CERIAS Tech Report 2007-66

An Exploration of Highly Focused, Coprocessor-based Information System Protection

by Paul Williams and Eugene H. Spafford

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Available online at www.sciencedirect.com
ScienceDirect
Computer Networks 51 (2007) 1284–1298
Computer Networks
www.elsevier.com/locate/comnet

CuPIDS: An exploration of highly focused, co-processor-based information system protection ☆

Paul D. Williams *, Eugene H. Spafford

Air Force Institute of Technology, Wright-Patterson Air Force Base, Department of Electrical and Computer Engineering, OH, United States
CERIAS, Purdue University, West Lafayette, IN, United States

Available online 23 October 2006

Abstract

The Co-Processing Intrusion Detection System (CuPIDS) project explores improving information system security through dedicating computational resources to system security tasks in a shared resource, multi-processor (MP) architecture. Our research explores ways in which this architecture offers improvements over the traditional uni-processor (UP) model of security. One approach we examined has a protected application running on one processor in a symmetric multi-processing (SMP) system while a shadow process specific to that application runs on a different processor. The shadow process monitors the application process' activity, ready to respond immediately if the application violates policy. Experiments with a prototype CuPIDS system demonstrate the feasibility of this approach in the context of a self-protecting and self-healing system. An untuned prototype supporting fine-grained protection of the real-world application WU-FTP resulted in less than a 15% slowdown while demonstrating CuPIDS' ability to quickly detect illegitimate behavior, raise an alarm, automatically repair the damage done by the fault or attack, allow the application to resume execution, and export a signature for the activity leading up to the error.

Keywords: Intrusion detection; Information system security; Co-processor; Multi-processor; Security policy compliance monitoring

1. Introduction

This paper describes research into the Co-Processing Intrusion Detection System (CuPIDS)—an exploration into increasing information system security by dedicating computational resources to system security tasks in a shared resource, multi-processor (MP) architecture. We demonstrate that this architecture allows the use of higher fidelity monitoring models, particularly with regard to the timeliness of detection, but also in terms of finer-grained visibility into the execution of a protected application than is reasonably feasible using current monitoring paradigms (e.g., internal function call pattern monitoring versus system call pattern monitoring). The resultant decrease in detection time coupled with highly focused security policy compliance monitoring enables quicker response to erroneous activity then was previously possible. We demonstrate responses that prevent further damage to a compromised system as well as responses focused on self-healing—returning a

☆ The opinions expressed in this paper are those of the authors, and do not necessarily reflect the views of the US Air Force, or the US Government.

* Corresponding author.
E-mail address: pdwillia@woh.rr.com (P.D. Williams).

1389-1286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.comnet.2006.09.011

compromised system to a secure state automatically and before the compromise is able to effect system operation.

Our philosophical foundations are fourfold: high assurance is important, a great deal of information about how systems are supposed to operate is often available but rarely used, MP computer systems are becoming commonplace, and finally that information systems will be vulnerable to attack or erroneous behavior for the foreseeable future. A body of research into co-processing techniques for tasks such as secure booting and digital rights management exists, much of which centers around the use of specialized hardware such as cryptographic co-processors [1,2]. More recent work has investigated possible security enhanced, multi-core chip architectures [3]. However, not nearly as much work has been done in investigating how generalized security tasks can benefit from dedicated co-processing. Most past and present Intrusion Detection System (IDS) architectures assume a uni-processor environment, or do not explicitly make use of multiple processors when they exist. The advent of multicore processors from the mainstream processor manufacturers such as Sun, Intel and AMD will result in MP systems becoming more common outside the server farm. We believe this affords us novel opportunities to be creative with how system resources are allocated. In addition to the use of dedicated co-processing, our research differs from existing work in our use of highly focused monitoring techniques.

We are concerned with a very 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 or faults may be caused by the activities of local or external users or a combination of both.
- Attacks or faults may result in a system compromise without ever causing a context switching event.

We believe that under some circumstances CuPIDS can be more effective than Standard Uniprocessor-based Intrusion Detection/Intrusion Prevention Systems (StUPIDS).[1]

[1] The name StUPIDS is in tribute to the work done in Purdue's Coast Laboratory on the IDIOT intrusion detection system [4].

For our purposes *more effective* is shown by demonstrating that:

1. Running concurrently with attack code affords CuPIDS opportunities to detect and respond to attacks that are not available to StUPIDS.
2. 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 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.

2. Background

This section describes the time and intrusion detection domain with which we are concerned and briefly references related research.

2.1. Time domain

Because some of the primary gains we anticipate from CuPIDS are time-related, we need to clarify what time domain we are working in. To do so we draw from a recent categorization of computer security systems. Kuperman's Ph.D. dissertation [5] describes four major timeliness categories in which detection can be accomplished: real-time, near real-time, periodic and retrospective. It is in the categories of real-time and near real-time that CuPIDS offers significant gains over StUPIDS.

