CUPIDS ENHANCES STUPIDS: EXPLORING A CO-PROCESSING PARADIGM SHIFT IN INFORMATION SYSTEM

by Paul D. Williams, Eugene H. Spafford

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

Proceedings of the 2005 IEEE
Workshop on Information Assurance and Security
United States Military Academy, West Point, NY, 15–17 June 2005

CuPIDS enhances StUPIDS: Exploring a Co-processing Paradigm Shift in Information System Security

Paul D. Williams Eugene H. Spafford

Abstract— **The CuPIDS project is an exploration of increasing information system security by 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. There are a number of areas to explore, one of which has a protected application running on one processor in a symmetric multiprocessing (SMP) system while a shadow process specific to that application runs on a different processor, monitoring its activity, ready to respond immediately if the application veers off course. This paper describes initial work into defining such an architecture and the prototype work done to validate our ideas.**

I. Introduction and Philosophy

This paper describes initial research into the Co-Processing Intrusion Detection System (CuPIDS). We start with the philosophy that security is more important than performance—particularly with regards to mission critical applications and servers. This assumption—that we can afford to trade some performance for increased security—affords us opportunities to be creative with how system resources are allocated. 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. Yet, especially in the server world, multiple processor machines are commonplace, and with the advent of multi-core technologies from mainstream processor makers such as Sun, Intel and AMD, commodity computers are likely to have multiple processors. We believe we can improve the effectiveness of the security system and therefore the overall robustness of the entire computing system by taking advantage of the parallel processing capability now commonly available. We do so by dividing the system into production and security components, embedding explicit knowledge of how the production components are intended to operate into specialized security monitors and ensuring the appropriate security component is running on a processor whenever a particular production component is running on a different processor. The overall architecture places all user tasks and most of the operating system (O/S) into the production component and all parts of the O/S that pertain to the enforcement of security policy security, including the security monitoring and validating tasks, into the security component.

This initial paper describes research into performing fine-grained, real-time monitoring of specific production tasks. We discuss the new capabilities made possible by switching to a MP paradigm, describe the architecture in general terms and report the results of initial experimentation into building a CuPIDS system. We believe this architecture will allow IDS to use higher fidelity monitoring models, particularly with regard to the timeliness of detection, and will also increase system robustness in the face of some types of attacks.

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

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!

The initial impetus for this project was a realization that we may be able to embed information about what a process is and is not allowed to do in a monitoring process. This monitor, given full access to the resources used by the production process and running simultaneously with it in a SMP computer, can act as a "tilt-sensor," detecting and responding to operational deviations very quickly. Sources of information about the "normal," or allowed behavior of an application, may include directives and assertions defined by the production process programmer, automatically generated "shadow" output from a compiler, system security specifications and training data derived from monitoring

P. D. Williams, CERIAS, Purdue University, West Lafayette, IN.
E. H. Spafford, CERIAS, Purdue University, West Lafayette, IN.

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

the execution of processes on a system. It is not our intent to monitor production process execution instruction-by-instruction; we focus instead on protecting key data structures and monitoring operations in critical regions of code.

While this paper focuses on IDS, the techniques described are likely to be valuable in other domains such as software debugging, fault tolerance, and computer forensics.

II. Background

Information system security is a widely studied field. Here we discuss the threat model we are addressing, the area in security in which we are focusing and briefly reference research related to ours.

A. 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 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 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.

B. Time Domain

To clarify what time domain we are working in we draw from a recent categorization of computer security systems. Kuperman's Ph.D. dissertation [2] 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.

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

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

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

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

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 $D_{StUPIDS}(b_i)$ 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.

C. 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 functions are both efficient in terms of resource usage, and are more robust because the possible legitimate activities of the system are well defined, with aberrations or anomalies easier to detect.

D. Prior Research

There exists an enormous body of work into techniques for detecting and preventing violations of security policy (The body of literature here is huge, Axelsson's in-depth, thorough taxonomy and survey of the field of intrusion detection in 2000 is a good starting point for those unfamiliar with the field [3]). 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 particular 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.

D.1 Debugging

An example from the separate runtime error checking body of research is that done by Patil and Fischer [4] 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 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.

