Micro-speculation, Micro-sandboxing, and Self-Correcting Assertions: Support for Self-Healing Software and Application Communities

PhD Thesis Proposal

Michael E. Locasto
Department of Computer Science
Columbia University
1214 Amsterdam Avenue
Mailcode 0401
New York, NY 10027
+1 212 939 7030
locasto@cs.columbia.edu

December 5, 2005

Abstract

Software faults and vulnerabilities continue to present significant obstacles to achieving reliable and secure software. The critical problem is that systems currently lack the capability to respond intelligently and automatically to attacks – especially attacks that exploit previously unknown vulnerabilities or are delivered by previously unseen inputs. Therefore, the goal of this thesis is to provide an environment where both supervision and automatic remediation can take place. Also provided is a mechanism to guide the supervision environment in detection and repair activities.

This thesis supports the notion of Self-Healing Software by introducing three novel techniques: *micro-sandboxing*, *micro-speculation*, and *self-correcting assertions*. These techniques are combined in a kernel-level emulation framework to speculatively execute code that may contain faults or vulnerabilities and automatically repair such faults or exploited vulnerabilities. The framework, VPUF, introduces the concept of computation as an operating system service by providing control for an array of virtual processors in the Linux kernel (creating the concept of an *endolithic* kernel). This thesis introduces ROAR (Recognize, Orient, Adapt, Respond) as a conceptual workflow for Self-healing Software systems.

This thesis proposal outlines a 17 month program for developing the major components of the proposed system, implementing them on a COTS operating system and programming language, subjecting them to a battery of evaluations for performance and efficacy, and publishing the results. In addition, this proposal looks forward to several areas of follow-on work, including implementing some of the proposed techniques in hardware and leveraging the general kernel-level framework to support Application Communities.

Contents

1 Introduction

In which we reflect on the nature of the problem,
discuss design parameters for our system, and
review related work.

A key problem in computer security is the inability of systems to automatically protect themselves from attack. It is unlikely that system security problems will ever completely disappear. New and creative exploits will emerge to take advantage of mistakes in system design, construction, configuration, and deployment. Since it is difficult, if not impossible, to perceive or predict all threats *a priori*, no system can be completely secure. Furthermore, most exploits are launched with little or no warning, and the window to protect systems against *known* vulnerabilities is shrinking. Symantec reported [55] that the period of time from the announcement of a vulnerability to the appearance of an exploit was about 5.8 days in the first half of 2004. Current estimates are much shorter, and the recent Zotob worm was released 3 days after the vulnerability was announced. The job of security professionals and system administrators is not getting any easier. In the case of worms, malware spreads so quickly as to defy meaningful human intervention. In order to have a reasonable chance at surviving or deflecting an attack, a system must incorporate automatic reaction mechanisms.

This central problem – the inability to automatically mount a reliable, targeted, and adaptive response [35] is magnified when exploits are delivered via previously unseen inputs. Network defense systems (typically composed of IDS's and firewalls) have shortcomings that make it difficult for them to identify and characterize new attacks and respond intelligently to them. These obstacles motivate the argument for placing protection mechanisms closer to the end host (*e.g.*, distributed firewalls [20]). This approach to system security can benefit not only enterprise-level networks, but home users as well. The principle of "defense-in-depth" suggests that traditional perimeter defenses be augmented with host-based protection mechanisms. This thesis advocates one such system to adaptively react to new exploits.

This thesis will seek to answer in the affirmative the question of whether automatic intrusion reaction capabilities can be built into computing systems. Furthermore, it will support the effort to employ useful and *correct* remediation mechanisms for automatically handling exploits. Current techniques (including some of ours) for automatic intrusion reaction and response do not provide provably correct remediation mechanisms.

The major contribution of this thesis is to add a policy-driven layer of indirection to the operating system to intercept and examine the actions of a process before they become "committed" or visible at the OS level. This approach is the basis of techniques like sandboxing, system call interposition [1, 14, 52, 38], and `chroot`-style jails. These approaches differ from my techniques because they only seek to detect or contain the damage, not prevent it from occurring – nor do they provide any way to fix the underlying fault or vulnerability. My approach consists of three novel techniques, *micro-sandboxing*, *micro-speculation*, and *self-correcting assertions*, that are applied to both detection and remediation. This work is particularly useful in the context of an Application Community [29] to support collaborative security. These contributions are enabled by a Virtual CPU Framework (VPUF), a way of treating computation as an operating system service.

1.1 Proposal Organization

The proposal describes two novel components, the *Virtual CPU Framework (VPUF)* and *Self-Correcting* `assert()`*'s (SCA)* that collectively provide a safe environment for automatic monitoring and remediation. The proposal also considers a novel microprocessor architecture called *Speculative Virtual Verification (SVV)* that is meant to provide low-level support to ameliorate the performance impact of our systems, but the actual realization of SVV is largely future work.

In part, this proposal maps out research efforts in the emerging area of host-based reactive security systems. This document elucidates the goals of the thesis, lays out the design space, and discusses the related work leading up to our novel contributions. The systems I propose express one particular vector in this design space. In order to illustrate the trade-offs of employing these systems, I incorporate test criteria and a sketch of how to evaluate the thesis work. The proposal closes with a statement of work that includes a timeline, a list of deliverables, and a plan for building the components, testing the system, publishing the results, and completing the dissertation.

1.1.1 VPUF: A Virtual Processor Framework

In order to enable computer systems to automatically respond to attacks, we need to provide an environment where defensive operations, including remediation, can take place. My current work using STEM for micro-speculation assumes that this environment exists at the application level. For this thesis, I would like to convert STEM[1] into an operating system service, thus enabling the use of micro-speculation at the OS level.

Since a brute force insertion of emulator code into the kernel isn't likely to be maintainable, extensible, or understandable, I propose the notion of *computation as an operating system service*. If a framework existed for providing computation as a service, then managing the insertion or removal of an emulator into or from the kernel would be a fairly straightforward task. No such framework exists. I plan to build one, and the resulting system is more powerful, flexible, and broadly applicable than a simple kernel module containing an emulator.

Most current COTS operating systems treat their set of microprocessors as a static array of entities. When viewing computation as a service, the CPU is no different than any other piece of hardware that the OS uses to satisfy user requests. In particular, the kernel should view the set of available CPUs as a dynamic collection where CPUs are free to join and leave.

While some high-end SMP systems support hot pluggable CPUs, the key idea behind VPUF (besides implementing this capability in mainstream OS's) is that a CPU that registers with the OS need not be hardware: it can be a CPU emulator. In addition, it could include extra monitoring functionality, perform Instruction Set Randomization (ISR), distribute or delegate work to a remote CPU (useful for an Application Community) or even provide a different CPU organization. On the other hand, the piece of software representing a CPU need not *be a CPU* (i.e., it doesn't necessarily need to interpret or emulate the execution of machine code). It could perform other duties, such as DRM, debugging, or data collection for performance tuning. Since this virtual CPU is a piece of software, the implementation possibilities are limited only by imagination. With this Virtual CPU Framework (VPUF), I am advocating changing the notion of execution from the simple concept of "fetech-decode-execute" to flexible programmatic supervision of runtime execution.

[1] More specifically, I will use the general concept of an emulator that supervises program execution for faults and vulnerabilities. I plan on leveraging both STEM and the QEMU emulator for this task.

1.1.2 Automatic Repair: Self-Correcting Assertions

While VPUF provides an environment for micro-sandboxing and micro-speculation, it requires some detection and remediation mechanism to trigger it and direct its actions when a fault occurs. We can potentially use a number of detection mechanisms such as taint tracking, ISR, DYBOC, etc., but we introduce Self-Correcting Assertions (SelCA) as a novel way to accomplish both detection and remediation. SelCA provides a transparent method for policy as well as locations in a program where this policy can be usefully evaluated. The high-level idea is to make the policy consistent at the instrumentation points by changing the relevant program state to match the expected value of the assertion (policy).