To specify what we mean by real-time and near real-time we borrow Kuperman's notation. We represent the set of events taking place in a computer system by the set E. This set contains suspect events B such that $B \subseteq E$ and there exist events a, b, and c such that $a, b, c \in E$ and $b \in B$ The notation t_x represents the time of occurrence of event x. Finally, we need a detection function $D(x)$ that determines the truth of the statement $x \in B$.

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 an event, c, dependant upon b takes place. Given E, real-time detection requires the ordering

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

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

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

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 $D_{\text{StUPIDS}}(b_i)$ exists a StUPIDS can at best claim near-realtime detection with $\delta = CPUQuantum$, where $CPUQuantum$ represents the average amount of time each process is allocated by the system scheduler. 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. Detection domain

Among the factors that make intrusion detection in generalized computing environments difficult is the wide range of capabilities that must be protected. By forcing the security system designer to cover a wider range of resources, the defensive assets are, in a sense, "stretched thinner" than they will be in the highly focused CuPIDS environment. CuPIDS' ability to concentrate the right defenses at the right time on critical tasks coupled with the ability to use well-defined security boundaries as defined by the program designer and system security policy allows the exploration of highly effective intrusion detection functions. These benefits are not without cost, however. The combination of parallel-based monitoring and tightly focused detectors is primarily effective when the detection algorithm can keep up with the monitored process. In the notation from Section 2.1, CuPIDS' benefits reside mostly in the domain where $t_{D(b)}$ is small enough that $t_{D(b)} < t_c$ can be guaranteed. This means CuPIDS' detectors are mostly limited to $O(1)$ algorithms with small constants. This restriction is significant; however, there exist many attacks or errors which can be detected by short, fast algorithms—data structure invariant violations such as buffer overflows or call stack violations are two examples. Furthermore, even in the cases where real-time detection is not possible, CuPIDS can claim improvements in near real-time detection times over traditional architectures. These improvements stem from CuPIDS application of the appropriate monitoring function at the appropriate points in monitored application execution.

While our research is generally applicable to any computing environment in which multiple processors are available, we anticipate that it will be most useful in the dedicated server environment. Ideally, these machines are not used for general purpose computing and run only a streamlined set of applications dedicated to the service the system provides. These simplified configurations are not only simpler to maintain, but their smaller attack surfaces [6] are simpler to defend as well.

2.3. Prior research

There exists an enormous body of work on techniques for detecting and preventing violations of security policy. Axelsson's in-depth taxonomy and survey of the field of intrusion detection in 2000 is a good starting point for those unfamiliar with the field [7]. We draw from those techniques and augment them in ways that make use of the MP paradigm. Many of the specific intrusion detection techniques a CuPIDS will use differ from their StUPIDS counterparts only in the real-time, simultaneous monitoring nature of their use. Of particu-

lar interest to us are those efforts that separate runtime error checking from runtime execution, those modeling the state of the production process externally, and those making use of coprocessors or virtual machine architectures in performing monitoring tasks. This section presents only a sampling of the relevant literature given in [8].

2.3.1. Debugging

An example from the separate runtime error checking body of research is that done by Patil and Fischer [9] on detecting runtime errors in array and pointer accesses. They point out that including runtime error checking may slow applications by as much as a factor of 10, which is an enormous price to pay 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.

2.3.2. External modeling

Research into performing intrusion detection via external modeling of application behavior such as the work done by Haizhi Xu et al. [10] in using context-sensitive monitoring of process control flows to detect errors is a good example of external modeling. 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 good 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. [11] 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. [12] 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 [13,14] 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 shadow process in CuPIDS, getting its inputs from the CuPIDS event streams.

2.3.3. Virtualization and co-processors

ID has been performed using both machine virtualization and the use of dedicated co-processors [1,2,15–18]. An example of the latter category includes the work done by Zhang et al. [16] in describing how a crypto co-processor is used to perform some host-based intrusion detection tasks. In their research 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. Strengths of this approach include high attack resistance for code running in the co-processor system. Drawbacks of the approach include the lack of ready visibility into

the actions of the main processor and operating system.

These strengths and drawbacks also exist in the use of virtual machine architectures. Garfinkel and Rosenblum discuss a novel approach to protecting IDS components [18]. They pull the IDS out of the host and place it in the virtual machine monitor (VMM) with the primary goal of enhancing attack resistance. This approach has the benefit of largely isolating the IDS from code running in the virtual host. The VMM approach has much in common with the reference monitor work discussed by Anderson [19] and Lipton [20] 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 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 has visibility only of the 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 O/S is placing on those hardware items. By running as part of the host O/S, CuPIDS maintains complete visibility of the software state of the entire system, but currently lacks the protection afforded to VMs and secure co-processor architectures. Future work on CuPIDS will use hardware protection mechanisms such as those in the Intel IA32 [21] processor line to provide protection of security specific components as well as critical operating system components.