D.2 External Modeling

Research into performing intrusion detection via external modeling of application behavior is very active right now. The recent work done by Haizhi Xu et al. in using context-sensitive monitoring of process control flows to detect errors is a good example of external modeling [5]. 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. [6] 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. [7] 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 [8], [9] by using inlined function call nodes to dramatically reduce the non-determinism in their model while applying clever 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 very naturally into the CuPIDS architecture. The model simulation can be run as a CSP, getting its inputs from the CuPIDS event streams.

D.3 Virtualization and Co-processors

ID has been performed using both machine virtualization and the use of dedicated co-processors [10], [11], [12], [13], [14], [15]. An example of the latter category includes the work done by Zhang, et al. in describing how a crypto co-processor is used to perform some host-based intrusion detection tasks[13]. 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 [15]. 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 [16] and Lipton [17] 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 has visibility only 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 O/S is placing on those hardware items. By running as part of the host O/S, CuPIDS maintains complete visibility into the software state of the entire system, but currently lacks the protection afforded VMs and secure co-processor architectures. Future work on CuPIDS will use hardware protection mechanisms such as those in the Intel IA32 [18] processor line to provide protection of security specific components as well as critical operating system components.

III. CuPIDS Architecture

CuPIDS is intended to operate using the facilities and capabilities afforded by a general purpose symmetrical multiprocessing (SMP) computer architecture. In such an architecture there are some number, typically even, of central processing units (CPUs), 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.

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 [19]. 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. One possible CuPIDS software architecture is depicted in Figure 1. 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..

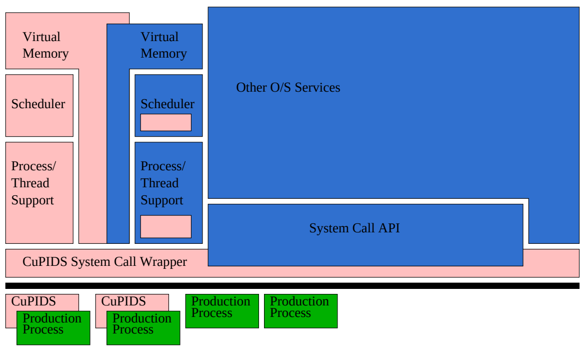

Fig. 1. Basic software architecture

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 hope to provide a real-time detection capability (using Kuperman's notation as defined in Section II-B). 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 security monitoring code and data segments from the memory segments used by the operating system and user programs.

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 2.

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. The monitoring performed by CuPIDS is both interactive and passive. In an example of interactive monitoring, the CPP

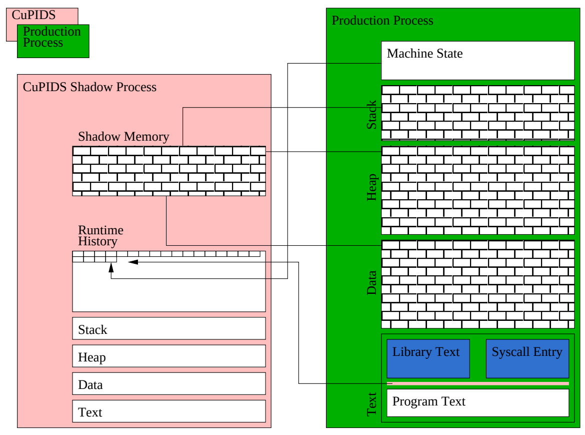

Fig. 2. CSP and CPP details

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. Other passive monitoring capability include 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, and does not interfere with or delay the processing of library or syscalls unless the security system detects a problem.

The flowchart in Figure 3 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 4 shows examples of the variable protection instrumentation embedded in the CPP (the *CuPIDS_var*... calls invoke the CSP notification mechanism), while Figure 5 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-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

