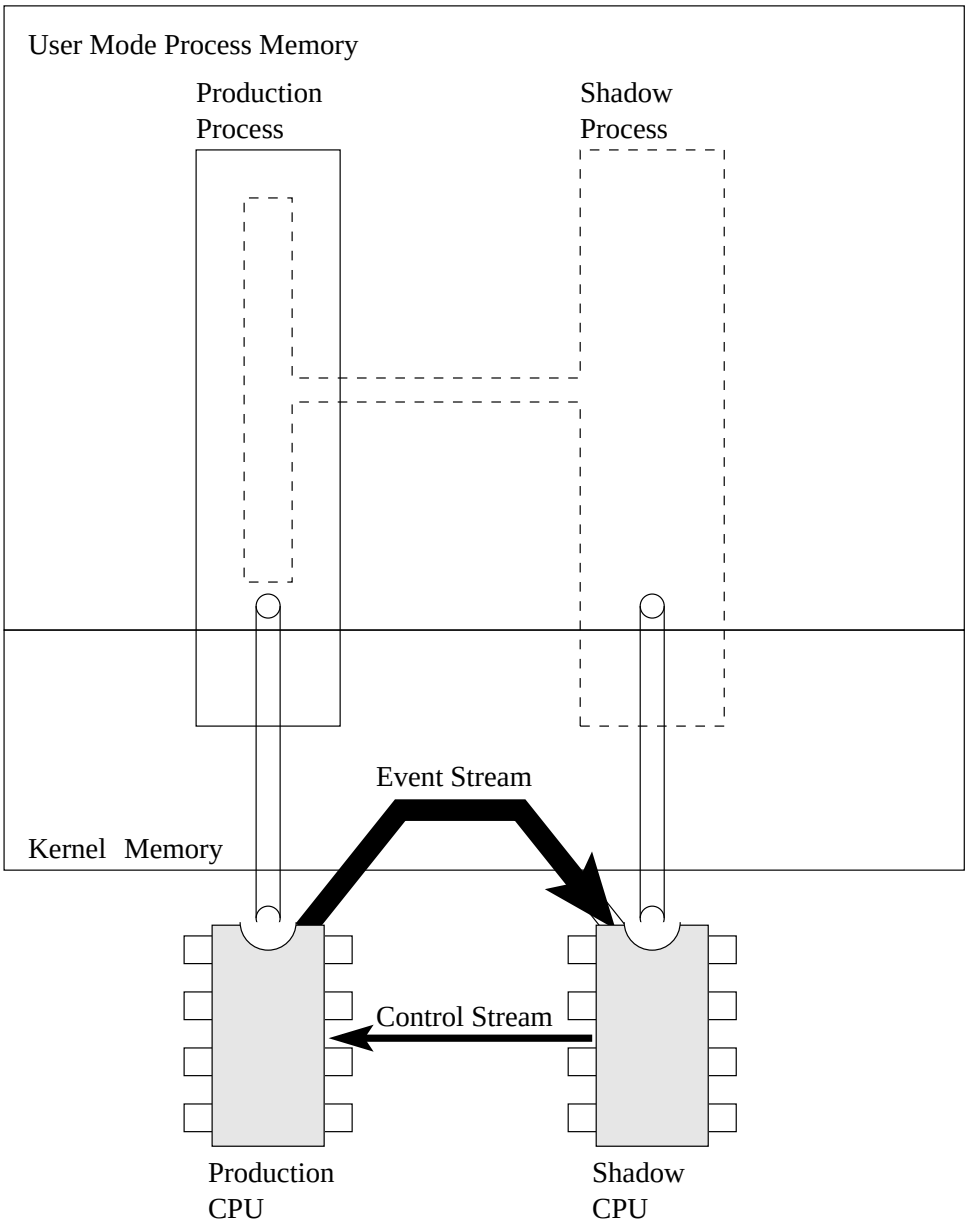

Figure 3.1. High Level Overview of the CuPIDS Architecture

3.2 Distinguishing Characteristics of the CuPIDS Architecture

The CuPIDS architecture has the following main characteristics:

1. Hardware resources are dedicated solely to security tasks. In the current architecture this characteristic applies primarily to central processing units (CPUs).

2. All of the monitored process' state is available to the monitoring tasks.[1]

3. The primary detection tasks are specification-based in nature. This allows us to explicitly use knowledge about how the protected application is intended to operate in detecting incorrect behavior. This also ensures our false positive error rate is zero, which affords us opportunities for corrective responses.

4. Security monitoring is done while the monitored tasks executes.

5. Because CuPIDS monitors run in parallel with attack code, any information arriving or analysis completing during a quantum in which the attack code is executing can be handled and acted upon immediately, rather than waiting until the IDS gets a chance to run. This may allow the security system to stop attacks from damaging the system such that the IDS cannot run.

6. The parallel monitoring is not instruction-by-instruction. Instead CuPIDS tracks the execution path of the protected process and ensures the appropriate monitoring code is running in parallel with the production code it protects.

7. CuPIDS is an event-based architecture in which the execution state of the monitored process is communicated to the monitoring processed via a series of events. Some event flows can be automatically generated, some require programmer involvement.

8. Monitoring occurs synchronously with the protected application based upon event flows.

9. Monitoring can occur asynchronously with the protected application. This means an attacker cannot know when or where monitoring is occurring.

[1]Given the processor architectures available at the time of this writing, only the internal processor state in unavailable during monitored process execution. This is because the instruction pointer register and other internal hardware state of a processor is not visible to another processor in the same SMP machine.

10. CuPIDS runs inside the host OS, and has complete visibility into the state of the OS as well as the complete state of the process (simply by forcing a context switch on the production CPU the monitored process can be suspended and examined.) This level of visibility is much greater than that afforded the virtual machine and crypto-coprocessor researchers.

11. The current CuPIDS architecture can be constructed using commodity hardware and modified commodity software.

The characteristics listed above are closely bound to each other in the architecture. As discussed in Section 1.2 we leverage the wealth of information about what a program is allowed and not allowed to do in protecting it from attack. While many programs do not have explicit security specifications there is a growing trend in defining such for new, mission critical applications [13, 40]. Building upon the debugging-based work in [45], summarized in Section 2.7, we define specification-based invariant tests for data structures in the protected process, embed those tests in a process that has visibility in the protected process' memory regions, and ensure the tests are run on a CuPIDS CPU while the protected data structure is being accessed by the protected process on a production CPU. It is in this domain that CuPIDS offers real-time detection guarantees [10].

Other, more traditional IDS capabilities are specified in CuPIDS; however, the architecture focuses on an area that will allow us to gain the most from parallelization. Merely taking a traditional StUPIDS and running it on its own CPU would have some benefit, primarily in reducing the average time to detect erroneous or illegitimate activity because it is no longer competing with production processes for CPU time; however, unless the specific activity in the IDS is tied to the activity in the monitored process these gains are still probabilistic in nature—there cannot be guarantees of real-time detection.

3.3 Symmetrical Multi-processing Foundation

CuPIDS is intended to operate using the facilities and capabilities afforded by a general purpose symmetrical multi-processing computer architecture. In such an architecture there are some number, typically even, of central processing units, all of which share a set of resources such as memory and hardware devices via a common system bus. Generally the CPUs are all capable of running the same sets of tasks.

A two CPU example is shown in Figure 3.2.

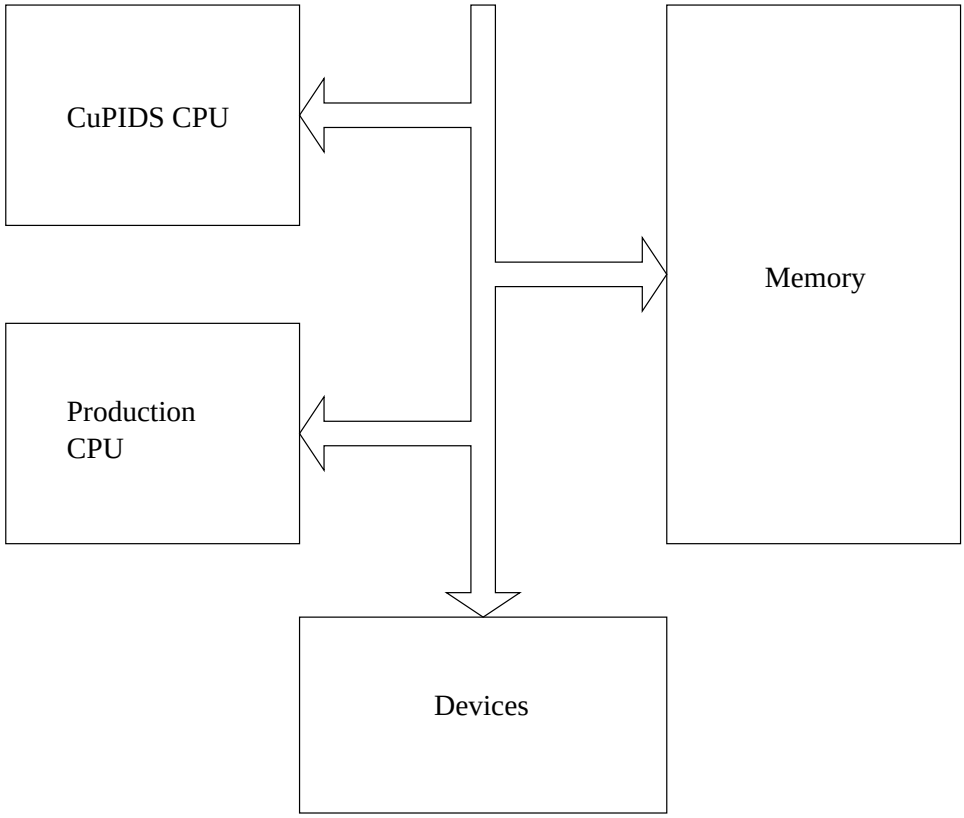

Figure 3.2. Basic Hardware Architecture with CuPIDS and Production CPUs Identified

3.4 Software Architecture

Common operating systems such as Windows, Linux, and FreeBSD running on SMP architectures use the CPUs symmetrically, attempting to allocate tasks equally across the CPUs based upon system loading [28]. CuPIDS differs from these architectures in that at any point in time one or more of the CPUs in a system are used primarily or exclusively for security-related tasks. This asymmetrical use of processors in a SMP architecture is a significant departure from normal computing models, and represents a shift in priority from performance, where as many CPU cycles as possible are used for production tasks, to security where a significant portion of the CPU cycles available in a system are dedicated solely to protective work.

3.4.1 Overall Software Architecture

One possible CuPIDS software architecture is depicted in Figure 3.3. The dark components represent production tasks and services running on one CPU while the light components represent the CuPIDS monitors running on a separate CPU. The regions of overlap depict CuPIDS ability to monitor the resource usage of production components.

The operating system as well as user processes are divided into components that are intended to run on separate CPUs. The intent behind this separation is twofold: performance and protection. We are concerned with two performance measures: speed and completeness of detection, and the runtime penalty imposed by the security system on the production processes. By ensuring the processes responsible for detecting bad events are actively monitoring the system during periods in which bad events can occur, we provide a real-time detection capability (using Kuperman's notation as defined in Section 2.1.2). The system protection derives in part from the ability to detect bad events as they occur but before the results of these events can cause a system compromise. Additional protection will come from the separation of

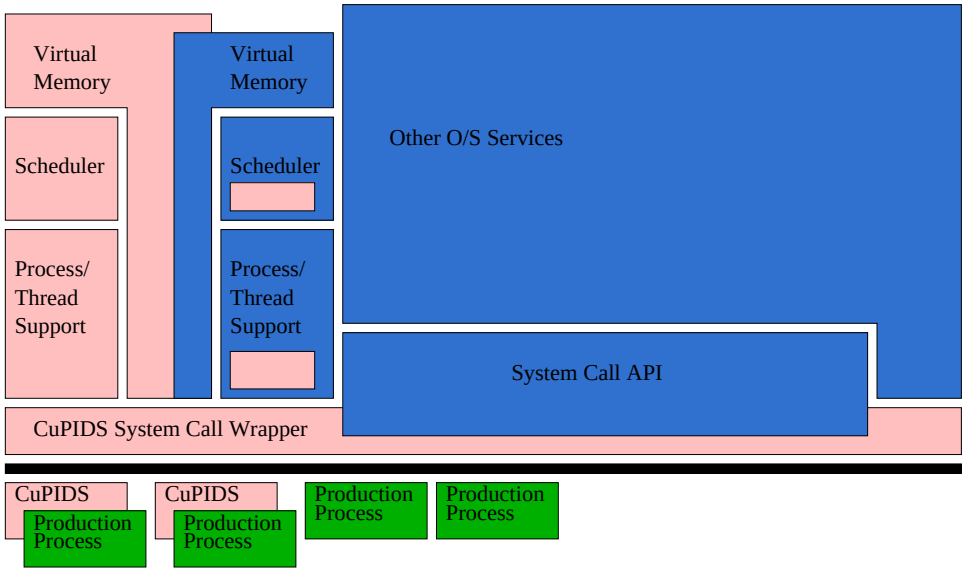

Figure 3.3. Basic Software Architecture

security monitoring code and data segments from the memory segments used by the operating system and user programs.

Supporting the monitoring process visibility into the monitored process state necessitates changes to the virtual memory subsystem. Monitoring the runtime activity of the monitored process is accomplished by CuPIDS connections into the system call and runtime interfaces, and the parallel monitoring requirement and control over the activities of the monitored process are handled by CuPIDS changes to the scheduler and thread support architecture.

3.4.2 Production and Shadow Processes

A program intended to operate in CuPIDS is divided into two components, a CuPIDS monitored production process (CPP) and a shadowing CuPIDS process (CSP) as depicted in Figure 3.4.