3. CuPIDS architecture

The CuPIDS architecture is fully described in [8], and is summarized here.

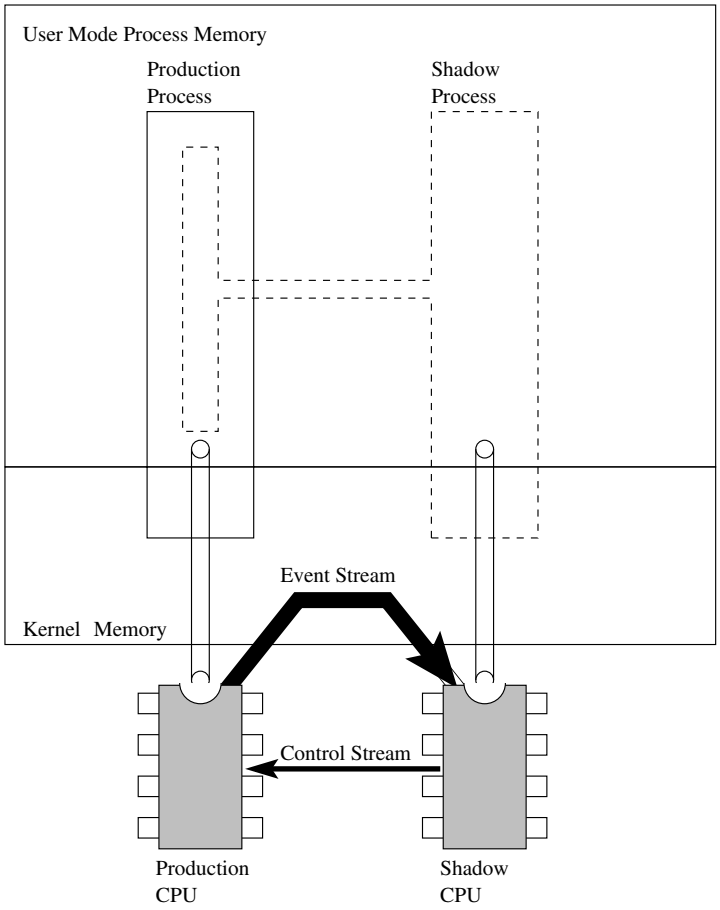

Fig. 1. High level overview of the CuPIDS architecture.

3.1. High level design

The basis of the CuPIDS architecture is parallel monitoring of the activities of a process by another process as the first process executes. Fig. 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 (only the internal state of the CPP's CPU is currently not visible to the CSP). This enables non-intrusive security monitoring while the protected process executes. Finally, the shadowing process is able to control the activities of the monitored application. This control capability allows CuPIDS to protect the process and system when illegitimate behavior is detected by preventing the illegitimate execution of a compromised process (e.g., halting a process in which a buffer overflow has occurred before any injected code can be executed).

CuPIDS operates using the facilities and capabilities afforded by a general purpose symmetrical multi-processing (SMP) computer 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 load [22]. CuPIDS differs from these architectures in that at any point in time one or more of the CPUs in a system are used 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. One possible CuPIDS software architecture is depicted in Fig. 2. The dark components represent production tasks and services and run on one CPU while the light components represent the CuPIDS monitors and run 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, where we seek to minimize the runtime penalty imposed by the security system, and protection, where we are concerned with the completeness of detection. By ensuring the processes responsible for detecting bad events are actively monitoring the system during periods in