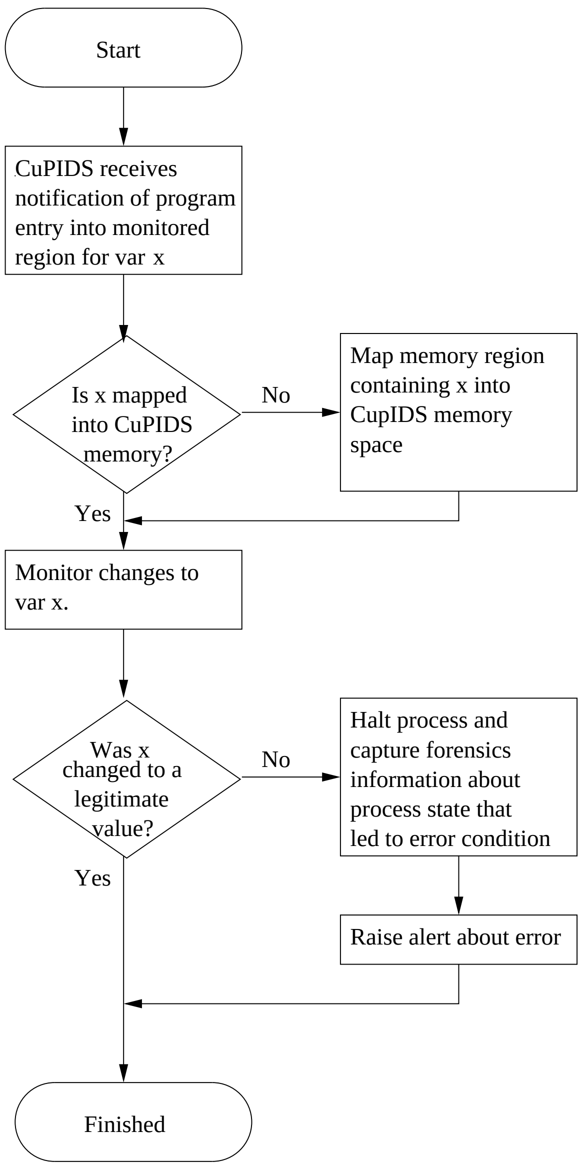
Start
CuPIDS receives notification of program entry into monitored region for var x
Is x mapped into CuPIDS memory?
No
Map memory region containing x into CupIDS memory space
Yes
Monitor changes to var x.
Was x changed to a legitimate value?
No
Halt process and capture forensics information about process state that led to error condition
Yes
Raise alert about error
Finished

Fig. 3. Variable protection flowchart

```
...
CuPIDS_var_create(varID=0,
   var_address = &protected_var);
int protected_var = 42;
CuPIDS_var_access_begin(varID=0,
   var_address = &protected_var);
std::cin >> protected_var;
CuPIDS_var_access_end(varID=0,
   var_address = &protected_var);
...
CuPIDS_var_delete(varID=0);
...
```

Fig. 4. CPP Variable Protection Code

```
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();
...
```

Fig. 5. CSP Variable Protection Code

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.

The CuPIDS architecture requires that when a CuPIDS production process is executing (actually running on a production CPU), that its associated protective process is executing on a CuPIDS CPU.

The process by which a CuPIDS protected process is loaded is illustrated in Figure 6.

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 process establishes any hooks it needs into the production process' memory space and operating environment (wrappers around library and system calls, etc.). 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.

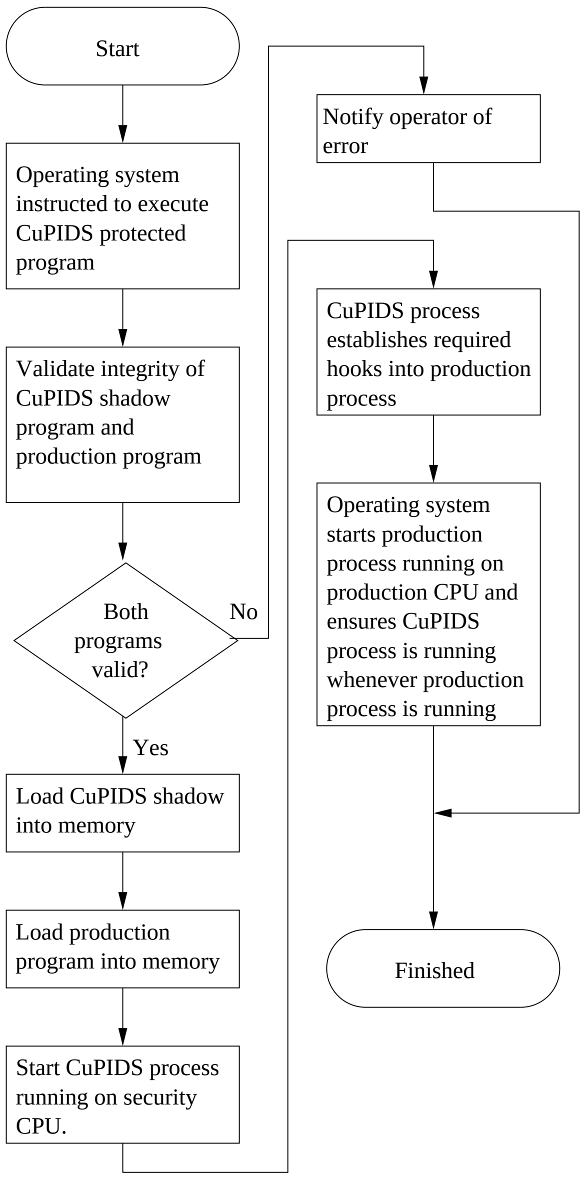
Start
Operating system instructed to execute CuPIDS protected program
Validate integrity of CuPIDS shadow program and production program
Both programs valid?
No
Yes
Load CuPIDS shadow into memory
Load production program into memory
Start CuPIDS process running on security CPU.
Notify operator of error
CuPIDS process establishes required hooks into production process
Operating system starts production process running on production CPU and ensures CuPIDS process is running whenever production process is running
Finished