As the figure shows, CuPIDS processes differ from the traditional process paradigm in the asymmetric sharing of memory between the CSP and CPP. The CPP

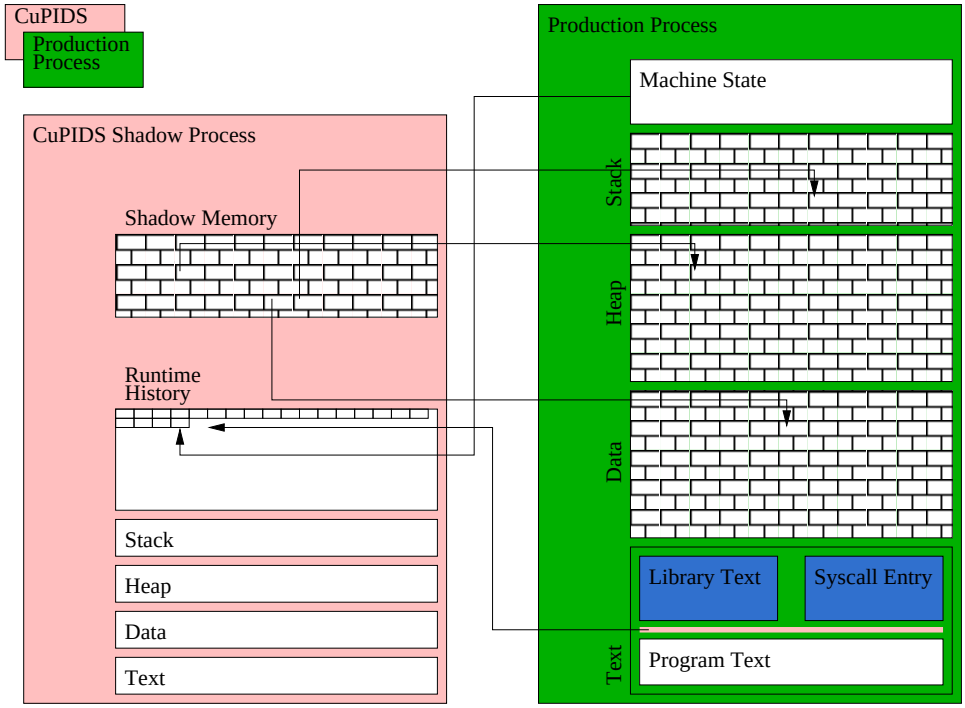

Figure 3.4. CSP and CPP Details

is a normal process and contains the code and data structures that are used to accomplish the tasks for which the program is designed. It may also contain code and data structures with which information about the state of the running process is communicated to the security component. In addition to the normal process code and data structures, the CSP's virtual memory is modified to contain portions of the CPP's virtual memory space (depicted in the figure as Shadow Memory). This allows the CSP to directly monitor the activities of the production component as it executes. The monitoring performed by CuPIDS is both interactive and passive. In an example of interactive monitoring, the CPP informs the CSP when it is about to enter a critical region or access protected variables. The CSP then tests the state of the CPP against invariants about what the CPP state should be. Passive monitoring takes place without the active involvement of the monitored process; an example of this may be frequently taking snapshots of the process state and verifying the

code being executed is legitimately part of the process. Another passive monitoring capability includes fine-granularity execution environment introspection, in which a process specification is created that describes what library and system calls are used, from where in the process' text region each call is made from, and possibly normal parameters and return values. This information is used by CuPIDS processes running on a co-processor to validate the CPP operation, and does not interfere with or delay the processing of library or system calls unless the security system detects a problem.

Additionally, the CSP may gather CPP runtime history data and use this for ID analysis, automated response or recovery, or forensic analysis.

3.4.3 Events

The CSP and CPP pairs are the core of the CuPIDS architecture. The CSP requires knowledge about what the CPP is doing to accomplish its goals. The current architecture embeds event generators in the CPP. The types of events are described below. As each generator is executed a message describing the particulars of the event is created and sent to the CSP.

Variable Creation/Use/Deletion

CuPIDS specifically leverages knowledge about how a program is intended to behave. One area in which this capability is used focuses upon lifecycle events of a program variable in the CPP. The flowchart in Figure 3.5 depicts how CuPIDS performs interactive monitoring of a protected variable. Here the protective process is notified of the production process' entry into a region in which a watched variable may be modified. This notification may come from instrumentation embedded in the production process, or it may result from the protective process setting a tripwire in the instruction stream of the production process or on the variable's memory location. The pseudo code illustrated in Figure 3.6 shows examples of the variable

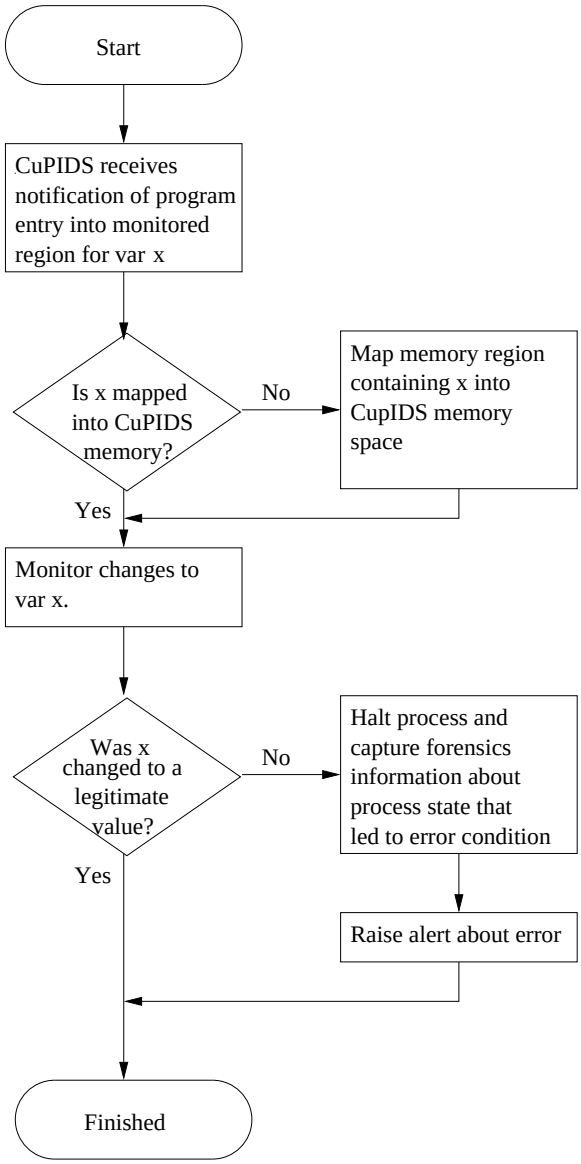

Figure 3.5. Variable Protection Flowchart

protection instrumentation embedded in the CPP (the `CuPIDS_var...` calls invoke the CSP notification mechanism), while Figure 3.7 depicts the actions taken by the CSP upon notification that variable access is complete.

Ideally, the programmer creating and using the variable knows what values the variable can legitimately take on; these values are used by CuPIDS as pre- and post-

```
...
CuPIDS_var_create(varID=0,
   var_address = &protected_var);
...
CuPIDS_var_access_begin(varID=0,
   var_address = &protected_var);
cin >> protected_var;
CuPIDS_var_access_end(varID=0,
   var_address = &protected_var);
...
CuPIDS_var_delete(varID=0);
...
```

Figure 3.6. CPP Variable Protection Code

condition invariant tests used to validate the changes or attempted changes to a variable. Other inputs are possible. For example, the size of a buffer is known when it is created, and this information can be used by the protective process to determine if data placed into a buffer overruns the ends of the buffer. If the changes to the variable were legitimate, the production process is allowed to continue execution. If not, the protective process will take some action ranging from annotating the problem in a log to halting the production process or potentially the entire system. In any case, it will likely capture forensics information about the state of the production system leading to the erroneous value being entered into the variable and the changes that took place.

Checkpoint Events

CuPIDS abilities stem from its ability to track the execution of the protected process. Ideally, the CPP's instruction pointer is visible to the CSP at all times;

```
bool CheckVar0PostCondition(void *var){
  if(*var > 42 || *var < 21)
    return false
  else
    return true
}
...
//Msg access end handler
if(varID = 0)
 if(!CheckVar0PostCondition(var_address))
    RaiseAlarm();
...
```

Figure 3.7. CSP Variable Protection Code

however, the processors available in commodity machines do not have this ability. To achieve an approximation of this capability we define a mechanism consisting of a set of software checkpoints in the CPP's code and an efficient inter-process communications system (see Section 3.4.3 for details).

The checkpoint mechanism will provide a stream of checkpoint passing events in the CPP, this stream is monitored by the CSP for events indicating the CPP has gone off course. Ideally the CPP writes each event into a first-in first-out (FIFO) queue and the CSP reads the events as they become available.

What forms of checkpoints are useful: There are at least two possibilities.

1. Notification only. A message is created and stored/sent notifying the CSP that a checkpoint location has been encountered.
2. Notification plus blocking. A event notification is sent and the CPP raises SIGSTOP or blocks on a mutex or other synchronization construct. When

the CSP has a chance to validate the state of the CPP it can handle any errors or resume the CPP as appropriate.

Determining where to set checkpoints: This parameter allows many possibilities for tuning. Known "dangerous" parts of the program can be finely inspected by placing many checkpoints while less interesting parts can be more sparsely checkpointed.

1. Before and/or after key data structure modifications. Ideally the program structure is such that a notification-only checkpoint is passed before key data modifications and then a matching blocking checkpoint is located after the modification is complete but before any code that makes use of the modified data in a dangerous way is entered. This allows the CPP to run as long as possible while the CSP is validating the modified data structure. This ability allows the CSP to perform real-time detection while minimizing the performance on the CPP.
2. In honeypot code (either code that should not be executed or should not be entered except from particular locations. Honeypots are an idea that originated with Stoll [52].)
3. Randomly located. It may also be possible or useful turn these off or on using a cryptographic key process of some sort (akin to the known k of n schemes). This may make it more difficult for an adversary to spoof a legitimate event stream.

Enabled/Disabled Events

It may not be possible or desirable to enable all possible events. Therefore it may be useful to allow the CSP to enable or disable specific events or blocks of events at runtime. Advantages to doing this include the ability to tune the level of inspection based upon factors such as timing (beginning of software operation versus after it

has been running for a while), external factors such as newly discovered threats, and increased attacker uncertainty (randomly turning on events makes it more difficult for the attacker to work safely). A possible mechanism that would allow this behavior is creating a fully instrumented binary, constructing in the CSP's inputs a map of all the event generators, and then disabling the unneeded ones by writing NOPs into the text in place of the calls to the function entry and exit routines. To re-enable the event the CSP can write the appropriate instructions back into the text.

3.5 Background Capabilities

In addition to the direct monitoring of the CPP performed by the CSP, CuPIDS has a number of background capabilities that support and augment the CSP's capabilities. These include the ability to intercept and direct low-level system activities such as interrupts and signals, controlling the system scheduler to enforce the segregation of the CuPIDS and system CPUs and ensuring that whenever a CPP is chosen to run, its associated CSP is also placed on the CuPIDS' CPU. Additionally, CuPIDS provides an efficient, interrupt-based, communication interface for moving event records from the CPP to the CSP running on a different CPU.

3.5.1 Inter-CuPIDS Communications Mechanisms

Both theory [10, 20, 28, 53] and early experimentation with CuPIDS highlight the need for efficient means of moving data between parts of CuPIDS. The following characteristics were determined to be desirable for the communications mechanism:

- Exclusivity: The communications mechanism should not be used by any other system in the host. This ensures that event reports can be obtained from a single source without having to filter extraneous messages. Additionally, the messages moving through the communications medium should not be visible to non-CuPIDS processes.

- Efficiency: Because large numbers of events are generated by the CuPIDS processes, the communications mechanism needs to use a minimum of resources in terms of memory and CPU. Also, events need to be available to monitoring tasks soon as possible after they are generated.

- Security: It should be difficult for an attacker to view or modify messages moving through the system. It should also be difficult for an attacker to disrupt the flow of messages by modifying the messaging system. Finally, it should be difficult for an attacker to insert messages into the communications system. This requirement protects the system against spoofed messages and flooding-based denial of service attacks.

3.5.2 Memory Mapping

The CuPIDS architecture requires that the CSP be able to directly monitor the virtual memory space of the CPP. We added a system call to FreeBSD that takes an address in the CPP process VM space and maps that address into the CSP VM space. The mapping is asymmetrical in that the CSP is aware of the shared memory but the CPP is not. The normal set of read and write protections can be applied to a mapping. Some of the self-healing and forensics tasks require the ability to save old state, thus we provide an ability to make copy on write (COW) mappings of memory locations. This allows for efficient copying of modified pages using the kernel's existing and efficient VM manipulation mechanisms.

One use of this capability involves making periodic copies of the CPP's VM map. Changes to the map as the CPP executes can be used to establish patterns of operating behavior as well as a way to detect deviations from "normal" behavior. Another use of this capability marks the CPP's copy of the VM map entries as copy on write so that CPP changes to pages are captured while preserving the original memory contents. Keeping a rolling set of memory snapshots such as this may allow

for forensic analysis of CPP behavior as well as a means of rolling back changes should illegitimate activity be detected.

In situations when it is not possible or desirable for CuPIDS to perform validation in real-time, this capability affords CuPIDS some flexibility in that invariant testing can be deferred for some or all of the possible tests. For example, if it is critical that the CSP verify postconditions quickly but precondition testing is still needed then precondition validation may be deferred.

3.5.3 Interrupt Routing

Hardware signals and interrupts can be intercepted and routed to the CSP for handling. Some ways that this may be useful include:

1. Detecting errors in the CPP. Incoming signals can be noted before forwarding them on to the CPP.

2. Portions of the CPP VM structure can be marked so that the CSP is notified that the CPP is accessing that page. This capability may be used to provide the CSP with information about what the CPP is doing without requiring input from the CPP.

   (a) Certain sets of variables can be placed into segments that when accessed will cause a page-fault event to occur. The fault handler will notify the associated CSP and clear the fault. Because the fault handler runs in the CPP's context there should be little in the way of penalty to the CPP.