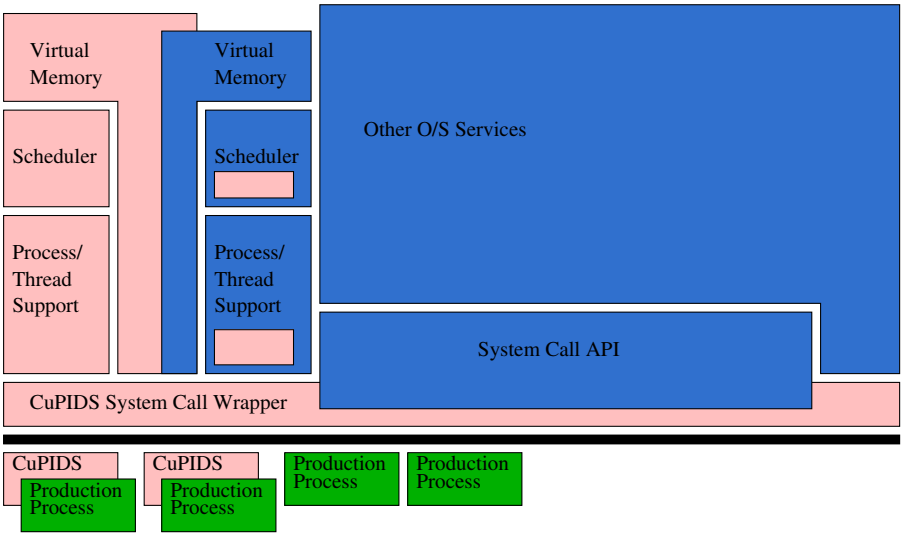

Fig. 2. Basic software architecture.

which bad events can occur—the CuPIDS architecture requires that when a CPP is executing its associated CSP is also on a CPU—we provide a real-time detection capability (using Kuperman's notation as defined in Section 2.1). 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.

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 Fig. 3.

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 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.

Our initial work assumes the CPP developer is aware of CuPIDS and the CPP communicates its state to the CSP by sending a stream of messages about events of interest to CuPIDS. Later work will investigate what types of real-time monitoring are possible for uninstrumented applications.

3.2. Protective activities

The CuPIDS architecture currently supports three types of protective activities: Application startup/shutdown validation, state monitoring, including invariant testing, and execution monitoring.

Application startup/shutdown: Startup tasks include verifying the authenticity of both the CSP and CPP as well as any supporting configuration files. The CSP is loaded and started executing. It then loads the CPP into memory, establishes any needed hooks into the CPP's VM space, initializes the various event communication systems, and finally starts the CPP running. Shutdown tasks include verifying that the CPP shutdown path followed a legitimate code path. Additionally, any runtime history data is saved to disk.

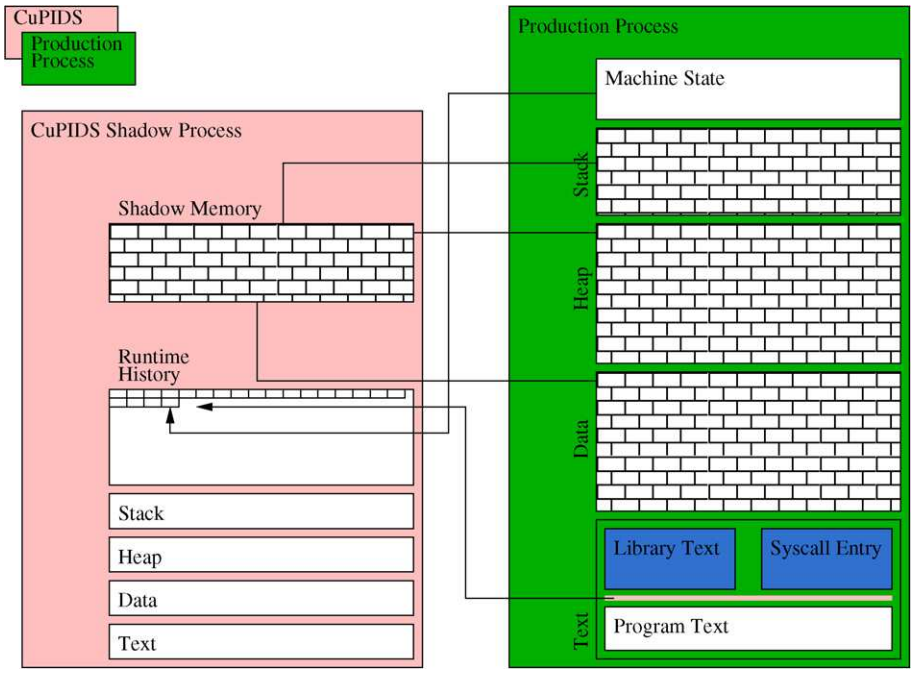

Fig. 3. CSP and CPP details.

State monitoring/Assertion verification: By creating appropriate hooks into the kernel, CSP is able to monitor nearly all aspects of the CPP's operating environment and state. This includes the CPP's entire VM space and any related kernel data structures and excluding 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 generation hooks 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 to the CPP's virtual memory space and executed. Currently these are manually written; however, work is underway to allow a programmer to indicate, to the compiler via pragmas, that a particular variable needs protection and the compiler will automatically generate the invariant testing code in the CSP.
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.

In addition to the direct monitoring of the CPP performed by the CSP, CuPIDS has a number of background capabilities that 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 a streamlined, interrupt-based communication interface for moving event records from the CPP to the CSP running on a different CPU.