SelCA can take advantage of micro-sandboxing and micro-speculation in two ways: speculatively execute the condition for the assert in the sandbox, and if repair is needed, run the repair algorithm in the sandbox as well. Both the condition and the repair algorithm can also be speculated so that if an error occurs, the repair mechanism can fall back to crashing the process instead of applying an invalid fix. Alternatively, the repair algorithm may be run natively for improved performance.

Automating a response strategy is difficult, as it is often unclear what a program should do in response to an error or attack. A response system is forced to anticipate the intent of the programmer, even if that intent was not well expressed or even well-formed. Ideal computing systems would recover from attacks and errors without human intervention. However, the state of the art is far from mature, and most existing response mechanisms are external to the system they protect. Some simply crash the process that was attacked (and do nothing to fix the fault, thereby ensuring that the system is still vulnerable when it is rebooted). Other systems may restrict network connectivity or resource consumption. None provide a fully acceptable response strategy, although the existing tools seem to be useful in server–type software, where continued execution is of paramount concern (as opposed to financial applications, where correctness is the major goal).

My insight is that correctness is only relative to the expression of an idea by a programmer. The programmer is assumed to be able to correctly define how the system should operate, and if they are given the opportunity, they can more completely specify how a program should behave, including appropriate failure semantics. An example of this situation is the Java Exception mechanism. Programmers are forced by the compiler to deal with possible failures of the code they have written as they write it. In major systems programming languages like C, programmers are not forced to explicitly consider or think about the consistency constraints on program and machine state as they write code. There is no opportunity to do so in the language or library, and tools like traditional `assert()` macros are underutilized.

Many fault-tolerant systems and current intrusion response systems seek to completely eliminate the human response from the system, but it is my belief that systems that provide correct failure semantics cannot be constructed without involving a human. All attempts to gloss over errors (like our work on error virtualization, or Rinard's failure-oblivious computing) will not be able to provide correct failure semantics with all applications, although they may work well for certain limited sets of related applications. One new technique that this thesis presents is Smart Error Virtualization, (SEV) an improvement on error virtualization that can learn a correct response at runtime by profiling the application.

I think that we cannot understand or build systems that undertake a completely automatic response without understanding how a human would provide remediation if given the chance. Therefore, I propose placing the human back into the system at the point where they can do the most good – during program specification. An advantage of doing so (besides gaining better semantics) is that the human is still not on the critical path (which is usually the stated goal of most automatic reaction systems) of healing or protecting the system as it is experiencing an attack.

1.1.3 SVV: Speculative Virtual Verification

Since the design of computer architectures is usually performance-driven, hardware often lacks primitives for tasks in which raw speed is not the primary goal. There is little architectural support for monitoring execution at the instruction level, and no mechanisms for assisting an automated response.

To solve this problem, I advocate modifying general-purpose processors to provide both program supervision and automatic response via a policy-driven monitoring mechanism and instruction stream rewriting, respectively. These capabilities form the basis of *speculative virtual verification* (SVV). SVV is a model for the speculative execution of code based on high-level security and safety constraints. SVV introduces architectural enhancements to support this framework, including the ability to supply an automated response by rewriting the instruction stream. In some ways, SVV can be thought of as a firewall or filter for machine instructions.

While I have published [30] work detailing the requirements and organization for SVV, the actual realization of SVV as a deployable technology is a multi-year, multi-disciplinary undertaking and an excellent platform for future research. In particular, I need to develop the components proposed in this thesis in order to have a reasonable basis for implementing SVV or any prototype thereof. I consider SVV to be both motivation for this thesis as well as follow-on work that provides one direction for my post-PhD research plan.

1.2 Design Space

One contribution of this thesis is the mapping out of a design space for host-based reactive security mechanisms. I have already explored some of this design space and have published work dealing with an application-level library that is minimally invasive and acts as an *x86* emulator [47]. I have also modified this emulator to perform Instruction Set Randomization (ISR) and act as a feedback mechanism for an anomaly-based filtering proxy for server applications [31]. This Feedback Learning Intrusion Prevention System (FLIPS) is a realization of the power and flexibility of combining application-level emulation and automatic remediation. This thesis addresses the limitations of an application level emulator by treating the emulator as an operating system service. The work represented by this thesis explores the design space of reactive self-healing software systems and proposes a particular combination that balances the trade-offs of these design space parameters. There are several degrees of freedom to consider.

1. *Host-based vs. Network-based.* Is the system host-centric or does it operate on the network? If at the network, does it operate on the LAN, WAN, or Internet scale, and does it handle encrypted traffic?

2. *Transparency.* How invasive is the system in terms of its impact on the application software?

3. *Support for Legacy Software.* Does the system provide support for and easily integrate with the large legacy base? Can legacy systems be executed unmodified?

4. *Performance.* Although performance may be affected by the system level placement, does the system in general add a performance penalty? If so, where does the penalty manifest, and how large is the impact? What is the trade-off between protection and performance?

5. *System Level.* Is the system placed at a particular level in the system hierarchy or split between two (or more) levels? Possible places to insert the system are raw hardware, the architectural level, the operating system kernel, meta-OS services, a virtual machine monitor (VMM), a user-level library, the application software itself, third-party applications, or remote monitors.

6. *Implementation Difficulty.* Is the system difficult to implement or reverse engineer? Are technical aspects generalizable or are they narrowly applicable and difficult to reproduce? Does the complexity of the system defeat attempts at verification and analysis? How correct is the system? Does the author actually use the system?

7. *Interoperability.* Does the system provide an architecture that is modular? Can it easily interact with similar systems and components written by third parties? Does the system use proprietary communications protocols or does it adopt accepted standards? Is the system useful in general?

We propose a system that is expressed as a particular vector in this space. The system:

- **is a host-based reaction mechanism.** The system focuses on protecting a single host from attack or compromise. The system is not a network-based IDS, but may use information from such systems as part of its sensors and detection mechanisms.

- **is minimally invasive for the application.** The application should not have to be recompiled. The operation of the monitoring and remediation should not be architecturally visible to applications, but may be visible to the OS. The OS *may* provide a means for exporting control or observation of these mechanisms to the application.

- **contains support for the legacy software base.** Legacy applications should be supported. Nevertheless, applications *should* be able to take explicit advantage of any new functionality exported by changes to the underlying systems.

- **is performance sensitive.** The heavy lifting of executing instructions should be placed as close to the hardware as possible. We assume that the bulk of instructions can be executed safely. These "good" instructions should be executed as quickly as possible.

- **is placed at the OS and architectural system levels.** We propose a hybrid system at the hardware and operating system levels, with optional application-level functionality.

- **has a moderate-to-hard implementation difficulty.** In order to obtain a system satisfying the other requirements, implementation difficulty is relatively high and requires substantial design work. We propose fundamental changes to both hardware (optional) and the operating system kernel.

- **provides a modular and flexible architecture.** The system should be able to plug in various signaling, detection, monitoring, and remediation strategies. Furthermore, the system should be able to communicate with other systems (*e.g.*, in an Application Community [29] fashion, although such efforts are only enabled by this thesis and largely remain as follow-on work).

This vector drives the design decisions for the system, although we reserve the right to alter these parameters slightly in response to unforeseen obstacles in implementation or operation.

1.3 Related work

Reactive security mechanisms are an emerging area of research. Intrusion reaction, the design and selection of mechanisms to automatically respond to network attacks, has recently received an amount of attention that rivals its equally difficult sibling intrusion detection. Response systems vary from the low–tech (manually shut down misbehaving machines) to the highly ambitious (on the fly "vaccination", validation, and replacement of infected software). In the middle lies a wide variety of practical techniques, promising technology, and nascent research. The history of this area (and that of the system proposed in this thesis) has its roots in the evolution of both hardware-based security measures and host-based protection techniques. This section briefly considers the aspects of this evolution, including my contributions to the emerging field of automatic intrusion reaction.