   (b) Portions of the text segment can be protected in a like manner, allowing CuPIDS to detect entry into critical regions. By forcing the CPP to enter and exit critical areas through a call gate [54], the CSP can receive both entry and exit events. Further, by delaying the clearing of the fault the CSP can ensure that any background work it needs to perform can take place before letting the CPP proceed.

This includes write protection faults for CPP, CSP, or kernel text segments. This mechanism allows the CSP to detect attempts by a process to modify its own code; this is true even in the case where some mechanism such as a buffer overflow injects into the system code that sets up signal handlers to catch this kind of fault. This ability of CuPIDS to validate system events that could lead to compromise is one of its key strengths. In this capacity it is acting as a reference validation mechanism [39].

CuPIDS has the capability of routing interrupts to the CuPIDS CPU as well as intercepting signals destined for the CPP and routing them to another process (such as the CSP).

3.6 Basic Monitoring Capabilities

This section describes a minimal set of capabilities required for detection of illegitimate behavior in a CuPIDS system.

3.6.1 Whitelists

In line with our focus on specification-based ID, the detectors in the CuPIDS architecture are defined around sets of information about what activities a program is allowed and not allowed to perform, and what resources it is and is not allowed to access. These sets of information are typically referred to as white and blacklists [55].

Whitelist Creation

Currently the whitelists for a CPP are created by parsing the program's binary file and extracting function, library, and system call source/destination pairs and storing them in files. This process is not precise; there are complications such as function pointers and occasional calls by non-CSP protected helper programs and libraries. Additionally, self-modifying code may require special handling for dynamic code segments. Currently CuPIDS has the ability to augment the whitelists with

call pair events that are not in the whitelist but that occur during training sessions. It may be possible to construct a complete whitelist directly from the binary without requiring training runs; however, it is simpler to handle it this way for our initial prototype.

These files are parsed by the CSP during initialization and used to construct data structures that are used during the CPP's operation to validate its activity. While the current whitelist data is merely text, it is possible to imagine easy and effective means of cryptographically protecting it from surreptitious modification.

Whitelist Validation

CuPIDS currently uses the function call entry and exit events to perform whitelist validation of control flow within the CPP. Each event contains a source and destination address pair for each function that is called, and these pairs are compared by the CSP to allowed pairs. The same is true for library calls, intra-library calls, and system calls. Any malfunction in the CPP process that leads to code being run in a pattern that is not in the whitelist will be caught by the CSP. In the case where the code makes a call to a function it is not allowed to call, the entry event for the illegitimate function invocation will be caught by the CSP. If the call into the function bypasses the prologue code and an entry event is not generated, the exit event may be caught by the CSP.

3.6.2 Stack Monitoring

CuPIDS uses function entry and exit events to model the CPP's program stack. Events that break the stack model legitimately can be added to the whitelist, either manually or automatically during training sessions. Examples of exceptional but legitimate events may include setjmp/longjmp pairs and exception or signal handlers.

One aspect of the CPP stack that may need to be modeled is how long each stack frame exists. If the top of the stack does not change for a long time, where "long"

needs to be determined, then it is likely that something has gone awry. Because all internal function calls and external environmental interactions are monitored and compared against the whitelists the application must be stuck in the current function. The most likely culprit is some sort of endless loop caused by a denial of service attack or a program fault.

3.7 Protective Activities

Towards validating our thesis we defined four main activities that could benefit from parallelization or were critical to the architecture. These activities involve application startup and shutdown, internal control flow monitoring, environmental monitoring, and finally, focused testing.

3.7.1 Application Startup

In application startup the modified O/S first cryptographically verifies the CSP and CPP. It then loads the CSP into memory and while the CSP is initializing itself the CPP is loaded, but not started. The CSP establishes any needed connections into the CPPs VM space and other needed initialization, then starts the CPP running. Once the CPP is started, whenever it is chosen by the system scheduler to execute the appropriate CSP is also ensured to be running on a parallel CPU. This satisfies the CuPIDS architectural requirement that when a CuPIDS production process is executing (actually running on a production CPU), its associated protective process is executing on a CuPIDS CPU.

The process by which a CuPIDS protected process is loaded is illustrated in Figure 3.8. Here the operating system is instructed to execute a protected program. It first validates the integrity of both the production and protective programs using a pre-computed cryptographic signature or some other mechanism. If both programs are valid, the O/S first loads the security process into memory, then the production process, and starts the security process executing on a security CPU. The security

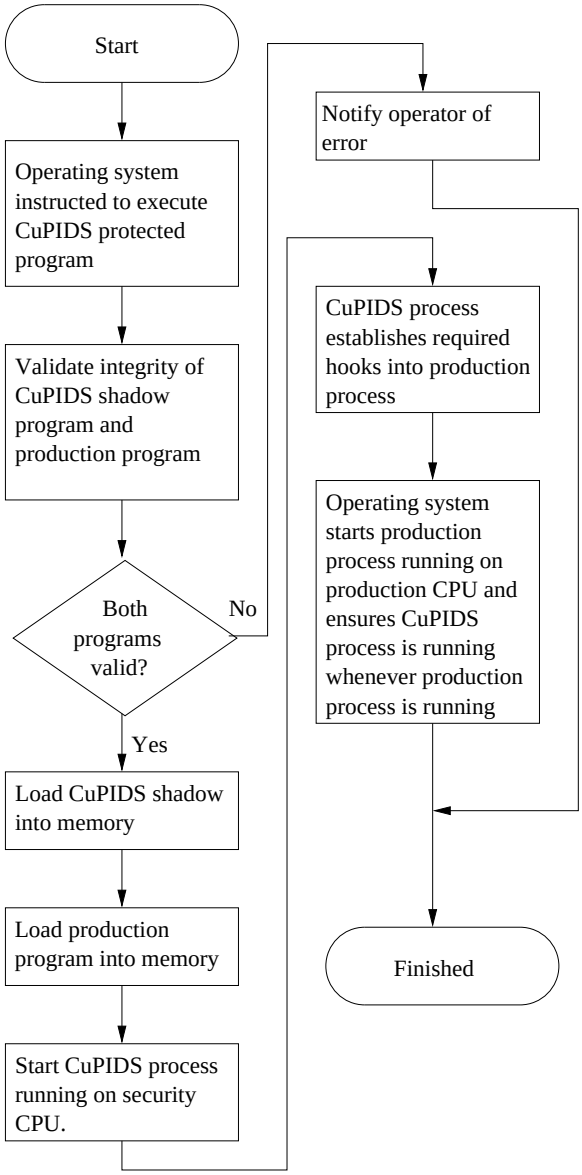

Figure 3.8. Protected Process Loading Flowchart

process establishes any connections it needs into the production process' memory space and operating environment. When the security process is ready the O/S starts the production process running on a production CPU. As the production process is switched onto and off of the production CPUs the operating system ensures the

protective security process is always running whenever the production process is running.

The CPP and runtime libraries have been instrumented to send an event message to the CSP for every function entry and exit. This evaluates to a large number of events, and a correspondingly large amount of inter-process communication, so we have defined and created an efficient communications mechanism to keep the runtime cost of this activity down. The environmental monitoring is concerned with the CPPs interactions with the operating system and runtime libraries. And finally, the focused invariant testing is done by specifically crafted monitoring modules. Examples include the type of invariant testing that is usually done during the testing phase of the software lifecycle, but that is typically removed from shipping products because of its high runtime cost.

In addition to these protective activities, which are focused upon detecting illegitimate behavior, the CuPIDS architecture defines response mechanisms. These response mechanisms not only include alerts and damage containment via stopping an errant process, but may in some cases repair corruption before the process is irreparably damaged. This capability stems from CuPIDS quick detection and some of the virtual memory-based background capabilities. The ability to efficiently take snapshots of a process memory during execution offers novel forensics opportunities.

Shutdown tasks include verifying that the CPP shutdown path followed a legitimate code path. Additionally, any runtime history data is saved to disk.

3.7.2 State Monitoring and Assertion Verification

The creation of appropriate interfaces into the kernel allows the CSP to monitor nearly all aspects of the CPP's operating environment and state. This visibility includes the CPP's entire VM space and any related kernel data structures, and excludes only the internal processor state while the CPP is on a CPU. One use of this capability is invariant testing. Invariant testing is a two stage process involving pre-

compilation work and runtime invariant checking. The pre-compilation task involves determining which variables need monitoring, defining invariants for those variables and exporting that information in a form that can be used by the CSP. The compiler is also used to automatically instrument the CPP by adding event generators into each function prologue and epilogue. Invariants are currently snippets of code that could be directly included in the CPP's code (similar to the run-time debugging tests discussed earlier). They are compiled into the CSP's code, and when one is used, it is given appropriate pointers into the CPP's virtual memory space and executed. Currently these are manually written; however, work is underway to allow a programmer to indicate, via pragmas, to the compiler that a particular variable needs protection and the compiler will automatically generate the invariant testing code in the CSP.

3.7.3 Runtime Execution Monitoring

Runtime monitoring includes a number of activities and capabilities that give the CSP visibility into the operation of the CPP. An example includes generating events so the CSP is made aware of the creation, accesses to, and deletion of a protected variable's lifespan. Other events export an execution trace to CuPIDS via function call monitoring, and interactions between the CPP and external environmental entities such as calls to runtime libraries and the operating system. Call monitoring consists of the CPP sending a stream of function/library/system call entry and exit events to the CSP. The CSP then uses a model based upon how the CPP is supposed to operate to verify if that stream is legitimate.

3.7.4 Non-interactive Monitoring

The initial model requires explicit CSP notification of protected variable life cycle and access events via CPP-based event generators. If this proves too expensive for some cases it may be desirable to designate some number of separate classes

of variables and require the compiler to place each class of variables into its own memory segments. The pages holding these segments can be marked using the hardware read/write protection structures. By trapping attempts to access these memory segments the CSP can be made aware of CPP's use of variable classes without requiring explicit CPP involvement or associated costs of message sending. There may be some CPP performance penalty depending on how quickly the fault can be handled.

3.8 Self-healing/Self-protection

CuPIDS' ability to perform real-time detection offers novel capabilities in system protection and self-healing. To leverage these capabilities the architecture must itself be protected. One way in which this is accomplished is through protection from attacks against the integrity of the IDS and its data inputs. Section 3.8.1 describes how the architecture is hardened against such attacks. Section 3.8.2 discusses how the architecture can be used to repair damage caused by a successful attack against a protected component. Finally, Section 3.8.3 introduces how the parallel and protected nature of the architecture allows for the gathering and preservation of forensic data.

3.8.1 Spoofing Protection

Attacks against the IDS are a threat we address with the CuPIDS architecture. The most commonly described threat seems to be attacks that attempt to bypass the IDS rather than shut it down directly. Wagner and Soto [56] describe the difficulties faced by common IDS architectures in detecting and defeating such attacks and conclude that constructing a system that is not vulnerable to such attacks is difficult. They present a systematic, theoretical, language-based framework for creating attacks based upon knowledge of how the IDS operates and use this framework to

support their thesis. Their concern largely depends upon an IDS trying to determine if a given set of environmental interactions such as system calls is valid [57, 58].

We address this threat in two ways: specification-based detection, and spoofing protection. The CSP for a given CPP has a whitelist of system resources that may be used by that process. This list is augmented with the locations in the CPP's text segment from which an invocation of a particular internal function, runtime library or system call can be made. Thus, for attackers to successfully carry out a mimicry attack against the IDS they must not only use resources that are on the whitelists, but they must make the calls from the original locations in the program text. By protecting the text segment from modification (via interrupt routing and monitoring the page fault errors), and guarding against spoofing (via event generation using hardware-based kernel capabilities) CuPIDS remains capable of detecting this type of attack.

To protect against an attacker spoofing the interrupt-based IPC mechanism, handlers capture the return address of the caller from the trap frame. Thus, a spoofed call coming from an illegitimate location will be caught.

By including calling location and preceding branch locations with system calls we gain context not used by the other models. Simple use of this information allows the IDS to use $O(1)$ tests to ensure that the system call came from the text segment of the program. An example of why making use full use of this context in an inline procedure (as on a uniprocessor system) is not feasible might be correlating system calls with legitimate calling addresses or analysis of system call input parameters. In this case, the CuPIDS can work on this type of correlation in parallel with the system call execution.

Finally, in an effort to make the attacker's task more difficult, CuPIDS can take random snapshots of the CPP's instruction pointer (and perhaps other process state) and verify that it is legitimate in terms of its location in the text segment and given the current model status. This will make successful mimicry/spoofing types of attacks more difficult.

3.8.2 Self-healing

There are a number of well-known-to-be-dangerous library and system calls [59]. Among the most common exploits publicly available are buffer overflows that use unsafe string handling library functions to overflow vulnerable buffers[2]. Using a combination of stack modeling, library call event monitoring and the virtual memory mapping capability it is possible for CuPIDS to automatically detect and generate detection signatures for certain common classes of vulnerabilities such as stack-based overflows. In many cases buffer overflows use known library function such as strcopy(3) [60]. When CuPIDS is notified of a call to strcopy it can create a copy on write (COW) mapping of the page(s) containing the buffer and surrounding memory region. If information about buffer sizes is available to the CSP, either automatically generated or inserted by the programmer in the form of CuPIDS memory operation events it becomes possible for CuPIDS to not only detect and generate signatures for anomalous events, but also to recover from them automatically. It does so by using the saved copy of stack (or heap) pages to recreate the process' memory state as it was before the overflow, and copying only the correct amount of data into the buffer from the corrupted pages. While in the case of an exploit attempt the data ending up in the buffer may not be what the CPP programmer intended, the overall effect to the program is the same as if a safe string copy function such as strncopy(3) [60] had been used. In addition, error variables or signals may be set to indicate that something unexpected occurred.

A further possible benefit of the memory mapping capability is the ability to automatically generate intrusion/misuse detection signatures based upon the changes to memory during illegitimate CPP operation. In particular, the contents of a successfully overflowed buffer during input from the network can be used to generate a signature for network firewalls and IDSs.

[2]See [59] and [43] for a detailed discussion of buffer overflows.