Fig. 6. Protected process loading flowchart

IV. Implementation

We are testing our research hypothesis by implementing a prototype CuPIDS. This section briefly describes the current state of that prototype. Our experimentation uses FreeBSD, currently 5.3-RELEASE [20]. 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 PP's address space into the address space of a CSP, a means by which signals destined for 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 III, 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 initial experimentation focuses on interactive monitoring in which the CPP programmer defines invariants for key variables, explicitly lists which system resources are used by the CPP, and then exports this information in a form that is usable by the CSP–essentially exporting runtime error checks to 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 syscall 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, etc.

We have used this prototype to verify basic CuPIDS functionality. The system is able to correctly load and execute CPP and CSP components, the CSP is able to detect invariant and security policy violations as well as illegitimate control flow changes. Upon detecting an error, the CSP is able to halt the PP, raise an alarm and capture the state of the CPP's memory. We have not yet performed any timing-related testing.

V. Assessment of CuPIDS' Security Properties

A significant research contribution is an evaluation of how CuPIDS increases security compared to StUPIDS architectures. To answer "why" and "how much," work is underway to define measurable contributions and benefits of CuPIDS' architecture. The experiments described above demonstrate that it is possible for one process to 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 must show that CuPIDS can detect certain attacks faster than can a StUPIDS equipped with a comparable detector set.

VI. Future Work

The architecture described is the initial work to achieve the broad goals of CuPIDS. The prototype is designed to serve as a platform for future research and development in areas of intrusion detection, forensics, and other security related tasks. In addition to completing an implementation of the CuPIDS architecture described in this paper and testing our research hypothesis there are a number of related avenues we intend to explore. These include overall self-protection and healing to enhance fault tolerance as well as support for computer forensics.

A. Self-protection

The growing body of work into mandatory access control (MAC) models such as Biba's integrity-based model [21], and Bell and LaPadula's multi-level security [22] models is 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. To reduce the vulnerability of the IDS to an attack on the production processes, the system components required by the IDS (e.g. the scheduler, critical kernel data structures, etc.) must be protected from modification by an attacker. It may be possible to design an architecture such that the vital components used by the security system are physically separate from the components accessible by production processors; however, this research focuses on logically separating the IDS components in the kernel and using the mandatory access control (MAC) protections available in some modern operating systems to harden those components against attack. It also attempts to ensure that the IDS is given an opportunity to verify the validity of attempts to access or modify those systems components or data structures vital to the IDS.

B. Self-healing

The low-level memory monitoring capabilities provided by CuPIDS may offer interesting self-healing capabilities. For example, the CSP may mark as copy on write (COW) the CPP's links to virtual memory pages the CSP is monitoring. This gives the CSP a capability for recovering from a buffer overflow. When the CPP writes into the COW memory it does so into a new page while the original is preserved by CuPIDS. The CSP can detect the overflow, pause the CPP and replace the corrupted page(s) with its copy of the original data, perhaps filling the buffer with the input but truncated to the correct length. The production application's behavior in this case is as if a safe input methodology were used. Because CuPIDS caught the overflow it can also alarm as well as take preventative actions like modifying the application to protect that buffer, build attack signatures from the data in the overflowed buffer, and pass those signatures out to a supporting infrastructure such as organizational firewalls.