1.3.1 Background on Protection Mechanisms

The general area of protecting software systems is a rich area of research, although the realization of systems that react to an attack in an online fashion is only beginning to be explored. Traditional detection mechanisms such as patching, signature-based IDS's, and firewalls aim at addressing previously known vulnerabilities and exploits. Other traditional forms of protection techniques include safe languages [17] and libraries [2], compiler modifications like StackGuard [8, 11], and static code analysis tools [40, 39, 53]. Valgrind [33] is another popular choice for code analysis and debugging. Valgrind has been used by Barrantes *et al.* [4] to implement ISR to protect against code injection attacks. Other work on ISR includes [21], which employs the *x86* emulator Bochs[2].

ISR is a type of aritificial diversity mechanism, and is typically combined with address-space obfuscation [5] and randomization [57] to prevent "jump into libc"–type attacks, even though address-space randomization has its limits [44]. In work inspired by the ideas fundamental to artificial system diversity [13], Holland, Lim, and Seltzer [19] introduce the idea of automatically generating randomized architectures to support system security. Since synthesizing the hardware for such every such generated architecture is an untenable approach, they recommend using VMMs to provide the necessary execution environments.

The contribution of this thesis is a *reactive* approach to system intrusions and faults in which the fault or vulnerability is automatically remediated in an online fashion using micro-sandboxing, micro-speculation, and self-correcting asserts. These mechanisms (and the proposed follow-on work dealing with SVV) draw on ideas from computer architecture, fault-tolerant computing, and computer security. We examined some hardware support [48] for an *x86* emulator (STEM) that supervises the execution of vulnerable code slices [47]. The combination of the micro-sandboxing

[2] `http://bochs.sourceforge.net/`

and micro-speculation approaches is akin to systems [45, 41, 36] that utilize a secondary host machine as a sandbox or instrumented honeypot: work is offloaded to this host, thus minimizing exposure to the primary host. My work on FLIPS is an example of micro-sandboxing that moves the supervision environment (STEM performing ISR) into the protected machine.

1.3.2 Speculative Execution

Speculative execution in the form of micro-speculation is a fundamental contribution of this thesis. In addition, a major follow-on area of work for this thesis is the proposal of a set of architectural components (SVV) that provide a basis for micro-sandboxing and micro-speculation at the processor level by speculatively executing the entire instruction stream.

Two recent efforts make use of speculation in three interesting ways. Work that is closely related to ours is Oplinger and Lam's proposal [34] for using Thread-Level Speculation (TLS) to improve software reliability. Their key idea is to execute an application's monitoring code in parallel with the primary computation and roll back the computation "transaction" depending on the results of the monitoring code. The Pulse system uses OS–level speculation to detect and break deadlocks [25].

Traditional speculative execution is a technique used in microprocessors to execute the instructions in a code branch before the evaluation of the branch conditional is finished. The need to perform speculative execution arises in pipelined processors because the conditional instruction that the branch depends on has proceeded deeply into the pipeline but has not been evaluated by the time the processor is ready to fetch additional instructions. While a complete discussion of the strategies for dealing with branch predication is beyond the scope of this proposal, a basic overview of the subject and pointers to other material are available in [18, 12]. My proposal differs from these techniques by introducing an additional layer of speculative execution in which the acceptance of a particular execution path is not based on the evaluation of a branch conditional, but rather a higher-order constraint on a set of instructions.

Evers *et al.* [12] investigate the predictability of branches and provide an overview of various branch prediction schemes that have been proposed to ameliorate the cost of incorrect predictions. Wang *et al.* [56] explore an interesting result: about 50% of mispredicted branches do not affect correct program behavior. This result is encouraging because it offers evidence that our previously published macro-level remediation technique of *error virtualization* (dynamically returning early from the current function context with an extrapolated error code) holds at the micro-level also.

1.3.3 Secure Hardware

Incorporating security mechanisms in hardware has traditionally been limited to providing implementations of cryptographic algorithms. McGregor and Lee [32] also investigate protecting cryptographic secrets. Of a more focused scope is Lee *et al.*'s proposal [24] of a hardware–based return stack (SRAS) to frustrate buffer overflow attacks. Suh *et al.* [54] propose hardware extensions to thwart control-transfer attacks by tracking "tainted" input data (as identified by the OS). If the processor detects the use of this tainted data as a jump address or an executed instruction, it raises an exception. Kuperman et al. [23] has a good overview of buffer-overflow related attacks and discusses some hardware-based approaches to protection, including SRAS (and related variants) and their own SmashGuard proposal.

Even contemporary approaches to this topic, such as the TCPA/TCG, only provide tamper-resistant hardware modules to store secrets. Recent efforts such as Cerium [7] and XOM [28, 26, 27] focused on providing a trusted computing base (TCB) and tamper-resistant architecture that can attest to the validity of a particular computation [16]. In the case of execute–only memory (XOM), the hardware performs encrypted program execution and makes several strong security claims.

While TCG does offer some measurement functionality [43], the state of the art in this field tries to leverage these stored secrets for attestation, and attestation is typically used for the purposes of DRM. Such uses provide a mechanism for a remote entity to control local execution. There are no mechanisms for the local entity to systematically prevent and control a remote entity from executing local code. My work on SVV is an attempt to provide a unified model for the supervision and online patching of machine instructions.

The Copilot system [37] by Petroni et al. is one expression of hardware security aimed at integrity protection. Much like the Tripwire[3] software, the goal of Copilot is to make sure that important data has not been corrupted. Copilot performs rootkit intrusion detection by monitoring changes to a host's kernel text segment and related data structures. The current implementation is based on a PCI card that monitors the host's main memory via DMA (without the host kernel's knowledge) and has a secure communications link to an administrative reporting station.

1.3.4 Execution Supervision Environments

Virtual machine emulation of operating systems or processor architectures to provide a sandboxed environment is an active area of research. Virtual machine monitors (VMMs) like Xen [3] are employed in a number of security–related contexts, from autonomic patching of vulnerabilities [46] to intrusion detection [15]. MiSFIT [50] is a tool that constructs a sandbox by instrumenting applications at the assembly language level. Program shepherding [22] works on uninstrumented IA-32 binaries and validates branch instructions to prevent transfer of control to injected code. Intel's Vanderpool and AMD's Pacifica designs are forward-looking architectures that provide support for hypervisors and VMMs.

1.3.5 Recovery and Repair

Effective remediation strategies remain a challenge. The typical response of protection mechanisms has traditionally been to terminate the attacked process. This approach is unappealing for a variety of reasons; to wit, the loss of accumlated state is an overarching concern. Several other approaches are possible, including failure oblivious computing [42], STEM's error virtualization [47], DIRA's rollback of memory updates [51], crash-only software [6], and data structure repair [9]. Remediation strategies sometimes include the deployment of firewall rules that block malicious input. The most common form of this strategy is based on dropping packets from "malicious" hosts. Even with whitelists to counter spoofing, this strategy is too coarse a mechanism. My system FLIPS [31] takes advantage of STEM's error virtualization and also allows for the generation of very precise signatures because the actual exploit code can be caught "in the act."

[3] `http://tripwire.org/`

2 VPUF: A Virtual CPU Framework

In which we introduce the notion of a virtual CPU framework inside the Linux kernel and consider its scope and impact.

2.1 Introduction

The core problem addressed by this thesis is the lack of an automatic repair mechanism for COTS software. The solution to this problem is the ability to repair faults and an environment in which to perform repair activities. The Virtual CPU Framework (VPUF) provides the environment for this *micro-sandboxing* and is a complementary approach to Virtual Machine Monitors (VMM's). VPUF is a novel and unique operating system organization that can be leveraged for far more than the self-healing capabilities examined in this thesis.

2.2 Operating Systems Organization

The role of most operating systems is to behave as a resource arbiter. A group of processes (representing programs in execution) request resource units, and the operating system grants or denies access to these resources according to some policy. Resources include both time-multiplexed (e.g., CPU) and space-multiplexed (e.g., memory, disk) entities. Processes are scheduled for execution by the OS on the bare hardware of a machine (or suitable abstraction thereof). This arrangement does not allow for fine-grained monitoring of the process, as the effects of process instructions become reality once they progress through the CPU and become architecturally visible.