3.8.3 Forensics

When CuPIDS detects problems in a CPP it can freeze that process, write its entire state out to disk, and start up a new instance of the program. The saved state can include the normal core dump, the kernel structures related to the process, the state of any files in use by the process at the time of error, and the runtime history stored in the CSP. This data will allow for complete analysis of the fault.

3.9 Strengths and Weaknesses of the CuPIDS Architecture

3.9.1 Strengths

The primary advantage of parallel architectures such as CuPIDS lies in the speed of detection in those cases that support parallel monitoring. Other advantages include the types of activities the architecture can observe as well as gains in the amount and type of security monitoring work that can be done.

Timeliness of Detection

Parallel monitoring allows CuPIDS to detect vulnerability exploitations or errors while they are occurring. In some cases, this allows CuPIDS to respond before events that depend upon the exploitation or error can occur (real-time detection [10]). An example is a stack-based buffer overflow detect. Because CuPIDS is able to detect the write past the end of the buffer as it occurs, it is able to stop the CPP before the currently executing function returns into the injected code.

Types of Activity Observed

Unlike a StUPIDS, CuPIDS is not limited to pre and postcondition invariant testing. Rather, it is able to observe changes to data structures as they occur. A particular advantage is the ability to monitor activity occurring in a process' criti-

cal sections. An additional strength is the high fidelity monitoring specified by the architecture, which includes low level activities such as internal flow control events, external environmental interactions, and programmer defined events protecting vulnerable or critical regions.

Work Distribution

A further advantage of the parallel architecture is the off-loading of non-production security or error checking tasks from the production application. This allows for detector tasks that may be too expensive to perform inline. CuPIDS emphasis on focused parallel operation further extends this advantage. Most StUPIDS architectures attempt to perform generalized detection tasks, constantly protecting many parts of the production system simultaneously. Conceptually, a general purpose IDS erects a barrier over a large area behind which multiple production components are protected. An attacker, interested in targeting a specific vulnerability in an application behind that barrier need only find a small chink in the armor through which an attack can slip. We believe that as systems get more complex, generalized defenses get spread thinner and such chinks are difficult to eliminate. There are at least two possibilities for reducing the protected area. The first is that taken with the Poly^2 architecture [61]. Here instead of running multiple services on a general purpose operating system, the computing architecture runs single applications on operating systems stripped to the bare minimum needed to support that application. This dramatically reduces the attack surface [62], and the associated defensive surface. CuPIDS operates in a related but orthogonal way, constructing smaller, highly focused shields around particular system components. The two architectures are complementary: a simple, well defined Poly^2 system makes the generation of CSP detector sets simpler, and CuPIDS can be used to protect those components and process which remain in the stripped Poly^2 system.

3.9.2 Drawbacks

An obvious cost of the CuPIDS architecture is the loss of a CPU to perform production work. Another weakness of the architecture stems from the communication flows necessitated by the CuPIDS application division into production and security components. Unlike some of the related virtual machine or separate hardware-based architectures discussed in Chapter 2, CuPIDS is part of the host system and is therefore vulnerable if that host is compromised. Finally, the application-based protection afforded by CuPIDS requires careful analysis to protect the system.

Vulnerable Communications

CuPIDS encodes certain activities in the CPP into event messages and sends them to the CSP. This requires that the developer of the CPP be aware of the CSP and necessitates that the CPP participates in its own defense. This allows attacks against the IDS through a compromised application corrupting the flow of events to the CSP. To handle this problem we require event messages to pass through a kernel layer that is used to validate their authenticity. This strategy is dependant upon the kernel not being compromised, and the current CuPIDS architecture does not currently offer any guarantees in this regard.

Embedded versus External Design

We chose to embed CuPIDS inside the host operating system instead of outside it with a separate operating system or inside a virtual machine monitor [44]. This gives us greater visibility into the operations of the application and operating system; however, if the operating system is compromised then CuPIDS is vulnerable.

Risks Related to CuPIDS Highly-focused Detection

Finally, as discussed in Section 3.9.1, the primary benefits of CuPIDS require focusing on individual applications. This requires careful analysis of the overall system to determine which applications must be protected as discussed in Chapter 25 of [13].

4 THE CUPIDS IMPLEMENTATION

This chapter describes the implementation of a prototype intrusion detection system based on the CuPIDS architecture. This prototype focuses on parallel monitoring and provides the primary testing and analysis platform for this dissertation.

4.1 Purpose of the Implementation

The hypothesis that underlies this dissertation (Section 1.5) is practical in nature. Our primary goals for the prototype were twofold—demonstrating both the validity and practicality of our research thesis. The prototype was used to verify that parallel monitoring using the CuPIDS model enables quicker detection than is possible with a comparable StUPIDS. It also showed that it is feasible to build an intrusion detection system using the CuPIDS architecture. Therefore, an implementation was critical for the development of this dissertation. It was used both for practical verification of the intended features of the architecture and to aid in reasoning about and experimenting with its characteristics.

We have implemented a prototype CuPIDS. This section briefly describes the current state of that prototype. Our experimentation uses FreeBSD, currently 5.3-RELEASE [63]. We have added to the operating system API a set of CuPIDS-specific system calls that give CuPIDS processes visibility into and control over the execution of a CPP. Examples of the new functionality include the ability to map an arbitrary portion of the CPP's address space into the address space of a CSP and a means by which signals destined for and some interrupts caused by the CPP are routed to the monitoring CSP. The operating system kernel has been modified to perform the simultaneous task switching of CPPs and CSPs, a CSP protected loading capability as discussed above in section 3.7, and connections into various

kernel data structures have been added to allow the CSP better visibility into CPP operation and for runtime history data gathering.

Our experimentation to date has focused on protecting specific applications[1]. We perform interactive monitoring based upon automatically generated instrumentation from the compiler as well as CPP programmer-defined invariants for key variables. CuPIDS has the capability to examine program binaries and extract explicit white-lists about which system resources are used by the CPP, and then save this information in a form usable by the CSP. As the CPP runs it sends messages to the CSP notifying it about operational activities such as protected variable lifetime events (creation, accesses and deletion) as well as control flow events (currently all function call entry and exits, to include library and system call invocations) are passed to the CSP as well. The CSP receives these messages and uses them to ensure the CPP is operating correctly. In the case of variables the CSP performs pre- and post-condition invariant checking, and in the case of flow control, it verifies that all function calls are to and from legitimate locations within the CPP text segment. It also maintains a model of the CPP call stack and verifies all function returns are to the correct locations.

4.2 Implementation Platform

Our prototype has been implemented in FreeBSD [63]. This version of the Unix operating system was chosen for the following reasons:

- FreeBSD is open-source. The source code is available, which makes it possible to understand and modify the system to meet our needs.
- Extensive documentation is available [53, 64–67] about the internals of the kernel.

[1]The techniques involved are largely applicable to operating system protection as well

- Significant effort has been made to build mandatory access controls (MAC) into the operating system [66, 68]. While the current research does not make use of these mechanisms, future research will explore how parallel architectures may be used to significantly enhance operating system robustness, even in the face of successful, root-level attacks.

- FreeBSD is representative of the operating systems used in our target environment—an organization's critical server infrastructure. Furthermore, it is capable of running most UNIX and BSD applications.

The CuPIDS implementation described in this document was constructed on FreeBSD 5.3-RELEASE. System development and experimentation were performed using computers with Intel or Intel compatible processors.

4.3 CuPIDS Kernel API and Data Structures

4.3.1 Kernel API

The additions to the system API are placed in loadable kernel modules wherever possible. This is done for two reasons: development, and to allow dynamic changes. Placing new system calls in a loadable kernel module allows changes to those interfaces without compiling the entire kernel and rebooting the machine. This saves a great deal of time while debugging, and also satisfies the desirable characteristic of dynamic change (see Section 1.2.1).

The following system calls were added to the FreeBSD API:

- `process_bind`: Based upon input parameters this system call binds either an entire process, including its full contingent of threads to a processor, or it can bind specified threads to a processor. The system scheduler has been modified to respect these bindings.

- `process_unbind`: Based upon input parameters this system call unbinds either an entire process, including its full contingent of threads from a processor, or it

will unbind a specified thread from a processor. Unbound processes are freely scheduled by the system scheduler.

- `c_init_cpp`: This system call initializes the system to support a CuPIDS CPP and associated CSP. It first allocates a CPU for use exclusively by CuPIDS. It then initializes the CuPIDS IPC data structures and sets up the CPP to use the initialized structures. Finally, it can be used to route signals destined for the CPP to the CSP for handling.

- `c_destroy_cpp`: This system call frees up system resources used by a CuPIDS CPP and associated CSP. It deallocates the CuPIDS IPC data stuctures used by the pair, and if appropriate returns the CuPIDS CPU to the system.

- `c_mmap`: This system call takes CPP and CSP process IDs and a memory address and length then maps the memory associated with that address in the CPP into the virtual memory space of the CSP. The location in the CSP's memory space is returned to the caller. An additional parameter determines what type of mapping is made.

  The two types of mapping currently used are direct and copy on write (COW). In direct mapping the pages containing the desired memory are mapped directly into the CSP address space. Changes to that memory by the CPP are seen immediately by the CSP, and changes to the the memory by the CSP are immediately seen by the CPP.

  COW mappings are used to assist analysis and response or recovery activities. A COW request maps the selected pages into the CSP's memory space the same as the non-COW mapping; however, changes to the memory by the CPP are not seen by the CSP. Memory mapped with this option is typically used in conjunction with a direct mapping, allowing CuPIDS to see the differences in memory caused by CPP activity.

- `c_unmmap`: This system call takes the address and length of a memory location mapped using the `c_mmap` system call and removes it from the calling process' memory map.

- `c_msgrcv`: This system call behaves similarly to the existing msgrcv(3) [69] system call, but uses the CuPIDS IPC mechanism.

4.3.2 CuPIDS IPC

We investigated how best to move information in the system, and selected two communications mechanisms to explore. The first mechanism is based upon the system call-based SysV message interface. Using this interface a CPP creates a message and sends it to the CSP using msgsnd(3) [70]. The CSP receives the message using msgrcv(3) [69] and processes it. This mechanism works well for low throughput traffic types of activities such as focused invariant monitoring; however, it does not perform well when large numbers of events need to be communicated between processes. To facilitate high throughput needs such as that imposed by instrumenting all function, library, and system calls we developed a high-efficiency interrupt-based communications mechanism. We observed that much of the cost in the SysV interface comes from the system call interface. A great deal of work is done after the kernel is invoked simply to validate the system call and route it appropriately. The message interface itself is also quite complex and that complexity adds more time costs to each invocation of msgsnd or msgrcv. The combined effect of these result in the execution of several thousand machine instructions for each system call. We designed a software interrupt-based mechanism for moving events between CPP and a kernel buffer that only requires about 50 machine instructions, thus eliminating much of the overhead imposed upon the CPP by the SysV IPC. Section 5.2 describes the performance gains realized by this mechanism.

The non-optimized mechanism built into the CuPIDS prototype was designed to be simple and fast. The user-mode invoking mechanism (illustrated in Figure 4.3)

is only three machine instructions. The kernel-mode portion of the interface is only slightly more complicated as seen in Figure 4.1. This CuPIDS IPC-specific code consists of 19 machine instructions yet replaces the normal system call handler which, because its generality, must execute many thousands of instructions to achieve the same goals.

```
/*
 * Handle CuPIDS function enter interrupt (int 0x81)
 */
    SUPERALIGN_TEXT
IDTVEC(cupidsfuncenter)
    PUSH_FRAME
    movl    $KDSEL, %eax
    movl    %eax, %ds       /* use KERNEL data segment */
    movl    %eax, %es
    movl    $KPSEL, %eax
    movl    %eax, %fs

    call    CuPIDS_func_enter

    movl    lapic, %eax
    movl    $0, LA_EOI(%eax)    /* End Of Interrupt to APIC */
    POP_FRAME
    iret
```

Figure 4.1. Kernel System Call Replacement Code

We should note that we did not use a shared memory message passing interface because of concerns that a compromised CPP could easily defeat the CSP using

spoofed communications. By forcing the communications to go through the kernel we gain the ability to verify the authenticity of events[2].

4.3.3 Kernel Background Capabilities

Interrupt Routing

The current implementation has the capability of routing hardware and software interrupts to the CuPIDS CPU or a specific CuPIDS interrupt handler thread. This capability is primarily intended to support self-protective activities such as allowing CuPIDS to validate the legitimacy of writes to particular memory regions (e.g. CuPIDS' text segment or the MAC logic regions.). This capability is not used in the experimental results described in this document.

System Scheduling

The system scheduler was modified to support automatic CSP/CPP switching. When the CPP is chosen to be placed on a CPU, the associated CSP is also chosen. This ensures the CSP is always running while the CPP is running. If the CSP has work to do it should not be switched out by any process, but if it is idle, it may be okay to allow another CuPIDS process to run until an event requiring the CSP's attention occurs. An example may be that there are no messages in the message queue so the CSP yields to a CuPIDS thread that is sweeping through memory looking for bugs.

Currently one CPU is designated and reserved for CuPIDS. This simplified the scheduling logic; however, it may not matter which CPU the CPP and CSP are on as long as the CSP is always running alongside the CPP when the CPP is on a CPU.

[2]While this does assume the kernel has not been compromised we can use CuPIDS-like techniques to protect the kernel as well

Application Startup/Shutdown

Startup tasks include automatic CSP loading and initialization upon CPP load. The CSP and CPPs are separate executables. The O/S automatically detects the CPP's being loaded and before executing the CPP the CPP and CSP are cryptographically validated as being unmodified, the CSP is loaded and allowed to initialize, and finally the CPP is allowed to execute. This is achieved in the prototype implementation by starting the CSP as a separate application that initializes itself and the system, then forks the CPP. The forked process waits upon a system semaphore until the CSP has established its required connections and then `execs` the CPP's code. The initialization, operational, and shutdown tasks include:

1. Upon startup the CSP opens the receive end of the message queue and sets up some basic parameters such as the CPP process identifier and other initialization.