3.3. Self-healing/self-protection

There are a number of well-known-to-be-dangerous library functions (such as those associated with string handling) and syscalls (such as those associated with invoking a system shell) [23]. Among the most common exploits publicly available are buffer overflows that use unsafe string handling library functions to overflow vulnerable buffers. Using a combination of stack modeling, library call event monitoring and 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). 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) had been used. In addition, error variables or signals may be set to indicate that something unexpected occurred.

Many of the shadowing ideas CuPIDS uses can be implemented without requiring the use of multi-processing. CuPIDS makes use of the data available to the shadow in a parallel environment to repair corrupted data structures without having to pause the CPP; however, these capabilities can also be effectively used in a non-CuPIDS architecture if detection is timely enough and the monitored

process can be paused. An example might be an IDS which interposes itself in the system call interface—a commonly used monitoring paradigm [24,25]. Upon an invocation of a dangerous system call the monitor could make a memory snapshot before passing the call to the operating system. If the call damages the data structure in use, the interposing IDS can use the snapshot to repair the damage in a manner akin to CuPIDS. The benefits offered by CuPIDS in this domain are the inverse of the interposing system call IDS' weaknesses. Wagner and Soto [26] discusses ways in which the pattern matching behavior of many system call interpositioning IDS can be evaded by mimicking legitimate patterns of system calls. CuPIDS' low-level visibility of the state of the CPP negates this type of attack by ensuring not only legitimacy of system call patterns for a particular CPP, but also by ensuring the call originated from a valid location in the CPP's address space and that the execution sequence prior to the call was correct. Garfinkel [27] discusses how difficult it is to get the monitoring interface right across the entire range of possible system call usage by user applications. CuPIDS' tight coupling of monitor with application, at the individual function call level allows, in many cases, precise validation of inputs too, and expected outputs from dangerous system calls. This precision can reduce or eliminate the need for expensive general-case anomaly detection algorithms. Finally, [28] highlights the (often prohibitively expensive) runtime cost overhead incurred in passing control to the monitors as well as the general-case anomaly detection which must be done every time the monitor is invoked. CuPIDS' ability to model the behavior of an interposing IDS without the context-switching overhead is a significant improvement.

4. Implementation

We have implemented a prototype CuPIDS. This section briefly describes the current state of that prototype. For a more in-depth discussion of the implementation see [8]. Our experimentation uses FreeBSD, currently 5.3-RELEASE [29]. 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, a means by which signals destined for—and some interrupts caused by—the CPP are routed to the monitoring CSP, etc. 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 3.2, and hooks 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.[2] 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 automatically 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 executes, the CuPIDS instrumentation compiled into it sends messages to the CSP notifying it about programmer-defined operational activities such as protected variable lifetime events (creation, accesses and deletion). The automatically generated control flow events (currently all function call entry and exits, to include library and syscall invocations) are passed to the CSP as well. In the case of execution flow events such as function or system calls, the event generation mechanism makes use of low-level system primitives[3] to include in the event a non-user spoofable source and return address for the flow changing call. 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, etc.

5. Results

Upon completing the initial CuPIDS implementation, a series of tests were performed to explore its behavior. The experiments as described in [8] and summarized here were designed to determine

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

[3] The last three branch records stack available in Intel IA32 processors as described in Section 15.5 of [30].

if the CuPIDS architecture, as embodied in the prototype, supported or refuted our research hypothesis. The prototype allowed us to verify basic CuPIDS functionality. The system is able to correctly load and execute CPP and CSP components, and the CSP is able to detect invariant and security policy violations as well as illegitimate control flow changes. Upon detecting a fault or attack, the CSP is able to halt the CPP, raise an alarm and save the state of the CPP's memory and execution trace history. In some cases automatic repair is possible. An example of such a case is where real-time detection has occurred (e.g., $t_{D(b)} < t_c$, where b is the corruption of the stack by a buffer overflow and c is the execution of that injected code), and sufficient information exists that repair of corrupted process state is possible (e.g., CuPIDS made a memory snapshot of the stack region immediately prior to the overflow). In these cases CuPIDS can repair the damage from the attack or error. This repair may necessitate pausing the errant process, but our experimentation demonstrated that in some simple cases such as stack repair, CuPIDS was able to repair the corrupted CPP stack before the return into the injected code occurred, thus allowing the CPP to continue execution without interference. Additionally, the experiments demonstrate 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 one equipped with a comparable detector set.

5.1. Test design and methodology

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 server-class software typical of that used in our target environment (an organization's public-facing demilitarized zone); their source code is available so that we could examine and instrument them; and because they contain exploitable vulnerabilities as demonstrated by publicly available exploits.