In order to allow the OS to closely monitor processes for faults or exploited vulnerabilities, each process needs to be executed in a confined environment that can be closely inspected. I therefore introduce the notion of micro-sandboxing, a lightweight complement to other sandboxing or isolation techniques like VMM's. The microsandbox is an emulation environment that allows the system to intercept and monitor instructions as they are executed. The combination of micro-sandboxing and micro-speculation enable the microsandbox to safely speculatively execute instructions and observe the results, undoing them when they are deemed unsafe.

Modern COTS kernels fall into three basic categories: (1) monolithic, (2) micro-kernel, and (3) exo-kernel. Monolithic kernels define a broad interface between applications and the hardware by providing a series of entry points that are collectively known as the system call interface. The components of a monolithic kernel are tightly integrated and run in the same address space. In contrast, a micro-kernel attempts to separate as much as possible the different functional modules of a kernel and support message-passing and other forms of IPC between them and the broader set of OS services that are implemented as user-space programs. The exo-kernel approach attempts to eliminate as much code as possible between application programs and the underlying hardware; OS services are typically provided by components called "library operating systems."

2.3 An *endolithic* Kernel Design

VPUF is a technique that is most easily applied to monolithic kernels. In contrast to a VMM, which provides a hardware abstraction in which multiple OS's can be hosted (guest OS's), VPUF adopts what I term an *endolithic* kernel ("endo-" meaning *within* and "-lithic" referring to the tight integration of its components with the primary kernel machinery). An endolithic kernel provides multiple execution engines within the kernel itself. In an endolithic kernel, a CPU is no different than any other piece of hardware that the OS uses to satisfy user requests. Whereas VMM's host multiple OS's on a single hypervisor, an endolithic kernel "hosts" multiple processes on multiple virtual CPU's within the operating system itself. A process executes normally until it requests (or is placed by some monitor in response to a signal) to be scheduled on a virtual CPU. Signals may include alerts or alarms from intrusion detection systems.

The design of VPUF changes the kernel to allow any number of software CPUs (virtual processors) to be added as loadable kernel modules. Once loaded, a new CPU will show up in `/proc/cpuinfo` and be treated much the same as any other processor that the kernel knows about. Processes can be scheduled for execution on these virtual processors. The execution of the actual code for these virtual CPUs would take place inside kernel memory and would be kernel code (i.e., code in supervisor mode) most likely run on a physical CPU (although interesting future work would allow any virtual CPU to run on any other virtual CPU). For the purposes of this thesis, an `x86` emulator with monitoring functionality will be the virtual CPU, although other interesting possibilities exist. One key advantage of the endolithic approach is that supervision of programs can extend to the kernel itself. Our current emulator does not follow execution into the kernel; when a system call is invoked, STEM relinquishes control to the kernel, temporarily ending supervision and protection until the system call returns.

2.4 System Requirements and Components

The kernel, if it doesn't currently support this ability, needs to treat processors like any other piece of hardware that can be inserted or removed at will. The kernel must be able to register a number of virtual CPUs, so mechanisms for maintaining this collection should be added. In addition, the kernel should:

- define a namespace of virtual CPU types. Examples include *x86*, *x86-stem*, *x86-svv*, *powerpc*, and *debugger*.
- contain a scheduler that supports the notion of multiple virtual run queues for each virtual CPU in the system.
- be able to stop a process running on a real CPU and transfer it from the regular run queues to the run queue of the appropriate virtual processor.
- support switching to the execution of the virtual CPU code itself in another kernel thread. The kernel needs to handle returning from this thread to resume normal execution of processes.
- handle a system call from within a kernel thread[4].
- contain a signaling framework to handle beginning and ending virtual execution.
- handle signals and exceptions raised by the virtual CPU.

[4]When the virtualized process makes a system call, it may be a good time for the kernel to switch back to the regular scheduler.

In order to support easy maintenance, any virtual CPU should be written as a Loadable Kernel Module. The changes to the kernel to make it endolithic should provide enough infrastructure that writing a virtual CPU to be installed as an LKM would be straightforward.

There are four promising approaches to providing a begin/end signaling mechanism for virtual execution. First, we can violate transparency and insert code into applications that explicitly signals the OS (via a new system call) to enter this mode of execution. Second, we can instruct the kernel to always execute a process on the virtual processor. Third, we can use a heuristic or outside signal (perhaps from an IDS) to automatically wrap certain “suspect” code sections. Finally, we can use predetermined ranges of instruction addresses (such as might be negotiated by an Application Community).

2.5 Conclusions and Future Work

VPUF introduces the notion of computation as an operating system service and will be implemented as an endolithic kernel – a novel kernel architecture. This framework provides the basis for hosting our software CPUs for closer supervision and monitoring. The framework also allows for other uses, as the virtual CPU doesn’t have to be an emulator for the underlying hardware, or even a CPU for another architecture. The virtual CPU could be a ’stub’ processor for sending instructions off to a remote machine for supervised processing. Such a facility would be useful in an Application Community (AC) where security checks are outsourced to other nodes in the AC. AC nodes may participate in a distributed bidding process to cover different slices of an application, and one node may ’pay’ another node to incur the performance cost of extra monitoring. This technique supports a robust management architecture for AC’s where a small subset of nodes can be asked to investigate suspicious behavior from other nodes in the AC.

It is of value to port VPUF from Linux to other operating systems such as OpenBSD or Solaris 10. We can also investigate creating a stub processor to support Application Communities, a debugging CPU that heavily instruments the code at runtime (similar to Valgrind), or even a CPU for a different hardware organization like PowerPC or Itanium. Other virtual CPU implementations may do more exotic operations on the supervised instruction stream, such as performing proofs on it. Additional work includes implementing an SVV emulator and supplying compiler provided or runtime learned ’alternate’ code blocks for automatic recovery.

3 Detection and Remediation Policy: Self-Correcting Assertions

In which we propose a basis for automatically correcting faults.

3.1 Introduction

Automatic diagnosis and repair of exploited vulnerabilities and the software faults underlying them is a new area of research. Our work on *error virtualization* [47] and Demsky and Rinard's automatic data structure repair [10, 9] are leading efforts in this field. In this thesis, I propose another method, Self-Correcting Assertions (SelCA), of supplying automatic response capabilities. While SelCA shares many of the same aims as data structure repair, there are a number of important differences.

SelCA is a more general approach than data structure repair; it is concerned with more general program constraints than just consistency on data structures. In addition, it utilizes a built-in feature of popular systems programming languages and is largely transparent. Third, since it takes advantage of already available programming language features, it does not need to invent a syntax or constraint language like that of the data structure repair technique. Finally, SelCA can also take advantage of Smart Error Virtualization (SEV), a way of learning a correct response at runtime.

3.2 Design of Self-Correcting Asserts

The motivating idea of SelCA is to overload the current use of `assert()` macro calls in C and Java. The semantics of assert statements are to evaluate a given condition and abort the process if the condition fails. Thus, assert statements are primarily used for notification and fault detection. I propose changing the semantics of assert statements to both notify and repair. If a condition fails, then the assert should attempt to adjust the values of the relevant program state in the condition to make the asserted expression true. A number of basic reaction strategies are possible:

1. *fix*: invoke auto-repair adjustment of expression variables
2. *return*: return from function with an error code; equivalent to error virtualization
3. *sev*: Smart Error Virtualization (SEV): return from function with a learned error code
4. *break*: break from current scope (mini-error virtualization)
5. *continue*: ignore error and move on, a rough version of failure oblivious computing

In addition, the system designer can extend this set of strategies by defining a new sequence of repair code, giving it a valid reaction strategy name, and linking it with program constraints.