2. Internally, the CSP spawns message processing and command accepting threads. The message processing threads reads messages from the various message queues, does some record keeping and dispatches the events embedded in the messages to the appropriate handlers. The command thread takes input from the user and forwards those commands to the appropriate handlers.

3. Shutdown tasks include verifying that the CPP shutdown path followed a legitimate code path. Additionally, if the CSP uses runtime history data any stored runtime information is saved to disk.

4.3.4 Data Structures

Most active CuPIDS functionality resides in the CSP. Therefore most event data is not needed in the kernel. The only CuPIDS kernel data structure is a per-CSP/CPP pair event queue. This queue and supporting synchronization mechanisms are stored in the CSP's process table entry.

4.4 CuPIDS Production Process

In general little work is required to create a CPP. The program is linked with CuPIDS-augmented and instrumented versions of the runtime libraries and message queue initialization code are added to the program's startup code. The compiler is used to automatically generate program flow events. Manual event generation for variable protection is handled by adding library calls as described in Section 3.4.3. Little work is required to instrument the CPP for variable monitoring. An example using SysV IPC is illustrated in Figure 4.2.

```
buf.mtype = CUPIDS_MSG_VAR_ACCESS_BEGIN;
buf.var_number = 1;
msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0);
strncat(wbuf, resolved, MAXPATHLEN);
buf.mtype = CUPIDS_MSG_VAR_ACCESS_END;
buf.var_number = 1;
msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0);
```

Figure 4.2. CPP Variable Access Notification Code Sample

4.5 CuPIDS Shadow Process

Creating a CSP is also straightforward. The majority of the interfacing with the CPP and CuPIDS support mechanisms are handled automatically by linking the CSP program with a CuPIDS runtime library. The focused runtime monitors described in Section 3.4.3 are created by writing runtime invariant tests into the CPP and then extracting that code out and into the CSP. The runtime tests use pointers to access protected data structures. When a protected variable is created the memory it occupies is mapped into the CSP's VM space. The appropriate invariant test pointers are set to point to the correct location in the CSP's memory.

When a variable access event is received by the CSP the appropriate runtime test is executed. In the parallel monitoring case the monitor is bound to the CuPIDS CPU for execution.

4.6 Compiler Support

We used the Gnu compiler collection (GCC) to build both the CPP and CSP as well as the kernel. The compiler is used to automatically instrument the CPP by adding event generators into each function call and return. Rather than require that each application include the support functions required by the compiler, we modified the C-runtime library to include the needed stubs. These stubs were simple to create, and add little overhead when run; see Figures 4.3 and 4.4 for the CuPIDS IPC versions and Figures 4.5 and 4.6 for the SysV IPC versions.

```
inline void __cyg_profile_func_enter(void *this_fn, void call_site){
__asm__("mov %[this_fn], %%edx"::[this_fn] "r"(this_fn):"edx");
__asm__("mov %[call_site], %%ecx"::[call_site] "r"(call_site):"ecx");
__asm__("int $0x81");
}
```

Figure 4.3. CuPIDS IPC Function Prologue Code

```
inline void __cyg_profile_func_exit(void *this_fn, void *call_site){
__asm__("mov %[this_fn], %%edx"::[this_fn] "r"(this_fn):"edx");
__asm__("mov %[call_site], %%ecx"::[call_site] "r"(call_site):"ecx");
__asm__("int $0x82");
}
```

Figure 4.4. CuPIDS IPC Function Epilogue Code

```
inline void __cyg_profile_func_enter(void *this_fn, void
*call_site){
    if(msqid == 0)
    return;
    buf.mtype = CUPIDS_MSG_FUNC_ENTER;
    buf.this_fn = this_fn;
    buf.call_site = call_site;
    int error = msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0);

    if (error != 0){
    perror("\nCuPIDS Library: func_enter msgsnd error1!");
    printf("\nCuPIDS Library: func_enter error code: %d", error);
    }
}
```

Figure 4.5. SysV IPC Function Prologue Code

4.7 Reporting Mechanism

The focus of this implementation was not on reporting, so a simple notification mechanism was used. Reports of bad events were printed to the system console and logged to a local file.

4.8 Test Platform

The experiments described below were run on a MP platform with dual Xeon 2.2GHz processors, 1G RAM, one 120GB ATA100 drive. Hyperthreading (HTT) was enabled so the operating system had available 4 CPUs. We recognize that the performance of HTT processors does not match that of separate CPUs [71]; however, the architecture is useful to us for other reasons. While the results discussed here

```
inline void __cyg_profile_func_exit(void *this_fn, void *call_site){
    if(msqid == 0)
    return;
    buf.mtype = CUPIDS_MSG_FUNC_EXIT;
    buf.this_fn = this_fn;
    buf.call_site = call_site;
    int error = msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0);
    if (error != 0){
    perror("\nCuPIDS Library: func_exit msgsnd error1!");
    printf("\nCuPIDS Library: func_exit error code: %d", error);
    }
}
```

Figure 4.6. SysV IPC Function Epilogue Code

do not make use of HTT specific features we use do make use of the fact that they share architectural components in research discussed in Section 1.2.1.

4.9 Conclusion

In this section we discussed the implementation of a prototype CuPIDS intrusion detection system based upon the CuPIDS parallel architecture. While the kernel modifications required to support the CuPIDS architecture were extensive, the work required to construct a CPP, as illustrated in Section 4.4 is minimal. This supports our theory that the CuPIDS architecture has reasonable construction costs.

5 TESTING THE CUPIDS IMPLEMENTATION

Upon completing the initial CuPIDS implementation, a series of tests were performed to explore its behavior. The experiments were designed to determine if the CuPIDS architecture, as embodied in the prototype, supported or refuted our research hypothesis.

The experiments described here demonstrate that it is possible for one process to efficiently perform realtime runtime error checking on variables in another process as well as perform simple flow control validation. To demonstrate the validity of our research hypothesis we demonstrate that CuPIDS can provide guaranteed detection of certain attacks before a context switching event occurs. This claim cannot be matched by a StUPIDS, even if equipped with a comparable detector set.

5.1 Test Design and Methodology

5.1.1 Test Applications

In our experimentation we used a combination of widely-used, open source applications and servers as well as applications created specifically to test certain aspects of CuPIDS' functionality. The commonly used applications were WU-FTP version 2.6.2 and gnats version 3.113.12. These programs were chosen because they represent software typical of that used in our target environment, their source code is available so that we could examine and instrument them, and because they contain exploitable vulnerabilities as demonstrated by publicly available zero-day exploits.

WU-FTP-2.6.2

WU-FTP's ftpd daemon was used to perform performance measurements of CuPIDS as well as to test CuPIDS' invariant violation detection and self-healing capabilities. The ftpd daemon was ideal for this purpose because it is fairly large (about 20,000 lines of code), its behavior is representative of many server-type applications in that it runs for long periods and forks off child processes to handle requests, and finally because it has a number of buffer overflow vulnerabilities[1].

gnats-3.113.12

We used gnats-3.113.12 because of the existence of a locally exploitable vulnerability[2]. gnats was used to test CuPIDS' ability to detect invariant violations.

CuPIDS Operating System

As discussed in Chapter 4 we modified FreeBSD 5.3 to support CuPIDS' requirements.

5.1.2 Test Platform

The test platform described in Section 4.8 was used to perform the experimentation described in this chapter. For experiments involving CuPIDS, the CSP is the only user of CPU1, the instrumented `ftpd` uses all of one CPU's cycles, and the ftp client uses all of another CPU's cycles, and the system, including the test drivers, mostly run on the fourth CPU. The test drivers ensure that all file I/O is done on local drives so that network overhead does not become a factor. During the non-instrumented experiments CPU1 is held idle to provide `ftpd` the same operating environment as it had in the instrumented runs. `ftpd` was run as root in

[1]CVE entries CVE-1999-0878, CAN-2003-0466, and CVE-1999-0368 [72]
[2]CVE CAN-2004-0623 [72]

Table 5.1
WU-FTP Runtime Performance Measurements (50 samples)

Event Comm. Method		Clock Time (sec)	User Time (sec)	Sys Time (sec)	Throughput (MB/sec)
Interrupt-based	mean	139.42	0.44	1.27	14.53
	stdev	1.43	0.05	0.09	0.17
	stderr	0.20	0.01	0.01	0.02
	min	133.55	0.34	1.08	13.87
	max	141.44	0.53	1.42	14.99
SysV IPC-based	mean	166.67	0.41	1.62	15.61
	stdev	0.37	0.05	0.07	0.30
	stderr	0.05	0.01	0.01	0.04
	min	166.04	0.28	1.51	15.32
	max	168.12	0.52	1.84	16.08
Non-Instrumented	mean	117.74	0.41	1.34	15.94
	stdev	0.24	0.04	0.08	0.07
	stderr	0.03	0.01	0.01	0.01
	min	117.47	0.29	1.16	15.76
	max	119.13	0.50	1.53	16.04

stand-alone mode (command line `ftpd -s` that causes it to stay in the foreground and fork processes as needed).

5.2 Runtime Efficiency Tests using WU-FTP

The initial experiments connect to the ftp daemon, log in, change local and remote directories, and perform 300 ftp file transfers and one `ls` for a total of 301 transfers. The file transfer workload is 1,881,832,400 bytes and the overall workload per experiment is 1,881,904,317 bytes. Three sets of 50 experimental runs were

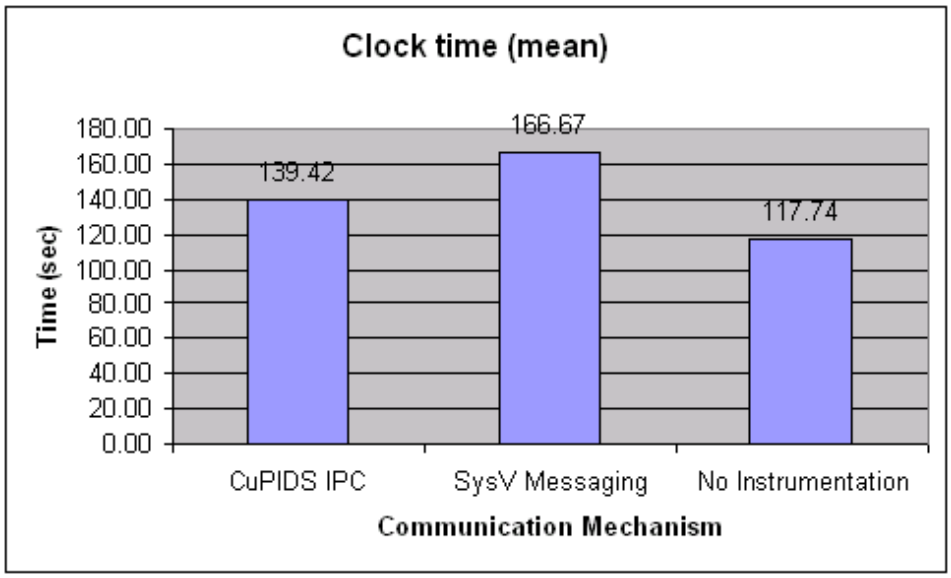

Figure 5.1. Runtime Summary (50 Samples)

made, one using the CuPIDS interrupt-based IPC, one using SysV IPC, and one baseline test was run against an non-instrumented version of WU-FTP. The results are summarized in Table 5.1.

The initial tests are intended to measure the overhead involved in getting CuPIDS events out of the CPP and into the CSP, therefore we constructed a worst-case event load based on program flow control monitoring. In the instrumented tests, all function calls generate entry and exit events. This includes internal functions, libc and intra-libc calls as well as system calls. Each event includes caller address and callee address information. These events are validated against a whitelist of calls statically extracted from the ftpd binary. The initial whitelist contained all the legitimate non-function-pointer-based function and shared library calls as well as a list of all function pointer uses. An initial experimental run identical to the timing runs was made to train the CSP on the actual function pointer usage. The CSP received each function/library/system call event, verified it against the whitelist,

and used it to model the CSP's program stack. The timing related tests did not include embedded invariant tests.

Each experimental run took between two and four minutes and generated approximately 1.4 million events corresponding to WU-FTP's activities. As shown in Table 5.1 the overhead of generating and using those events was around fifteen percent for the CuPIDS IPC as opposed to approximately 100 percent for the SysV-based IPC. Note that this overhead should be balanced against the removal of an inline IDS doing the same tasks. Even a standalone IDS with a similar detector set would be competing for CPU cycles with the CPP, likely degrading application performance.

5.3 Control Flow Change Results

A number of experiments were run to validate CuPIDS' ability to detect illegitimate control flows in the CPP.

5.3.1 Illegitimate System Call Invocation Detection

Both gnats and WU-FTP were used in these tests. In both applications a buffer was overflowed in such a way that bytecode contained in the overflow string was executed. The injected code made a number of system calls from the stack. CuPIDS was able to detect all of the illegitimate system calls invocations.

5.3.2 Illegitimate Internal Function Call Invocation Detection

Both gnats and WU-FTP were used in these tests. In both applications CuPIDS was able to detect an internal function call that had been removed from the whitelist (simulating the activity of injected code that makes calls to functionality embedded in the vulnerable application). CuPIDS was also able to detect calls into functions that

bypassed the prologue event generator. It did so by detecting illegitimate program stack activity in the stack model.

5.3.3 Illegitimate Library Call Invocation Detection

Both gnats and WU-FTP were used in these tests. In both applications CuPIDS was able to catch a call to a library function that was removed from the whitelist.

5.3.4 Spoofing/Masquerading Detection

CuPIDS detected attempts to make library or system calls from locations other than those specified in the whitelists. This prevents attackers from performing masquerading attacks such as those described in [73]. The CuPIDS IPC mechanism guards against spoofed event generation by including in each event the return address for the generating function as taken from the stack. As the address is placed on the stack by the processor and reading it occurs in kernel space there is no way for a user program to spoof this information.