5.1.1. Test platform

The experiments described below were run on a SMP platform with dual Xeon 2.2GHz processors, 1G RAM and a single 120GB ATA100 drive. Hyperthreading (HTT) was enabled so the operating system had 4 CPUs available. CuPIDS only controlled the scheduling of tasks on CPU1; the system scheduler was responsible for scheduling CPU0, CPU2 and CPU3. In a representative example experiment involving CuPIDS, the CSP was the only user of CPU1, the instrumented ftpd daemon used all of CPU0's cycles, the ftp client used all of CPU2's cycles, and the operating system, including the test drivers, ran mostly on CPU3. The test drivers ensure that all file I/O is done on local drives so that network overhead does not become a factor. We observed that the loading on the operating system CPU (CPU3 in the above example) was typically low, on the order of 5–10%, and that releasing CPU1 from CuPIDS control did not significantly impact system performance (CPU1 and CPU3 would both be mostly idle while CPU0 and CPU2 were saturated by the demands of fptd and the test client, respectively). 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 standalone mode (command line `ftpd -s` which 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 (the CPP), log in, change local and remote directories, and perform a series of 300 ftp file transfers and one `ls` (directory listing) for a total of 301 transfers. The intent of this workload was to stress ftpd, both internally and externally through file I/O system calls for long enough that meaningful time measurements could be made. 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 made, one using the CuPIDS interrupt-based IPC, one using SysV IPC, and one baseline test was run against a non-instrumented version of WU-FTP. The results are summarized in Table 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

Table 1
WU-FTP runtime performance measurements (50 samples)

Event comm. method		Clock time (s)	User time (s)	Sys. time (s)	Throughput (MB/s)
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

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 1 the overhead of generating and using those events was around 15% for the CuPIDS IPC as opposed to approximately 100% 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.

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 call invocations.
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 to functions that bypassed the prologue event generator. It did so by detecting illegitimate program stack activity in the stack model.
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.
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 [31]. 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. *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 to 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 might have been able to make the same repair.

5.4. Time to detect

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 invariant was violated.[4] 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 a 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 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 2 are as follows:

1. *Simultaneous monitoring*: 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.
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.
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.

[4] 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.

Table 2
Buffer overflow time-to-detect measurements (Pin means that the CSP was pinned to the CuPIDS CPU and synchronized with the CPP; non-pin means the CSP was scheduled like any other process. Blk indicates the detector function was run inline with CPP execution; non-blk means the detector function was not inline.) (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

6. Future work

6.1. 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 such as CuPIDS are built. Some areas we anticipate exploring include:

Compiler support: The compiler can automatically generate events for variable lifecycle operations. As an example, as buffers are allocated and used appropriate events can be generated and dispatched. Another alternative is to allow the programmer to direct the compiler to do this work using a mechanism such as pragma, or assertions.

Hardware support: 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 that can operate 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.

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 [32] would probably be useful.

6.2. Self-protection

Mandatory access control (MAC) models such as Biba's integrity-based model [33], and Bell and LaPadula's multi-level security [34] models might be 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. We are currently investigating the use of hardware primitives such as Intel's virtualization technology [35] to protect the portions of CuPIDS which reside in the operating system from compromises of kernel-level processes.

7. Conclusion

For many information systems, high assurance, in terms of keeping an application running in spite of faults or attacks, is more important than raw performance. This is particularly true for an organization's mission critical applications and servers. We believe and demonstrate that dedicating one or more processors in a MP system specifically to security tasks can increase system robustness in the face of faults and attacks. We further believe offloading the security work from the production parts of the system will allow the use of security techniques which may be too computationally expensive when performed inline.

Examples of such techniques include the runtime debugging checks and assertions employed during

the software development process. Checks such as these are commonly placed in vulnerable or critical code during the debugging and testing phases of the software lifecycle but are removed from shipping code because of the runtime performance degradation they impose [9]. A great deal of specifically focused information about how an application is intended to behave is available to system architects and developers; however, we do not believe this wealth of information is commonly used in runtime security monitoring of production systems. The CuPIDS architecture is specifically designed to make use of such information in a reasonably efficient manner.

While trading performance for security is not a new idea, we believe our combination of dedicating computational resources to running highly focused monitoring functions in parallel with protected production code is both novel and worthwhile. We believe the CuPIDS architecture to be more effective than StUPIDS architectures in terms of real-time detection of bad events as well as offering some novel detection techniques based upon the low-level and parallel nature of the monitoring. By dedicating computational resources explicitly to security tasks we are trading performance for security; however, by offloading some security tasks from the production process to the security process and running them in parallel we are decreasing the workload of the system production components. We have constructed a prototype of this architecture and used it to verify CuPIDS basic functionality.