Changing Semantics Since current assert statements most likely exist as a last-resort mechanism to forcibly crash the program if the assert fails, changing their semantics by directly overloading them may not be the best solution. However, they still present a valuable opportunity because a programmer felt that the particular constraint was important enough to remain inviolate – thus, the expression deals with state that is critical to the continued correct operation of the program. I plan to examine a number of open source applications to detail the nature of exactly how assert statements are currently used and what the complexity is of the expressions they evaluate. This knowledge will be useful in deciding whether or not assert statements should be redefined for a particular type of program.

In the meanwhile, it would be useful to construct a parallel mechanism that does not change the semantics of existing assert statements, but rather performs the self-correcting work in a micro-speculated environment. This mechanism can be invoked *before* a call to a regular assert statement, thereby correcting the expression before it reaches the regular assert statement and allowing the regular assert statement to retain its operational semantics. In addition, we can broaden our focus from simple assert statements to thinking about the general state of an application. Constraints on program state are relatively ill-managed – there are no language constructs for doing so besides those that the programmer creates and uses himself. We propose the use of named assertions in groups of assertions called *cassert contexts* such as that shown in Figure 1.

```
/* Define a cassert context 'state' type. */
cassert_context state
{
  int __a__; //binds to integer 'a' in the evaluation context
  int __b__; //binds to integer 'b' in the evaluation context
  condition mycondition1  : (__a__<__b__);
  /* possible reactions are: 'fix', 'return', 'break', 'continue' */
  mycondition1.reaction = fix; //implies 'continue' after fix
};

/* Declare an instance of this cassert context state type. */
cassert_context state program_state;

/* Declare variables that those in the cassert context can bind to */
int a = 7;
int b = 2;

/* evaulate and repair conditions defined by 'program_state' */
cassert(program_state);

/* The regular assert statement should now always succeed. */
assert(a<b);
```

Figure 1: **Conceptual Example of Managing Constraints and Responses.** *Each cassert context defines a number of constraints on program execution and associates a repair activity with the constraint. Repair activities may include four basic strategies (new ones are future work). These strategies are fix (the essence of SelCA), return (return from current function with a heuristically determined value), sev (Smart Error Virtualization), break (break from current scope), and continue (ignore errors and forge ahead). These are default names for fixing strategies, others could be written by the programmer or generated by the compiler.*

Managed Program State: Scoped Repair Policy If we can invent a notion of a structure that contains relations between pieces of program state (variables and data structures) and link that relationship with a series of repair actions, then we can provide a general policy for directing automatic remediation and repair when those constraints are violated. Thus, calls to a correcting assert (`cassert()`) function are really just instrumentation points at which the policy for the current program scope is evaluated, and repair actions take place if the policy is violated.

These instrumentation points could be automatically inserted into a program's instruction stream or source code when entering or leaving a new scope (such as a function call or new looping or decision statement). The combination of all current assert conditions in a particular scope is called the *cassert context*. All these conditions can be evaluated (along with the primary condition) to provide a measure for how closely the application is matching the expected range of execution.

In addition, the instrumentation points provide a convenient place to perform checkpointing in preparation for micro-speculation.

Speculative Execution Calls to traditional assert macros actually evaluate the expression that is supplied to the assert. The assert itself only checks the final value of this expression. This arrangement can actually be problematic if the condition has side effects – traditional assert statements can actually be the cause of difficult to track bugs which disappear when debugging code is enabled[5].

It would be useful for automatic repair if the expression for a self-correcting assert could be speculatively executed and any side effects undone. When a cassert context is evaluated at an instrumentation point, execution of the process can switch to a microsandbox performing micro-speculation. If the expression fails, then recovery actions can take place and the side effects of the failed expression undone.

While the repair code and algorithm could be manually specified, it would be most useful if the repair code were automatically supplied by calls to a library inserted by a programmer or compiler. As part of my research efforts, I plan to examine some open-source applications and manually provide repair code for each assert that is encountered. I will then compare these efforts to the use of source code transformation tools to automatically generate the repair algorithm.

I propose implementing this system for two programming languages: Java and C. The two implementations will demonstrate the applicability of SelCA at the application level and the compiler level. The Java implementation will be a library component that a programmer can invoke directly. The C component will be provided as an additional stage of compilation that employs Antlr to transform the source code of the application automatically whenever it encounters an assert. Like the current assert implementation, these implementations will provide a flag for disabling their operation.

3.3 Implementation Details

Regardless of the language SelCA is implemented for or whether it is inserted by the programmer or compiler, the core library must provide an algorithm for evaluating a program expression at runtime, running a Satisfiability algorithm, and adjusting the values of program state to correct the expression if it is not satisfied. While the general algorithm for determining satisfiability is NP-Complete (via reduction to 3SAT), the good news is that this problem is actually decidable! Furthermore, I plan to show that a large portion of applications do not use very complicated expressions, and so the time for determining a satisfiable assignment of values to the expression atoms is short. The satisfiability algorithm will engage in a brute force generation of the truth table for the expression. It will then collect the set of satisfiable assignments and select one that is the closest to the current assignment. For the purposes of this thesis, "closest" will mean changing the fewest atoms. The definition of "closest" beyond this notion is interesting future work, especially as atoms can be weighted (e.g., it may be relatively inexpensive to change the value of a primitive variable as opposed to creating a new struct type).

Two challenges remain: obtaining the atoms so that the satisfiability algorithm can run, and actually adjusting the values if the condition is violated. In order to capture the expression parts

[5]From the `assert` man page: "assert() is implemented as a macro; if the expression tested has side-effects, program behaviour will be different depending on whether NDEBUG is defined. This may create Heisenbugs which go away when debugging is turned on."

for our automatic analysis (remember, regular assert statements are really just macros that test the final value of a series of program instructions), we introduce a data structure capable of storing and describing any legal program expression. This data structure can be generated and populated by either the programmer or compiler. In the case of the compiler, we augment compilation with a pre-processing phase that parses the expression passed to the assert and stores the relevant information. The programmer could do this procedure manually. We define a library that the parser or programmer can use. Figure 2 provides an example of the code that would be generated using code from this library for the runtime functionality.

```
import edu.columbia.cs.nsl.scalib;

public class Test
{
  public void run()
  {
    int a = 7;
    int b = 2;
    SCAssertion myAssert = new SCAssertion();

    /* Two ways to build expressions. */
    myAssert.setExpression("a<b");
    myAssert.compileExpression();
    // -- or --
    myAssert.startExpression();
    myAssert.buildExpression(Assertion.LEFT_PAREN);
    myAssert.buildExpression(Assertion.namedVariable("a", a));
    myAssert.buildExpression(Assertion.LESS_THAN);
    myAssert.buildExpression(Assertion.namedVariable("b", b));
    myAssert.buildExpression(Assertion.RIGHT_PAREN);
    myAssert.endExpression();
    myAssert.compileExpression();

    /* Call the correcting-assert method.*/
    SCAssertEngine.cassert(myAssert);

   /* Can also provide regular assert functionality. This call
    * should always succeed if invoked immediately after the
    * cassert() method on the same assertion object.
    */
    SCAssertEngine.assert(myAssert);
    assert(a < b);
  }
}
```

Figure 2: **Example of Generated SelCA Code.** *This example demonstrates code that would be produced by the compiler or programmer using the Java version of the SelCA library. Java can provide another way (just a string) to build expressions because it contains facilities for runtime reflection of program code.*

All that remains is the actual implementation of the fixing strategy. The strategy consists of a number of smaller sub-strategies. Once particular expression atoms that need to be changed are identified, the atom can be treated in a number of ways according to its type and the types of its operands. Atoms can be unary, binary, or ternary operators and their operands (which can potentially include function calls) and can express logical, relational, or arithmetic operations. For example, if the atom $a < b$ needs to be true for the assertion to succeed, then we can decrement the value of a until it is less than b or simply set a to the value $b - 1$. If the expression is $a == b$, then we can set the values equal. An expression like $mystruct! = NULL$ can be fixed by creating a new struct of the appropriate type (this illustrates the importance of knowing the type information of the operands). If atom operands include function calls, the ability to micro-speculate helps out tremendously by providing an opportunity to checkpoint program state. Finally, valid values for atom operands can be learned by observing previous executions of this program instance, past program instances, or remote instances (input from an Application Community).