5.3.5 Direct Variable Protection

WU-FTP was used for these experiments, which involved performing invariant testing on simple variables (int, char, simple structs) and a string buffer. As discussed earlier, CuPIDS was able to detect illegitimate changes to both classes of variables. In the case of a stack-based buffer overflow it was able to detect the overflow, save the overflowing data, repair the corruption to memory following the buffer, terminate the string in the buffer appropriately (by writing a zero into the end of the buffer), allow the CPP to continue running, and write the overflow string and information about the overflow out to disk. In these experiments the detection took place as the overflow occurred, so CuPIDS was able to halt the CPP before it could return into the corrupted instruction pointer on the stack. Therefore the

attack was stopped before any control flow change took place–a capability unique to a parallel monitoring architecture such as CuPIDS. Even had the buffer overflow not been directly detected, CuPIDS would have detected the control flow change to the stack and may have been able to make the same repair.

5.4 Time to Detect

Table 5.2
Buffer Overflow Time-to-Detect Measurements (40 samples)

Monitor Type (Results based on 40 samples)	Mean (# instr)	Stdev (# instr)	Max (# instr)	Min (# instr)
MP, Pin, Parallel	0	0	0	0
MP, Pin, Blk, Postcond	0	0	0	0
MP, Non-pin, Blk, Postcond	0	0	0	0
MP, Pin, Non-blk, Postcond	32,142	15,000	73,134	5,481
MP, Non-pin, Non-blk, Postcond	258,207	547,000	2,478,906	18,081
UP, Blk, Postcond	0	0	0	0
UP, Non-blk, Postcond	33,807,836	23,676,701	85,376,601	0
UP, Parallel	8,250,607	3,710,207	12,687,579	1,518,471

We ran a number of experiments to determine how quickly CuPIDS detected illegitimate events. Two types of tests were run: one that performed an invariant test upon notification that a variable access was complete, and one in which real-time monitoring was used. Measuring the detect times for these tests without a hardware-based in-circuit emulator (ICE) proved challenging. Our theory stated that CuPIDS' ability to perform simultaneous monitoring of memory shared using the virtual memory mapping capability would result in detection at the point the

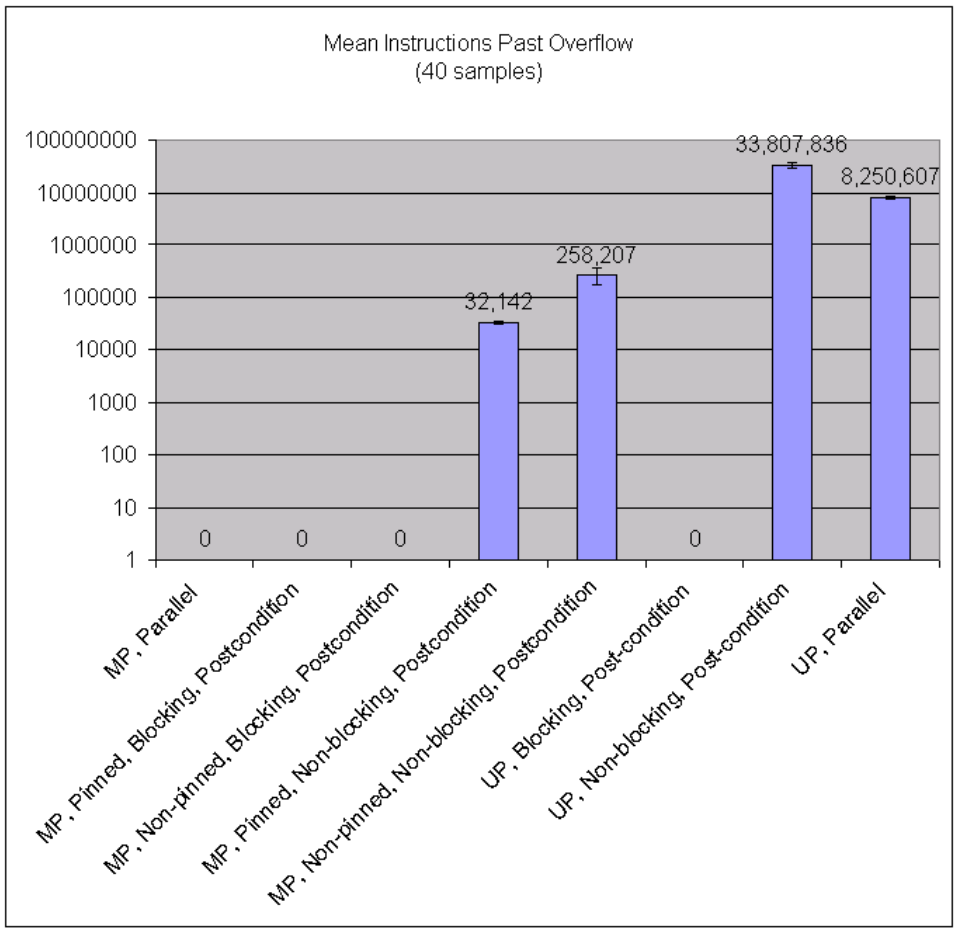

Figure 5.2. Mean Instructions Past Overflow Summary

invariant was violated[3]. Using the O/S clocks to mark violation and detection times was not feasible because of the overwhelmingly large overhead imposed by system calls. To quantify how quickly the CSP detects a problem we instrumented the CPP by adding counter that starts incrementing immediately following the completion of a monitored variable access. When the CSP detects a violation it immediately takes a snapshot of this counter. A buffer overflow in WU-FTP was used as the invariant violation. Each set of tests was run in both CuPIDS multi-processor (MP) mode

[3]Actually, at the point the cache snooping mechanism detected the shared usage of the memory location and propagated the change from the CPP's CPU into main memory and the CSP's CPU's cache.

and StUPIDS uni-processor (UP) mode, and the postcondition invariant tests were also run in blocking mode, where the CPP waited until the CSP signaled it was done with the invariant test, and non-blocking mode where the CPP notified the CSP that it was done with the variable modification and continued execution without waiting. 40 experiments were run for each of these eight configurations. The results of these experiments, summarized in Table 5.1 and Figure 5.2, are as follows:

5.4.1 Simultaneous Monitoring

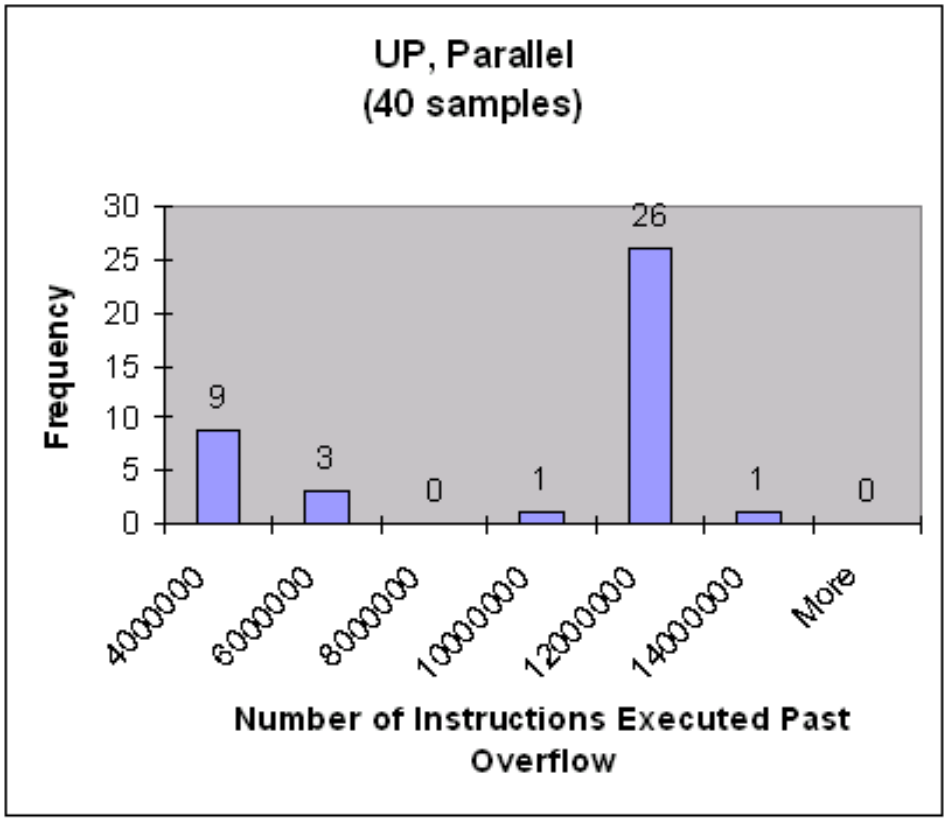

Figure 5.3. Mean Instructions Past Overflow: UP, Parallel

In these tests a monitoring task is started upon notification that a protected variable is to be accessed. In the CuPIDS case this monitor is placed on the CuPIDS

CPU and runs parallel with the CPP. In the StUPIDS case the monitor is scheduled as is any other task and its execution is interleaved with the execution of the CPP. The average of 8.2 million instructions executed by the UP CPP before overflow detection takes place compared to the immediate detection of the overflow in the CuPIDS CPP validates our research theory—that architectures such as CuPIDS can detect illegitimate events faster than can UP architectures. The histogram in Figure 5.3 shows the distribution of the experimental results.

5.4.2 Blocking Invariant Checking

In these tests, the CPP sends a blocking checkpoint event to the CSP immediately following the variable access. Because the CPP is not allowed to continue execution until the invariant test is complete it is not surprising that both MP and UP mechanisms immediately caught the overflow.

5.4.3 Non-blocking Invariant Checking

In these tests, the CPP sends a non-blocking checkpoint event to the CSP immediately following the variable access and continues execution. The consistent results from the CuPIDS CSP are expected, and reflect the amount of time it takes to perform the invariant test. The much higher and inconsistent results from the UP CSP reflect the scheduler-based non-determinism faced by all StUPIDS architectures. The distribution of the results are depicted in Figures 5.4, 5.5, and 5.6.

5.5 Conclusion

In this chapter we discussed the experimentation we performed using the prototype CuPIDS intrusion detection system. The results support our theory that the CuPIDS architecture has reasonable runtime costs.

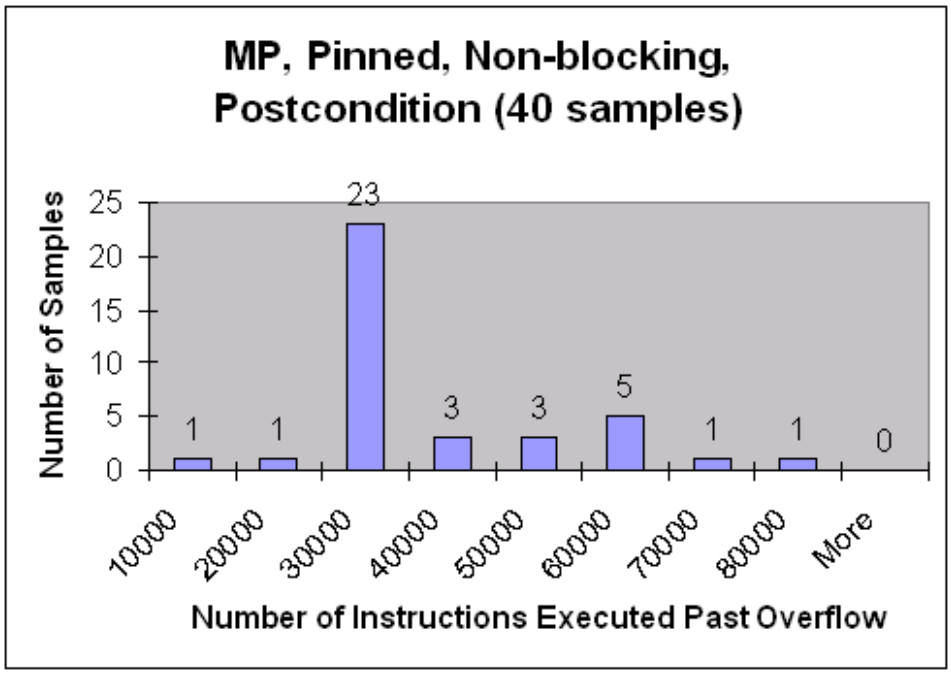

Figure 5.4. Mean Instructions Past Overflow: MP, Pinned, Non-blocking, Postcondition

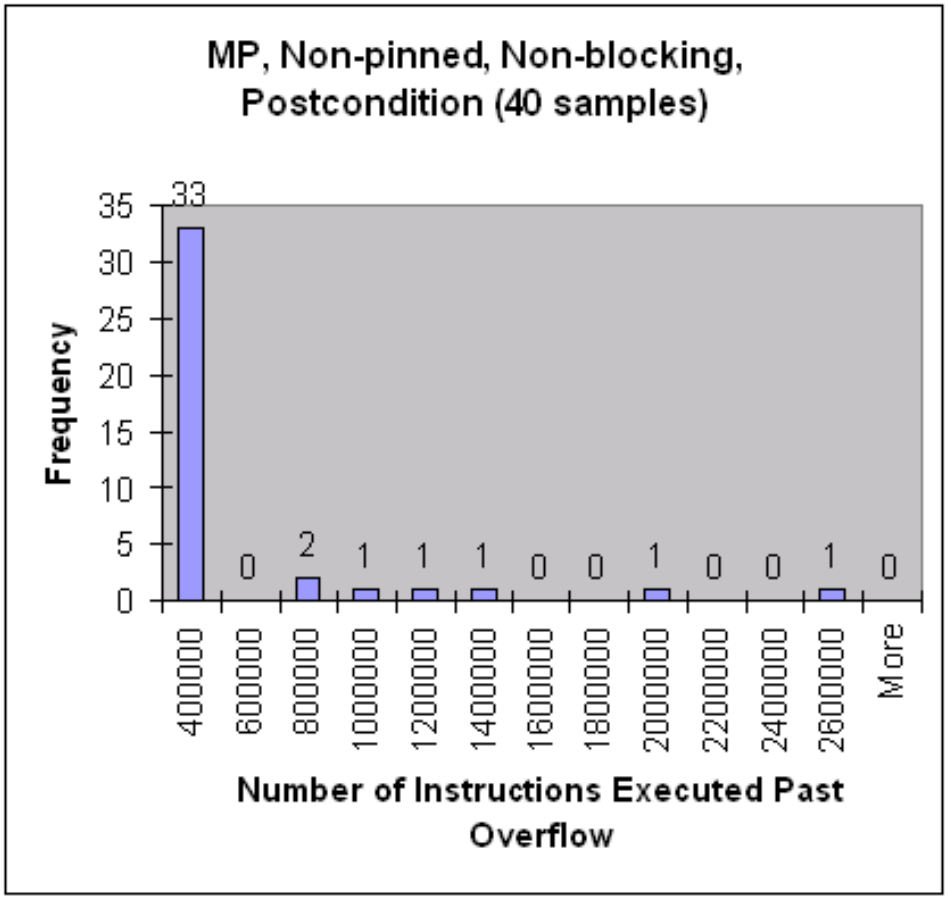

Figure 5.5. Mean Instructions Past Overflow: MP, Non-pinned, Non-blocking, Postcondition