The CuPIDS architecture is novel in that we explicitly divide the system into production and security components, embed explicit knowledge of how the production components are intended to operate into specialized security monitors and ensure the appropriate security component is running on a processor whenever a particular production component is running on a different processor. The architecture allows fine-grained visibility into the operation of a protected process. We intend the CuPIDS architecture to be detection model agnostic—capable of supporting many different IDS.

The detection capability of CuPIDS is currently all specification or white-list-based. Therefore it has a zero false positive error rate; thus the alarms output from CuPIDS are suitable for use by automated response systems. In fact, much of CuPIDS strength derives from its automated response capabilities. Its tightly focused, parallel monitoring capability allows for rapid detection of and response to illegitimate behavior. The combination of real-time detection (discussed in Section 2.1) allowed by parallel processing and an ability to automatically repair some damage afforded by CuPIDS' low-level interface into the host operating system let CuPIDS not only stop the attacks, but help maintain operation of critical components of systems.

References

[1] J.D. Tygar, B. Yee, Dyad: A system for using physically secure coprocessors, in: IP Workshop Proceedings, 1994. URL: http://www.citeseer.nj.nec.com/tygar91dyad.html.

[2] W.A. Arbaugh, D.J. Farber, J.M. Smith, A secure and reliable bootstrap architecture, in: Proceedings 1997 IEEE Symposium on Security and Privacy, May 1997, 1997, pp. 65–71. URL: http://www.citeseer.nj.nec.com/arbaugh97secure.html.

[3] W. Shi, H.-H.S. Lee, G. Gu, L. Falk, T.N. Mudge, M. Ghosh, An intrusion-tolerant and self-recoverable network service system using a security enhanced chip multiprocessor, in: ICAC, IEEE Computer Society, 2005, pp. 263–273.

[4] M. Crosbie et al., IDIOT Users Guide, Tech. Rep. COAST TR 96-04, Department of Computer Sciences, cSD-TR-96-050, 1996. URL: http://www.cerias.purdue.edu/techreports-ssl/public/96%-04.ps.

[5] B.A. Kuperman, A categorization of computer security monitoring systems and the impact on the design of audit sources, Ph.D. thesis, Purdue University, West Lafayette, IN, CERIAS TR 2004-26 (08 2004).

[6] P. Manadhata, J.M. Wing, Measuring a system's attack surface, Tech. Rep. CMU-CS-04-102, Carnegie Mellon University, Pittsburgh, PA (January 2004).

[7] S. Axelsson, Intrusion detection systems: A survey and taxonomy, Tech. Rep. 99-15, Chalmers University (March 2000). URL: http://www.citeseer.nj.nec.com/axelsson00intrusion.html.

[8] P.D. Williams, CuPIDS: Increasing information system security through the use of dedicated co-processing, Ph.D. thesis, Purdue University, West Lafayette, IN, CERIAS TR 2005-50 (08 2005). URL: http://www.cerias.purdue.edu.

[9] H. Patil, C. Fischer, Low-cost, concurrent checking of pointer and array accesses in C programs, Software Practice & Experience 27 (1) (1997) 87–110.

[10] H. Xu, W. Du, S.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.

[11] H.H. Feng, O.M. Kolesnikov, P. Fogla, W. Lee, W. Gong, Anomaly detection using call stack information, in: SP'03: Proceedings of the 2003 IEEE Symposium on Security and Privacy, IEEE Computer Society, 2003, p. 62.

[12] R. Gopalakrishna, E.H. Spafford, J. Vitek, Efficient intrusion detection using automaton inlining, in: Proceedings of the 2005 IEEE Symposium on Security and Privacy, IEEE Computer Society, 2005.

[13] D. Wagner, D. Dean, Intrusion detection via static analysis, in: SP'01: Proceedings of the IEEE Symposium on Security and Privacy, IEEE Computer Society, 2001, p. 156.