3.4 Conclusions and Future Work

Automatic remediation is a hard problem. Often, detection mechanisms are not perfect, and initiating an automated response based on a false positive is an undesirable event. However, most remediation strategies are quite primitive. Basic but widespread techniques include manually removing a machine or application from service, simply crashing an application, or deploying firewall rules to prevent traffic from reaching a networked service. These mechanisms are quite coarse, result in self-induced DoS, and do not offer any sort of robust or continued service in the face of a recurring attack, nor do they address the actual fault or vulnerability. Error virtualization and failure-oblivious computing are slightly more elegant attempts to continue service and avoid newly detected faults.

SelCA provides a tool for automatically specifying the semantics of failure recovery so that when a constraint is violated, repair activities can take place as quickly and correctly as possible. SelCA attempts to constrain the execution of automatic remediation in a customizable way. Although SelCA is an important step forward, there remains a great deal of future work. One important problem is rigorously proving that a particular response is correct. An example of this problem is the following code snippet:

```
assert(num_sorted < 10);
```

where `num_sorted` depends on a higher-level property of some program data structure that needs to be algorithmically determined. Computability theory indicates that this proof cannot be constructed in general, but there may be specific instances were a remediation strategy can be proved appropriate. For this particular example, simply changing the value of `num_sorted` to be 9 isn't likely to be useful, as the *real* state isn't changed, just a representative quantity or *state delegate*. We would need to link the contents of the tested data item (the state delegate) with the results of a particular code fragment (say, the routine that sorts the related data collection). In causing `num_sorted` to acquire a new value, we would also have to modify the data collection itself.

Future work includes an evaluation of the main SelCA ideas for the paper I am preparing for USENIX ATC 06. Finally, although SelCA is inspired by a specific programming language construct (i.e., `assert()` statements), the idea of explicitly managed program state combined with repair actions is applicable to systems in general. Such a state management policy could even be written in a policy language much like the Java 2 Policy mechanism and enforced by the runtime system independently of the source code for the application.

4 SVV: Speculative Virtual Verification

In which we consider a new pipeline organization for automatically responding to attacks.

4.1 Introduction

I have proposed work (SVV) [30] for modifying hardware to include policy-based monitoring functionality in conjunction with a higher-level software supervision framework. SVV is motivated by earlier joint work [48, 49] that recognizes the limitations of an application-level emulator.

Static analysis techniques or improved programming practices alone are unlikely to provide a complete solution to the types of errors that threaten system stability or create exploitable vulnerabilities. Even systems that dynamically monitor process execution often impose a noticeable performance cost. Furthermore, these systems may reinvent the same primitives because the hardware does not supply them. However, even if such capabilities existed, system security is often a matter of *policy*; these utilities would need some level of flexibility to be applicable and remain useful in a wide variety of diverse and evolving environments. Finally, systems currently lack the capability to respond intelligently to both attacks and non-malicious faults.

The main contribution of SVV is the proposal of a set of architectural components that provide a basis for such systems by speculatively executing the entire instruction stream. In much the same way that a superscalar processor speculatively executes past a branch instruction and discards the mis-predicted code path, we propose that processors operate on the instruction stream in two phases. The first phase executes instructions, optimistically "speculating" that the results of these computations are benign. The second phase makes the effects of the speculated instruction stream visible to the OS and application software layers and potentially rewrites the instruction stream if it has been deemed harmful.

4.2 Motivation and Feasibility

SVV is motivated by work on constructing an emulator [47] to supervise program execution in response to exploits and errors. Unfortunately, the use of an emulator imposes a considerable performance overhead since the emulator executes every program instruction in software. The first way to ease this burden, which was adopted in [47], is to limit the scope of emulation to portions of the program demonstrated to be vulnerable, thereby reducing the time that is spent in the emulator. The second approach is to eliminate the emulation penalty altogether by executing the process directly on the CPU. Adopting this approach currently means relinquishing the monitoring capabilities that the emulator provides. Therefore, I advocate adding monitoring mechanisms to processors so that a certain level of safety is relatively inexpensive. In order to address more complex attacks, I also propose that execution can be delegated to the software emulator as needed.

The goal is to push common-mode security monitoring functionality further down the system stack. Arguing for the widespread adoption of fundamental changes to hardware is a controversial proposition. I believe the hardware necessary to support the system is easily implementable. Indeed, large parts of the system are already present in modern processors to support thread level speculation (TLS). The design parameters of general-purpose microprocessors have traditionally been driven by raw performance. SVV advocates design parameters aimed at more high-level feature support.

4.3 The Design of SVV

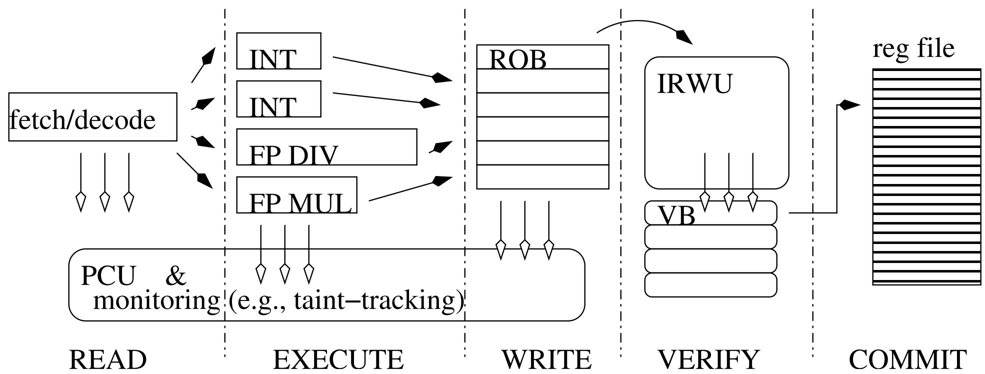

Figure 3: **Pipeline organization for SVV**. *Here, a simplified pipeline for a superscalar processor is modified to add an extra verification stage as well as policy-driven hardware-based monitoring mechanisms. The IRWU can optionally rewrite the instruction stream and cause the new version (stored in the VB) to be executed. Traditional hardware components are shown as full rectangles, new components are rounded. Not shown is the VERU, which holds the address for an emulator capable of higher-level supervision.*

As illustrated in Figure 3 the core features of SVV form a two level monitoring environment. The first level includes hardware mechanisms for monitoring instruction execution (bounds checking, taint-tracking [54], SRAS [24], transfer control validation [22], *etc.*). The second level of monitoring is provided by the Policy Constraint Unit (PCU) and the Virtual Emulator Registration Unit (VERU). Instructions are filtered by the PCU according to some policy constructed by the programmer, compiler, or runtime profiling. The policy could range from filtering on a particular class of instructions (integer vs. load/store) to more complex constraints that require keeping state. The design of a constraint language to express these policies is future work, but we envision the PCU to be a FSM much like the instruction decoding unit that is able to filter instructions based on properties like target and source registers and memory locations, instruction type, and processor status flags, other processor state (as supplied by other components such as a SRAS or an array length tracker), and data dependencies.

The VERU stores an address for code that should be executed if the PCU identifies a sequence of instructions that require more resources than the hardware can easily provide. Finally, the Verification Buffer (VB) and the Instruction Rewrite Unit (IRWU) provide some basic support for an automatic response capability.

4.3.1 SVV Execution Model

The execution model for SVV (see Figure 3) is similar to current superscalar execution models. Instructions are fetched, decoded, issued to functional units (possibly out of order), executed, and gathered in a re-order buffer (ROB) to be committed in program order. However, at each stage, instructions are filtered by the PCU and monitored by hardware-level security mechanisms. Additionally, the VB accumulates completed instructions as they leave the ROB and commits them only if they pass the monitoring tests.