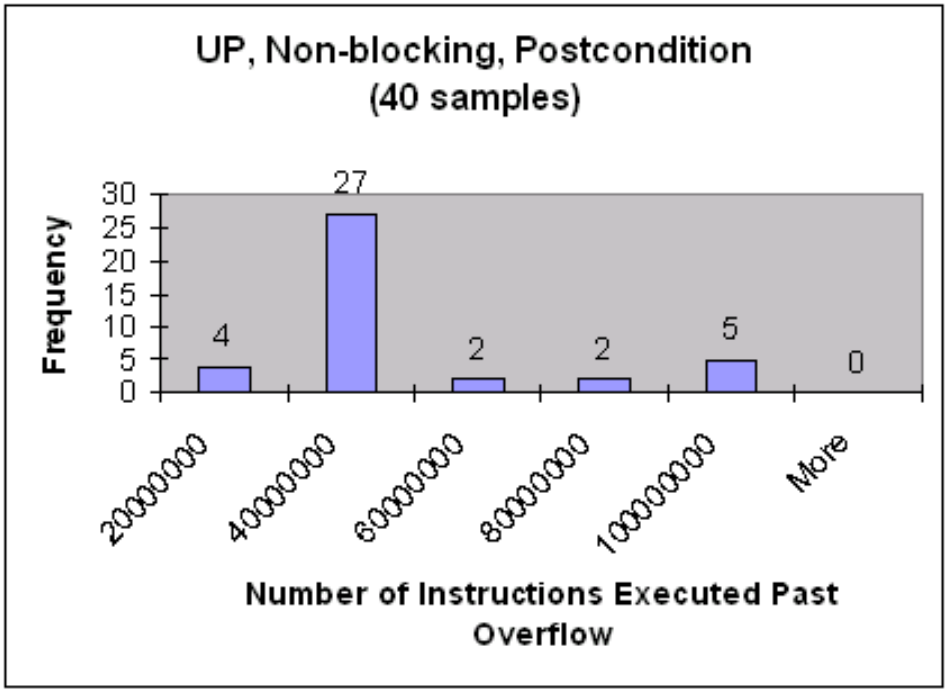

Figure 5.6. Mean Instructions Past Overflow: UP, Non-blocking, Postcondition

6 CONCLUSIONS, SUMMARY AND FUTURE WORK

Here we summarize all the important results of the dissertation.

6.1 Conclusions

CuPIDS represents a paradigm shift in information system security—one in which we shift away from the standard uni-processor intrusion detection (StUPIDS) model used by most architectures in this domain to a multi-processor model in which highly focused monitoring tasks run simultaneously with the processes they protect. Our results uncover gains in detection speed while incurring reasonable and minimal overhead costs. We demonstrate that running concurrently with attack code affords CuPIDS opportunities to detect and respond to attacks that are not available to StUPIDS. Additionally, because the opportunity exists to detect attacks while they occur without waiting for a context-switching event (either between user processes or between user and kernel mode) CuPIDS is able to respond more quickly and attacks are detected in realtime[1] and with high fidelity. We believe the use of parallel monitoring in the CuPIDS architecture resulted in the first real-time policy-non-compliance detection capability. These results represent advantages that are difficult or impossible to achieve on a uni-processor system—no matter how powerful.

6.1.1 Desirable Characteristics

Section 1.2.2 introduces ten desirable characteristics of an IDS. Here we describe how the CuPIDS architecture satisfies those characteristics, referencing the appropriate architectural or implementation sections as needed.

[1] As defined by Kuperman in [10]

1. Run continually: Our parallel monitoring requirements satisfies this goal literally, in ways a StUPIDS cannot. Section 3.4.2 on production and shadow processes describes how this is accomplished. Additionally, parts of the CuPIDS architecture are built into the host operating system and are therefore running whenever the system is running.

2. Fault tolerant: The operating system-based CuPIDS components reload with the rest of the system upon recovering from a crash, as do the program specific shadows. Additionally, under some circumstances CuPIDS is able to detect successful attacks and repair any damage done while the system continues to operate. For an example see Section 5.3.5.

3. Subversion resistant: The architecture is designed to make spoofing or subverting the security system difficult. For definitions of these terms see [56, 73]. Sections 3.5.1 and 3.8.1 discusses examples of how we address this difficulty.

4. Reasonable overhead: Our experimentation with a CuPIDS prototype has shown the overhead is reasonable. See Section 5.2 for details of the runtime overhead.

5. Configurable: The detector set described by the CuPIDS architecture is completely driven by security policy. See Section 3.7 for details.

6. Adaptable: In the current architecture, all of the intrusion detection tasks are performed by the CuPIDS shadow processes as defined in Section 3.4.2. As new CuPIDS applications are added to the system they will include their own shadow processes with associated detector sets.

7. Scalability: The primary target for this research is an organization's server farms—machines running a small number of critical applications with well-defined security boundaries. In this environment CuPIDS should scale well because the architecture's runtime overhead is per application. This means

that even as the number of protected applications increases, the overhead does not.

8. Graceful degradation: Each CuPIDS shadow is an independent process and is tied to a particular protected application. If an application is compromised, and manages to spoof or defeat its shadow, the protection for other protected processes should remain unaffected. The architecture is intended to make compromise of the CuPIDS kernel components difficult so that a successful attack against one user-level part of the system should not allow attacks against the whole CuPIDS mechanism.

9. Detection: It must be able to detect attacks:

   (a) No false positives: CuPIDS current detector set is specification-based. Any alarms are based upon violations of a specification or policy and are not false.

   (b) No false negatives: Given the current detector set, CuPIDS does not miss any attacks that it is searching for and therefore has a zero false negative error rate. However, there are certain to be attacks against the system that CuPIDS is not looking for, and these should perhaps be counted as false negatives should they occur.

   (c) Attack masking: CuPIDS was designed to make attacker masquerading [73] or spoofing attacks difficult. See Sections 3.6.2, 3.4.3 and 3.8.1 for more details about how this is accomplished.

   (d) Realtime detection: The parallel detection capability of CuPIDS allows for hard realtime detection as defined by Kuperman [10].

   (e) Generality: CuPIDS is designed to be policy agnostic in that it does not embody a particular security policy, but rather allows for a large variety of security policies to be represented by detector sets and whitelists. See

Section 3.6.1 for an example of how policy can be encoded in a form usable by the CuPIDS architecture.

10. Dynamic: Each protected application has its own monitor as described in Section 3.4.2. Changing the IDS is as simple as loading a new protected application. The background capabilities provided as operating system services are constructed as kernel modules wherever possible so that changes to these capabilities can be made without rebooting the machine. See Section 4.3.1 for more information.

6.1.2 IDS Challenges

Section 1.2.2 introduced a number of challenges for IDS architectures. This section discusses how we address several of those challengers.

- Attacker sophistication: Increasingly attacker sophistication, as well as increases in the complexity of software, is making intrusion detection more difficult. We address this concern by shifting away from a general detection system that attempts to protect the system as a whole, to one that tightly focuses its efforts on individual system components based upon explicitly-defined knowledge about operations a particular component is allowed and not allowed. This specification-based type of protection is similar to application IDS [74, 75] and is discussed further in Section 2.1.1. An application IDS, or application-aware IDS focuses much of its attention on a single, usually more vulnerable, application. We extend this definition to all system processes including user-mode applications as well as kernel-mode services such as device drivers.

- Encrypted messaging: By working at the application or service level, performing detection using well-defined operating environment specifications, and by having visibility into the state of the protected application we make more difficult the establishment of undetected, illicit encrypted or otherwise obfuscated

communications channels. Additionally, because CuPIDS has visibility into the state of the protected application it has access to the plaintext portion of encrypted communications.

- Automated response risks: As discussed in [9], in the general case, automated response to suspected intrusive activity is risky. Risks associated with responses involving only the local machine include denial of service to legitimate users if the alarm responded to is a false positive. The current, specification-based detector set in CuPIDS has no false positives. Therefore any alarm represents a real problem, and in some cases automated responses are safe and appropriate. An example of such an automated response mechanism is explored in Section 5.3.5.

- Attacks against IDS: We designed the CuPIDS architecture to be difficult for an attacker to subvert. The functionality of the architecture is separated into distinct shadow processes (see Section 3.4.2 for details). Each shadow process operates independently of the others, and the compromise of one should not make compromising another less difficult. There are some shared architectural components; however, these are intentionally simple and should be difficult to compromise.

- High false positive error rate: As discussed above in Section 1.2.1.9a, CuPIDS addresses the high false positive error rate using specification-based intrusion detection techniques as defined in Section 2.1.1.

- Unsecure infrastructures, limited network visibility and fast networks: CuPIDS is a host-based architecture and is not dependent upon gathering and deciphering network traffic to determine what is happening to a protected host.

6.2 Future Research Opportunities

Here we outline some possibilities for future research uncovered during the development of CuPIDS.

6.2.1 Desired Supportive Capabilities

While the results presented above show promise, we believe that a paradigm shift towards multi-processor security may lead to changes in the basic platform upon which architectures like CuPIDS are built. Some areas we anticipate exploring include:

Compiler support

Work is underway in exploring how a compiler can automatically generate events for variable lifecycle operations. As an example, as variables and data structures are allocated and used, appropriate events can be generated and dispatched. This work may be fully automatic, or it may allow or require that the programmer direct the compiler to do this work using a mechanism such as pragma, or assertions. The compiler will automatically generate code that informs the CSP when the variable is created, used, and destroyed. It will also automatically generate the code needed by the CSP to monitor that variable.

A related possibility is to have the compiler automatically construct a CSP using runtime invariant tests. This work is currently done manually, is neither difficult nor complex, and is a good candidate for exploration.

Hardware support

There are a number of areas in which better hardware support would ease the implementation of parallel architectures such as CuPIDS. Among these are the following:

- Better support for moving blocks of information between specific CPUs will be useful. As an example, the shared registers on the Xeon HTT processors provide a convenient scratchpad for small amounts of information.

- Additionally, better debugging capabilities can be designed. A capability similar to the debug registers but on shared memory, and possibly on larger data areas would be useful.

- The ability to set a memory write breakpoint on a CSP CPU and have it detect writes to that memory location by other CPUs would reduce the number of messages needed to keep track of CPP activity. It may be more practical to do this type of operation on multicore processors.

- Better visibility into the hardware state of a processor by another processor would dramatically ease the task of determining where the CPP is currently executing. In particular, an ability to see the state of the instruction pointer, stack pointer, and segment pointers would allow a CSP to determine, with precision, where the CSP is executing without the current instrumentation overhead. Read access to other registers may allow CuPIDS to monitor operations like string copies. The ability to write into registers may offer advantages as well. For example, if the CSP needs to pause the CPP briefly while completing an invariant test it could write the address of a NOP loop into the CPP's instruction pointer. This would freeze the CPP without the cost of a context switch. This capability may be easier to implement in a multi-core processor [21, 22].

Operating System Support

More efficient means of IPC designed specifically around an asymmetrical MP design such as CuPIDS are possible. CuPIDS' extensions to the FreeBSD API are

a start in this direction, and the extended inter-processor-interrupt (IPI) message passing system from the DragonFly BSD variant [76] would be useful.

Virtualization

As this line of research progresses we are interested in pursuing some way to virtualize the processor for times when it is useful to have low level visibility into the process' state. Virtualization as described in Section 2.6 would give visibility into the software/hardware interface. We do not intend to do this for the whole OS, only the processes under inspection, and then only in places where it may be useful. Our current alternative requires repeatedly preempting the process, or catching it while it is suspended.

6.2.2 Dynamic Binary Modification

In cases where program source is not available or for some reason it is not possible to instrument the CPP source with CuPIDS event triggers it may be possible to accomplish the same goals through modification of the program binary. Binary modification can be done before runtime such as described in [50] or at runtime such as illustrated in [77]. Initial thoughts into how the current function prolog and epilog event functionality may be duplicated involve inserting the interrupt-based IPC calls prior to e call and return instruction.

6.2.3 O/S Hardening/Self-protection

The growing body of work into mandatory access control (MAC) mechanisms such as those based on Biba's integrity-based [78], and Bell and LaPadula's multi-level security [31] models are used to provide a first-line defense against user application compromise. While MAC protection systems are not novel, the CuPIDS architecture uses hardware protection mechanisms in commodity CPUs to define

and protect the MAC mechanism and CuPIDS themselves against direct attacks that attempt to bypass its controls.

6.2.4 Relationship between Thesis and MAC

The secondary goal of protecting the security subsystem will likely involve the use of protective mandatory access control mechanisms. Additionally, it is likely that the security model defined as part of satisfying the primary goal will include the use of a policy agnostic MAC framework such as is available in Trusted BSD [66]. We hypothesize that the critical OS components may be adequately protected by MAC as long as the MAC mechanisms are protected by hardware. From a software perspective we believe that a combination of a low watermarking (LOMAC) integrity-based security policy implementation [79] and possibly a Multi-Level confidentiality security policy implementation will provide sufficient protection of critical components.