C. Computer Forensics

The purpose of computer forensics is the collection, preservation, analysis and presentation of computer-related evidence. This evidence is used to determine exactly what happened to cause, and who was responsible for, an event in such a way that the results are useful in a legal proceeding. Aspects of the CuPIDS architecture may improve the efficacy of forensics data gathering as well as allow the system to make use of this data to recover from errors and possibly prevent their reoccurrence. The COW capability discussed above may be useful in the aftermath of a computer security incident by providing a record of what the CPP's memory contents (and therefore state) were prior to a damaging attack.

D. Formal Modeling

The majority of modern day systems are far too complex to formally model and verify that specific confidentiality and integrity policies are met. The segmented CuPIDS architecture, however, provides us with a framework to investigate the applicability of existing models, such as the Bell-LaPadula confidentiality model [22], and integrity models such as Lipner's Integrity Matrix [23] and Clark-Wilson [24], to the environment. Benefits of formally modeling the CuPIDS architecture include providing a basis for what security mechanisms should be in place to enforce policies and satisfy design requirements. Using the Bell-LaPadula model, the CuPIDS specific or critical data structures and code can be classified at a higher security level or placed in a separate category than the rest of the system, even those parts of the system running with root privileges. There are many possibilities that can be explored by modeling each component of the underlying architecture. Further research can be done in this area to investigate the possibility of proving the system's security, and in further refining existing models.

VII. Conclusion

We have proposed a paradigm shift in computer security, one that challenges conventional wisdom by trading performance for security. Our approach is based upon running dedicated monitoring functions parallel with the code they monitor on a MP system. 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 into 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.

References

[1] e. a. Mark Crosbie, "Idiot users guide," Tech. Rep. COAST TR 96-04, Department of Computer Sciences, 1996. CSD-TR-96-050.

[2] B. 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.

[3] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Tech. Rep. 99-15, Chalmers Univ., Mar. 2000.

[4] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in c programs," *Softw. Pract. Exper.*, vol. 27, no. 1, pp. 87–110, 1997.

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

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

[7] R. Gopalakrishna, E. H. Spafford, and J. Vitek, "Efficient intrusion detection using automaton inlining," in *To appear in SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, IEEE Computer Society, 2005.

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

[9] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection.," in *IEEE Symposium on Security and Privacy*, pp. 194–, 2004.

[10] J. D. Tygar and B. Yee, "Dyad: A system for using physically secure coprocessors," in *IP Workshop Proceedings*, 1994.

[11] W. Arbaugh, D. Farber, and J. Smith, "A secure and reliable bootstrap architecture," 1997.

[12] O. S. Saydjari, "LOCK: An Historical Perspective," in *Proceedings of the 18th Annual Computer Security Applications Conference, 2000*, (www.acsac.org), pp. Online, www.acsac.org, AC-SAC, 2000.

[13] X. Zhang, L. van Doom, T. Jaeger, R. Perez, and R. Sailer, "Secure coprocessor-based intrusion detection," in *ACM European SIGOPS 2002*, 2002.

[14] 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.)*, vol. 2513 of LNCS, pp. 291–302, 2002.

[15] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[16] 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.

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

[18] I. Tm-I, "Ia-32 intel architecture software developers manual volume 1: Basic architecture."

[19] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts.* John Wiley & Sons, Inc., 2001.

[20] TrustedBSD, "TrustedBSD." www.freebsd.org.

[21] K. Biba, "Integrity considerations for secure computer systems," Tech. Rep. TR-3153, Mitre, Bedford, MA, Apr. 1977.

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

[23] S. Lipner, "Non-discretionary controls for commercial applications," in *Proceedings of IEEE Symposium on Security and Privacy*, (Oakland, CA), pp. 2–10, April 1982.

[24] D. Clark and D. Wilson, "A comparison of commercial and military computer security policies," in *Proceedings of IEEE Symposium on Security and Privacy*, pp. 184–194, IEEE Computer Society Press, 1987.