Instruction flow for SVV can be categorized by the following three scenarios. First, the instruction may be harmless. In this case, it proceeds normally to the ROB, graduates when appropriate, moves to the VB, and is committed. Second, an instruction may be harmful as determined by the monitoring mechanisms (*e.g.*, it is actually tainted input data, or will write input data to the code area of the process address space) or the PCU. In this case, the IRWU flushes the scope of the harmful instruction and constructs a 'safe' version of the flushed code. The processor then executes this alternate instruction stream, including a return to the normal path of execution. The third scenario enables an emulator to be loaded on the CPU and supervise code execution. If the PCU decides that a particular sequence of instructions requires more complex supervision, it can invoke execution of this emulator. Note that there is no requirement for the software invoked by the VERU to be an emulator. The VERU simply holds an address and transfers control to the code at this address. Such an approach enables a more general response mechanism than software emulation. For example, the VERU may transfer control to an OS routine that kills the process, or suspends the process and transfers it to an isolated host for analysis, auditing, intrusion detection, or debugging.

Another way to envision the SVV execution model is as an operating system that schedules a process for execution on two cooperating microprocessors, as shown in Figure 4. This type of organization provides more motivation for a framework such as VPUF to avoid needlessly complicating the OS.

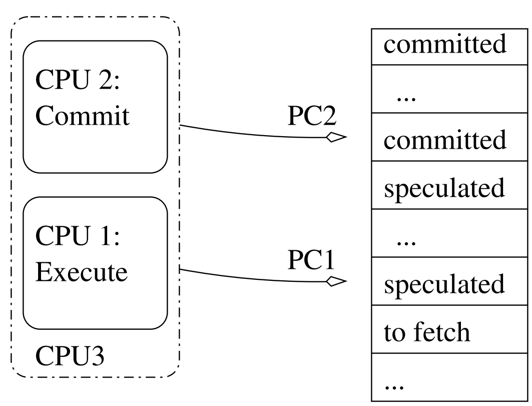

Figure 4: **High-level execution model for SVV**. *The instruction stream for a process is scheduled for execution on two processors. CPU1 supervises instruction execution while CPU2 commits instructions that are benign. CPU2 can optionally re-write the instruction stream as a basic form of active response. The conceptual processors CPU1 and CPU2 are actually one physical unit, CPU3.*

4.3.2 Scope of SVV

The largest obstacle to overcome for SVV is a three part problem that involves determining the scope of supervision. First, even though SVV is meant to run continuously, some applications (especially those working in a power-constrained environment) may wish to avoid the overhead associated with constant monitoring. Second, hardware is fundamentally limited in the number of virtual execution contexts it can support concurrently. Finally, it is likely that the basic monitoring mechanisms, while capable of stopping large classes of attacks, may be unable to cope with more sophisticated attacks (some forms of DoS, multi-step attacks, information leaks, improperly set permissions, phishing attacks, *etc.*) or analysis tasks that require copious amount of state (anomaly or intrusion detection via data mining).

To address the latter two problems, we use the VERU to register a software emulator that can perform high-level monitoring of an instruction stream. An emulator has the flexibility to be more intrusive and is easily customizable. This hybrid approach to monitoring is more promising than an approach based solely on hardware or software. To address the first problem, SVV can be selectively invoked. Control over this invocation can be handled by the OS (a new system call to invoke or halt the SVV hardware) or the compiler (new assembly instructions can delimit an SVV monitored code region).

4.3.3 Micro-patching: Automated Response

SVV includes the ability to rewrite a vulnerable sequence of instructions without recompilation. In effect, SVV supports the ability to generate and insert a micro-patch into the protected application's instruction stream. This mechanism is general enough that a wide variety of response techniques can be implemented, including self-correcting asserts, data structure repair [9], failure oblivious computing [42], and error virtualization [47]. Compilers can be augmented to provide "alternative execution paths" to some code sections. These alternatives can be driven by explicit program code, programmer annotation, purely compiler-generated, taken from profiling information for the application, or gathered by the processor itself from previous runs of the same code block as a form of machine learning.

The rewritten instruction stream can be propagated to the code section of the process address space to protect future execution. The new instruction sequence could be applied (with OS support) to the on-disk binary as a rudimentary patch. The question of whether or not to propagate the micro-patch out to the process memory space or even to disk is a high–level policy question. One difficulty with automatically propagating the patch (beyond the current invocation) is that attacks and faults are relatively rare, and executing the micro-patch for all subsequent normal requests would needlessly change the normal operation of the software. One solution is to have a prologue to all micro-patches such that they are conditionally executed based on site policy (as set by an administrator who knows the needs of the environment). Another solution is to have the micro-patch conditionally executed based on markers seen in the environment. For example, at the moment of patching, a software-level monitor can take a snapshot of important state (network packets seen, important data structures), and if those conditions are recreated, the monitor can set a flag so that the micro-patch does execute.

Micro-patching via instruction stream rewriting can be seen as a type of automatic diversity mechanism. While automated diversity is a good protection mechanism, we argue that micro-patches should be recorded somewhere (even if they are not automatically propagated to the process image or binary); failing to do so can make it difficult to debug an application, as there would be no exact record of what code the processor generated and executed.

There are many pitfalls to automating a response. One interesting possibility is for an attacker to implement a covert channel by continuously causing SVV to flush the current set of instructions and replace it with a micro-patch. Such an attack would seem to be difficult, as the current execution context (and thus, presumably the attacker's code) would be replaced with completely different instructions, but it not at all outside the realm of possibility. The micro-patch itself would have to cause an externally measurable phenomena for the consumer of the covert channel.

4.4 Future Work

Most of the core mechanisms of SVV remain future work. VPUF is an attempt to provide an environment within the operating system that can host an emulator that implements SVV. Given such an environment and software implementation of SVV, the core ideas can be validated for expression in hardware.

Future work for SVV itself includes augmenting the current set of monitoring and detection mechanisms and supplying additional remediation mechanisms (such as SelCA). There remain a multitude of challenging problems to be addressed in the construction, testing, and deployment of SVV. After finishing the work proposed in this thesis, I intend to study these issues and implement SVV in a variety of execution environments, including *x86* emulators, the Java Virtual Machine, and simulators for the MIPS and ARM architectures.

4.5 Conclusions

This section described the architectural components needed to support a new execution model for secure and reliable computing: *speculative virtual verification* (SVV). This model complements previous work on trustworthy and tamper-resistant computing architectures but is not meant as a replacement for the capabilities such systems provide.

There is no silver bullet for system security, and SVV is not meant to address all possible attacks. However, given the current state of the arms race between attackers and system designers, a paradigm shift is necessary. I advocate modifying general-purpose processors to (a) provide implicit supervision functionality, (b) export a policy-driven monitoring mechanism, and (c) provide the foundation for an automatic response capability via instruction stream rewriting.

5 Research Plan

The research plan covers a twenty-six month period from April 2005 to May 2007. The remaining 17 months represent the research and development of VPUF and SelCA. The plan and statement of work establish criteria for evaluation of these components and a timeline in which to accomplish the work. This proposal shows that I have accumulated a base of work and contributed to an emerging field. It should also show that the work I propose to do as an extension to this base is novel, interesting, and applicable to the key problems. Finally, it shows that I have laid groundwork for several interesting research efforts post-PhD.

5.1 Statement of Work

This statement of work details the deliverables and specific tasks by which progress toward the PhD will be measured. Each of the thesis parts above can have their own continuing evaluation, especially against new attacks. One of the major challenges for evaluation is obtaining a malware collection to test if VPUF and SelCA are effective against real attacks. The major items in the SOW are listed below. Starred items (*) are not crucial or promised.