In addition to the work done by the system to validate user actions against the discretionary and mandatory access controls, we believe that performing either systematic or random checks of system state against the access control lists may provide a means of detecting attacks that bypass the access controls.

Saltzer and Schroeder's historic good security design principles [25] include the need for complete mediation. This states that all accesses to an object must be checked to ensure they are allowed. Enforcing this requirement literally with MAC is difficult. For example, every single access of a particular memory location may need to be checked against the current ACLs. This is completely impractical in current systems without hardware support, and so is simulated by the operating system. The level of checking that can be done must be offset by the amount of work required of the system, and in a uni-processor system it seems that relatively course-grained mediation is the best that can be achieved. We hypothesize that by using a secondary processor to monitor the activity of the production processor and validating its

activities against the ACLs the granularity of mediation can be decreased. Because the co-processor will likely have other work to do, this type of activity may not take place all the time, but can be used when production processes are in code regions known to be vulnerable [20], or when they are performing potentially dangerous tasks.

6.3 Summary

The introduction to this dissertation contained our research hypothesis stating that under some circumstances IDS architectures such as CuPIDS can be more effective than uni-processor-based IDS. We have designed an architecture based upon the paradigm of parallel security monitoring, constructed a prototype CuPIDS system and used the prototype to generate support for our hypothesis.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Douglas R. Hofstadter. *Gödel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979.

[2] Roff Smith and Joel Sartore. Nebraska: Standing tall again. In *National Geographic*, pages 114–140, Washington District of Columbia, United States of America, 1998. National Geographic Society.

[3] United States Joint Chiefs of Staff Chairman. Joint pub 3-13: Joint doctrine for information operations. URL `http://citeseer.ist.psu.edu/352281.html`, 1998.

[4] Paul D. Williams. *CuPIDS: Increasing Information System Security Through The Use of Dedicated Co-processing*. PhD thesis, Purdue University, West Lafayette, IN, 08 2005.

[5] CERT Coordination Center. CERT/CC statistics 1988-2005, 2005. `http://www.cert.org/stats/`.

[6] Lawrence Gorden, Martin Loeb, William Lucyshyn, and Robert Richardson. 2004 CSI/FBI computer crime and security survey. *Computer Security Institute*, 2004.

[7] F. Cohen. Computer viruses: theory and experiments. *Computers and Security*, 6(1):22–35, 1987.

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

[9] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy. Technical Report 99-15, Chalmers Univ., March 2000.

[10] Benjamin A. Kuperman. *A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources*. PhD thesis, Purdue University, West Lafayette, IN, 08 2004. CERIAS TR 2004-26.

[11] B. Mukherjee, T. Heberlein, and K. Levitt. Network Intrusion Detection. *IEEE Network*, May 1994:26–41, 1994. Available electronically at URL `http://www.csc.ncsu.edu/faculty/lee/publications.html`.

[12] Jai Sundar Balasubramaniyan, Jose Omar Garcia-Fernandez, David Isacoff, Eugene Spafford, and Diego Zamboni. An architecture for intrusion detection using autonomous agents. In *Proceedings of the 14th Annual Computer Security Applications Conference*, pages 13–24. IEEE Computer Society, December 1998.

[13] Matt Bishop. *Computer Security, Art and Science*. Addison Wesley, San Francisco, CA, 2003.

[14] Dorthy Denning. An Intrusion Detection Model. *IEEE Transactions on Software Engineering*, Feb 1987:222–232, 1987.

[15] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. State of the practice of intrusion detection technologies. Technical Report CMU/SEI-99-TR-028, Carnegie Mellon Software Engineering Institute, 1999.

[16] V.I. Gorodetski. Agent-based model of information security system: Architecture and formal framework for coordinated intelligent agents behavior specification. Technical Report Feb 2000, St Petersburg Institute for Informatics and Automation, 2000.

[17] Tim Bass. Intrusion detection systems and multisensor data fusion. *Association for Computing Machinery. Communications of the ACM*, Apr 2000:99–105, 2000.

[18] Mark Crosbie and E. H. Spafford. Defending a computer system using autonomous agents. Technical Report COAST TR 95-02, Department of Computer Sciences, 1995. CSD-TR-95-022.

[19] Mark Crosbie and Eugene Spafford. Defending a computer system using autonomous agents. In *Proceedings of the 18th National Information Systems Security Conference*, volume II, pages 549–558, October 1995.

[20] Diego Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Purdue University, August 2001.

[21] Intel Corporation. Intel's roadmap for multi-core processors. Web page at `http://www.intel.com/cd/ids/developer/asmo-na/eng/201969.htm?page=6`, July 2005.

[22] Advanced Micro Devices. AMD multi-core technology. Web page at `http://multicore.amd.com/en/`, July 2005.

[23] Intel Corporation. Hyper-threading technology. Web page at `http://www.intel.com/technology/hyperthread/`, July 2005.

[24] Timothy W. Curry. Profiling and tracing dynamic library usage via interposition. In *USENIX Summer*, pages 267–278, 1994.

[25] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[26] WikiPedia – the free encyclopedia definition of the term satisficing. Web page at `http://en.wikipedia.org/wiki/Satisficing`, 2005.

[27] Mark Crosbie, Bryn Dole, Todd Ellis, Ivan Krsul, and Eugene Spafford. IDIOT—users guide. CSD-TR 96-050, COAST Laboratory, Purdue University, 1398 Computer Science Building, West Lafayette, Indiana, September 1996.

[28] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley & Sons, Inc., 2002.

[29] Paul. K. Harmer, Paul. D. Williams, Gregg. H. Gunsch, and Gary. B. Lamont. An artificial immune system architecture for computer security applications. *IEEE Transactions on Evolutionary Computation*, 6:252–280, June 2002.

[30] Paul Williams, Kevin Anchor, John Bebo, Gregg Gunsch, and Gary Lamont. CDIS: Towards a Computer Immune System for Detecting Network Intrusions. In *Proceedings of the 4th International Symposium, Recent Advances in Intrusion Detection 2001*, pages 117–133, Berlin, 2001. Springer-Verlag.

[31] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corp., Bedford MA, May 1973.

[32] Benjamin A. Kuperman. Categorizing CSM Systems to Derive Audit Source Specifications. In *Proceedings of the 2003 CERIAS Research Symposium.* 2003 CERIAS Research Symposium, 2003.

[33] J. D. Tygar and Bennet Yee. Dyad: A system for using physically secure coprocessors. In *IP Workshop Proceedings*, 1994.

[34] W. A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *SP '97: Proceedings of the 1997 IEEE Symposium on Security and Privacy*, page 65, Washington, DC, USA, 1997. IEEE Computer Society.

[35] Xiaolan Zhang, Leendert van Doorn, Trent Jaeger, Ron Perez, and Reiner Sailer. Secure coprocessor-based intrusion detection. In *ACM European SIGOPS 2002*, 2002.

[36] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. CoPilot - a coprocessor-based kernel runtime integrity monitor. In *USENIX Security Symposium*, pages 179–194. USENIX, 2004.

[37] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, Fairfax, Virginia, November 1994. ACM Press.

[38] J. Molina and W. A. Arbaugh. Using independent auditors as intrusion detection systems. In *Proceedings of the Fourth International Conference on Information and Communications Security (S. Qing, F. Bao, and J. Zhou, eds.)*, volume 2513 of LNCS, pages 291–302, 2002.

[39] James P. Anderson. Computer security technology planning study. Technical Report ESD-TR-73-51, Vol. II, HQ Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, 01730, 1972.

[40] National Institute of Standards and Technology and National Security Agency. Common criteria for information technology security evaluation version 2.1. Technical Report 1, National Institute of Standards and Technology, 1999.

[41] R.J. Lipton, S. Rajagopalan, and D.N. Serpanos. Spy: A method to secure clients for network services. In *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, 2002.

[42] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the First International Conference on Security in Pervasive Computing*, 2003.

[43] John Wilander and Mariam Kamkar. A comparison of publicly available tools for dynamic buffer overflow prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium*, pages 149–162, San Diego, California, February 2003.

[44] Tal Garfinkel and Mendel Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Internet Society's Network and Distributed Systems Security Symposium*, February 2003.

[45] Harish Patil and Charles Fischer. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software — Practice and Experience*, 27(1):87–110, 1997.

[46] Haizhi Xu, Wenliang Du, and Steve J. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. In *Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection*, 2004. In Proceedings of the Seventh International Symposium on Recent Advances in Intrusion Detection.

[47] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly detection using call stack information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62. IEEE Computer Society, 2003.

[48] Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek. Efficient intrusion detection using automaton inlining. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2005.

[49] David Wagner and Drew Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, page 156. IEEE Computer Society, 2001.

[50] Henry Hanping Feng, Jonathon T. Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy*, 2004.

[51] Diego Zamboni. *Doing Intrusion Detection Using Embedded Sensors*. PhD thesis, Purdue University, 2000. CERIAS TR 2000-21.

[52] Clifford Stoll. *The cuckoo's egg: tracking a spy through the maze of computer espionage*. Doubleday, New York, NY, USA, 1989.

[53] Marshall Kirk McKusick and George V. Neville-Neil. *The Design and Implementation of the FreeBSD Operating System*. Addison-Wesley Longman, Inc, Boston, 2005.

[54] Intel Corporation. Ia-32 intel architecture software developers manual volume 3: System programming guide. Web page at `http://developer.intel.com/design/pentium4/manuals/245472.htm`, 2005.

[55] WikiPedia – the free encyclopedia definition of the term blacklist. Web page at `http://www.wikipedia.org/wiki/Blacklist`, 2005.

[56] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, 2002.

[57] Stephanie Forrest, Steven Hofmeyr, Anil Somayaji, and Thomas Longstaff. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128. IEEE Computer Press, 1996.

[58] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[59] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Pearson Higher Education, 2004.

[60] FreeBSD. *strcpy, strncpy – copy strings*, 2005. FreeBSD 5.3 manual page.

[61] Eric Bryant, James Early, Rajeev Gopalakrishna, Gregory Roth, Eugene Spafford, Keith Watson, Paul Williams, and Scott Yost. Poly^2 Paradigm: A Secure Network Service Architecture. In *Proceedings of the 19th Annual Computer Security Applications Conference*, 2003.

[62] Pratyusa Manadhata and Jeannette M. Wing. Measuring a system's attack surface. Technical Report CMU-CS-04-102, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, January 2004.

[63] FreeBSD. FreeBSD Operating System. Web page at `http://www.freebsd.org`.

[64] Marshall Kirk McKusick, Keith Bostic, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Longman, Inc, Boston, 1996.

[65] Greg Lehey. Improving the FreeBSD SMP implementation. In *Proceedings of the 2001 USENIX Annual Technical Conference*, 2001.

[66] Robert N. M. Watson, Wayne Morrison, Chris Vance, and Brian Feldman. The TrustedBSD MAC framework: Extensible kernel access control for FreeBSD 5.0. In *Proceedings of the 2003 USENIX Annual Technical Conference, FREENIX Track.*, pages 285–296, San Antonio, Texas, June 2003.

[67] FreeBSD. FreeBSD Operating System. Web page at `http://www.freebsd.org/`, June 2005.

[68] TrustedBSD. Web page at `http://www.trustedbsd.org/`, January 2005.

[69] FreeBSD. *msgrcv – receive a message*, 2005. FreeBSD 5.3 manual page.

[70] FreeBSD. *msgsnd – send a message*, 2005. FreeBSD 5.3 manual page.

[71] Intel Tm-I. Ia-32 intel architecture software developers manual volume 1: Basic architecture. Web page at `http://developer.intel.com/design/pentium4/manuals/245472.htm`.

[72] Mitre. Common vulnerabilities and exposures. Web page at `http://www.cve.mitre.org/`, July 2005.

[73] Thomas H. Ptacek and Timothy N. Newsham. *Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection*, 1998. Secure Networks paper, available electronically at URL `http://www.nai.com/media/ps/nai_labs/ids.ps`.

[74] Andy Cuff. *Intrusion Detection Terminology (part one)*, 2003. Available electronically at URL `http://http://www.securityfocus.com/infocus/1728`.

[75] Andy Cuff. *Intrusion Detection Terminology (part two)*, 2003. Available electronically at URL `http://http://www.securityfocus.com/infocus/1733`.

[76] DragonFlyBSD. DragonFlyBSD Operating System. Web page at `http://www.dragonflybsd.org`, 2005.

[77] Vladimir Kiriansky, Derek Bruening, and Saman P. Amarasinghe. Secure execution via program shepherding. In *USENIX Security Symposium*, pages 191–206, 2002.

[78] K. Biba. Integrity considerations for secure computer systems. Technical Report TR-3153, Mitre, Bedford, MA, April 1977.

[79] Timothy Fraser. LOMAC: Low Water-Mark Integrity Protection for COTS Environments. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 230, Washington, DC, USA, 2000. IEEE Computer Society.

VITA

VITA

Paul Williams was born in Amarillo, Texas[2]. He dropped out of high school after the 11th grade and worked as an automobile mechanic, gas station attendant, commercial heating and air conditioning technician, and structural steel worker. Several years of fun but non-intellectually stimulating work convinced him that education is important so he went back to high school, earned his diploma, and joined the US Air Force, both to serve his country, and to work on an education. Over the last sixteen years he has served as a Cryptographic Technician and later as a Communications and Information Officer in four countries and seven states. Along the way, in addition to the various levels of technical and professional military education provided by the Air Force, he received his Bachelor's Degree in Computer Science Cum Laude and with Distinction in Computer Science from the University of Washington in 1996, his Master's Degree in Computer Science from the Air Force Institute of Technology as a Distinguished Graduate with the Commandant's Award for Best Master's Thesis in 2001, and his Doctor of Philosophy Degree in Computer Science from Purdue University in 2005. Yet somehow he feels that this is just the beginning....

[2]He holds both Texas and United States citizenships.