[14] H.H. Feng, J.T. Giffin, Y. Huang, S. Jha, W. Lee, B.P. Miller, Formalizing sensitivity in static analysis for intrusion detection, in: IEEE Symposium on Security and Privacy, 2004.
[15] O.S. Saydjari, LOCK: An Historical Perspective, in: Proceedings of the 18th Annual Computer Security Applications Conference, 2000, ACSAC, http://www.acsac.org, 2000, pp. Online, http://www.acsac.org.
[16] X. Zhang, L. van Doom, T. Jaeger, R. Perez, R. Sailer, Secure coprocessor-based intrusion detection, in: ACM European SIGOPS 2002, 2002. URL: http://www.research.ibm.com/vali/sigops2002_monitor.ps.
[17] J. Molina, W.A. Arbaugh, Using independent auditors as intrusion detection systems, in: S. Qing, F. Bao, J. Zhou, (Eds.), Proceedings of the Fourth International Conference on Information and Communications Security, vol. 2513 of LNCS, 2002, pp. 291–302.
[18] T. Garfinkel, M. Rosenblum, A virtual machine introspection based architecture for intrusion detection, in: Proceedings of the Network and Distributed Systems Security Symposium, 2003. URL: http://www.citeseer.nj.nec.com/garfinkel03virtual.html.
[19] J.P. Anderson, Computer security technology planning study, Tech. Rep. ESD-TR-73-51, Vol. II, HQ Electronic Systems Division (AFSC), Hanscom Field, Bedford, MA, 01730, 1972.
[20] R. Lipton, S. Rajagopalan, D. Serpanos, Spy: A method to secure clients for network services, in: Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops, 2002.
[21] Intel Corporation, IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture. URL: http://www.developer.intel.com/design/pentium4/manuals/245472.htm.
[22] A. Silberschatz, P.B. Galvin, G. Gagne, Operating System Concepts, John Wiley & Sons, Inc., 2001.
[23] G. Hoglund, G. McGraw, Exploiting Software: How to Break Code, Pearson Higher Education, 2004.
[24] S. Forrest, S.A. Hofmeyr, A. Somayaji, T.A. Longstaff, A sense of self for UNIX processes, in: Proceedings of the 1996 IEEE Symposium on Security and Privacy, IEEE Computer Society, 1996, p. 120.
[25] S.A. Hofmeyr, S. Forrest, A. Somayaji, Intrusion detection using sequences of system calls, Journal of Computer Security 6 (3) (1998) 151–180, URL: http://www.citeseer.ist.psu.edu/article/hofmeyr98intrusion.html.
[26] D. Wagner, P. Soto, Mimicry attacks on host based intrusion detection systems (2002). URL: http://www.citeseer.ist.psu.edu/wagner02mimicry.html.
[27] T. Garfinkel, Traps and pitfalls: Practical problems in in system call interposition based security tools, in: Proceedings of the Network and Distributed Systems Security Symposium, 2003. URL: http://www.citeseer.ist.psu.edu/garfinkel03traps. html.
[28] R. Sekar, M. Bendre, D. Dhurjati, P. Bollineni, A fast automaton-based method for detecting anomalous program behaviors, in: SP'01: Proceedings of the IEEE 2001 Symposium on Security and Privacy, IEEE Computer Society, Washington, DC, USA, 2001, p. 144.
[29] TrustedBSD, TrustedBSD. http://www.freebsd.org.
[30] Intel Corporation, IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide (2005). URL: http://www.developer.intel.com/design/pentium4/manuals/245472.htm.
[31] T.H. Ptacek, T.N. Newsham, Insertion, evasion, and denial of service: Eluding network intrusion detection, Technical report, Secure Networks, Inc. (January 1998).
[32] DragonFlyBSD, DragonFlyBSD. http://www.dragonflybsd.org.
[33] K. Biba, Integrity considerations for secure computer systems, Tech. Rep. TR-3153, Mitre, Bedford, MA (April 1977).
[34] D.E. Bell, L.J. LaPadula, Secure computer systems: Mathematical foundations and model, Tech. Rep. M74-244, The MITRE Corp., Bedford MA (May 1973).
[35] R. Uhlig, G. Neiger, D. Rodgers, A.L. Santoni, F.C.M. Martins, A.V. Anderson, S.M. Bennett, A. Kagi, F.H. Leung, L. Smith, Intel virtualization technology, Computer 38 (5) (2005) 48–56.

Paul D. Williams, Major, USAF, Ph.D., is an Assistant Professor of Computer Science and Cyber Operations in the Department of Engineering at the Air Force Institute of Technology, Wright-Patterson AFB, Ohio. He has served in many information operations roles, both operational and supporting, for seventeen years. His research interests center on cyber operations, and include algorithms, artificial intelligence, and computer architecture.

Eugene H. Spafford is one of the most senior and recognized leaders in the field of computing. He has an on-going record of accomplishment as a senior advisor and consultant on issues of security, cybercrime and policy to a number of major companies, law enforcement organizations, and government agencies, including Microsoft, Intel, Unisys, the US Air Force, the National Security Agency, the GAO, the Federal Bureau of Investigation, the National Science Foundation, the Department of Energy, and two Presidents of the United States.

He is a professor with a joint appointment in Computer Sciences and Electrical and Computer Engineering at Purdue University, where he has served on the faculty since 1987. He is also a professor of Philosophy (courtesy) and a professor of Communication (courtesy). He is the Executive Director of the Purdue University Center for Education and Research in Information Assurance and Security (CERIAS). He serves on a number of advisory and editorial boards, and has been honored several times for his writing, research, and teaching on issues of security and ethics.