5.1.1 Tasks for Self-Correcting Assertions

- set up CVS repository for SelCA
- write design document for SelCA
- identify open-source applications to survey for assert() statistics and usage
- gather data on how many asserts are present in a given application
- gather data supporting the hypothesis that the conditions in assert statements are relatively simple. Identify cases where this hypothesis does not hold.
- manual changes to the selected set of applications to perform the following mutually exclusive operations in order to glean some qualitative information about asserts that is not indicated by the quantitative data of the previous tasks:
  - force asserts to succeed, run through 'make test'
  - insert calls to cassert() with manual fix code for every instance of an assert
- create a Java package the provides an application-level implementation of SelCA, including an object that represents an assert, an assert expression, and an engine that can evaluate the assert expression object and repair it if need be
- create an Antlr-based parser that:
  - recognizes calls to `assert()`. For each assert, generate a parallel correcting assert function `cassert()`
  - parses the expression supplied to the assert

  - allocates and builds a data structure representing that expression
  - generates a function named `cassert_ZZZZ` where ZZZZ serves to differentiate between all instances of cassert statements in the program. Save this function in a file with other generated cassert() functions to be compiled with the rest of the application
  - insert call to new cassert immediately before the assert call that triggered this process
  - the generated function should check the value of 2 flags: one called CASSRT_ON that is a global switch for turning all cassert() functions on or off, and another that is function-specific (e.g., `cassert_table[cassert_function_id]`)

- *learn from manually generated fixing code what the best strategies for actual repair may be

- integrate calls to the micro-sandbox environment by using the endolithic kernel (VPUF) framework to speculate and sandbox the evaluation of the assert expression and repair activities. If repair activities do not succeed, then we can fall back to error virtualization.

- measure two things for SelCA:
  - compilation overhead in terms of extra time and space requirements for compiled code
  - overhead due to system operation, including both speculated and non-speculated forms of SelCA on a range of applications from those using sparse amounts of asserts and those using a heavy amount

- determine if SelCA assist the application in surviving faults (can use fault-injection)

- *potentially evaluate the impact of inserting SelCA at each function entry and exit point

5.1.2 Tasks for VPUF

- set up CVS repository for VPUF

- explore the QEMU emulator for adaptation and use as a virtual processor

- design regression test suite for modified QEMU

- *design regression test suite for modified kernel

- look at an example of a device emulator for the kernel

- implement a small device like a virtual USB calculator to get a sense of what pieces of the kernel need to be notified about new hardware and as practice for writing virtual CPUs as loadable kernel modules

- examine Linux kernel code that detects and keeps track of CPUs (take hint from code that maintains /proc/cpuinfo)

- create framework/library, namespace, and API for virtual CPUs as LKM's

- add virtual process sleep and run queues, and modify scheduler to use these queues if any process requests it

- add system call to “vexec” or virtually execute a process (given the PID) on a virtual CPU (given the legal and standard virtual CPU name) if such a CPU is registered with the kernel
- exercise the framework by running standard CPU benchmarks

5.2 Timeline

Table 1: **Research Plan Outline.** *This table provides an overall monthly roadmap for completing the research, development, and writing of this thesis.*

Timeline	Work Item	Progress
April 2005	Background and related work research	completed
May 2005	Thesis proposal outline	completed
June 2005	Break, Design of SVV (Snakeyes)	completed
July 2005	Break, Requirements for VPUF (Snakeyes)	completed
Aug. 2005	Break, (Snakeyes)	completed
Sept. 2005	Present SVV, Requirements for SelCA	completed
Oct. 2005	Thesis proposal writing	completed
Nov. 2005	Thesis proposal distribution, defense	ongoing
Dec. 2005	Design for Self-Correcting Asserts (SelCA)	ongoing
Jan. 2006	Implementation of SelCA	XXX
Feb. 2006	Design, Construction of VPUF	XXX
Mar. 2006	Construction of VPUF	XXX
April 2006	Integration	XXX
May 2006	Testing	XXX
June 2006	Data Collection and Testing	XXX
July 2006	VPUF Evaluation and Data Analysis	XXX
Aug. 2006	SelCA Evaluation and Data Analysis	XXX
Sept. 2006	Outline of dissertation, conferences	XXX
Oct. 2006	conferences and applications	XXX
Nov. 2006	conferences and applications	XXX
Dec. 2006	Dissertation construction and writing	XXX
Jan. 2007	Dissertation construction and writing	XXX
Feb. 2007	Dissertation talk construction	XXX
Mar. 2007	Dissertation distribution and defense	XXX
April 2007	Dissertation revisions	XXX
May 2007	Deposit and Graduation	XXX

I plan to complete the research and writing of the thesis according to the timeline in Table 1. I plan to apply for tenure-track faculty positions in the Fall of 2006, defend my thesis in March of 2007, incorporate the suggestions of my committee the following month, and deposit the completed thesis in May of 2007.

6 Summary

A key problem in computer security is the inability of systems to automatically protect themselves from attack. Therefore, the goal of this thesis is to provide an environment where both supervision and automatic remediation can take place to support Self-Healing Software. We also introduce ROAR (Recognize, Orient, Adapt, Respond) as a conceptual workflow for Self-healing Software systems.

This thesis proposal outlines a 17 month program for developing and evaluating the major components of the proposed system: the *Virtual CPU Framework (VPUF)* and *Self-Correcting* `assert()`*'s (SelCA)* that collectively provide a safe environment for automatic monitoring and remediation. VPUF introduces the concept of computation as an operating system service by implementing a new *endolithic* kernel architecture that provides control for a collection of virtual processors within the Linux kernel. SelCA provide a largely transparent policy mechanism to employ useful and *correct* remediation mechanisms for automatically handling exploits. Current techniques for automatic intrusion reaction and response do not provide provably correct remediation mechanisms.

Three novel techniques support VPUF and SelCA: micro-sandboxing, micro-speculation, and self-correcting assertions. These techniques are leveraged by VPUF to speculatively execute code that may contain faults or vulnerabilities and automatically repair such faults or exploited vulnerabilities. While VPUF provides an environment for micro-sandboxing and micro-speculation, it requires some detection and remediation mechanism to trigger it and direct its actions when a fault occurs. Automating a response strategy is difficult, as it is often unclear what a program should do in response to an error or attack. The key idea of SelCA is to make the assertion policy consistent at the instrumentation points by changing the relevant program state to match the expected value of the policy.

Finally, this proposal looks forward to several areas of follow-on work, including the implementation of a novel pipeline organization called *Speculative Virtual Verification (SVV)* that is meant to provide low-level support to ameliorate the performance impact of our techniques. The work in this thesis can also be leveraged to support the notion of collaborative security in Application Communities.

Unless programmers develop prescience and programs become capable of computing uncomputable functions, the design of a program will always lack a complete description of how to handle all errors. The opportunity and motivation to take advantage of these errors will not disappear as long as computing systems are trusted with the task of managing important data and resources. No system can be perfectly secure, but we can provide well-formed recovery mechanisms and invoke them automatically. This thesis assists in bridging the gap between current systems and systems that are able to automatically self-heal.

References

[1] The SUBTERFUGUE Project. `http://subterfugue.org/`, April 2002.

[2] A. Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack Smashing Attacks. In *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In 19^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[4] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanovic, and D. D. Zovi. Randomized Instruction Set Emulation to Distrupt Binary Code Injection Attacks. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[5] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the* 12^{th} *USENIX Security Symposium*, pages 105–120, August 2003.

[6] George Candea and Armando Fox. Crash-Only Software. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems (HOTOS-IX)*, May 2003.

[7] Benjie Chen and Robert Morris. Certifying Program Execution with Secure Processors. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems*, pages 133–138, May 2003.

[8] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. 1998.

[9] Brian Demsky and Martin C. Rinard. Automatic Data Structure Repair for Self-Healing Systems. In *Proceedings of the* 1^{st} *Workshop on Algorithms and Architectures for Self-Managing Systems*, June 2003.

[10] Brian Demsky and Martin C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, October 2003.

[11] J. Etoh. GCC Extension for Protecting Applications From Stack-smashing Attacks. In *http://www.trl.ibm.com/projects/security/ssp*, June 2000.

[12] Marius Evers, Sanjay J. Patel, and Yale N. Patt. An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work. In *Proceedings of the* 25^{th} *International Symposium on Computer Architecture*, June 1998.

[13] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. In *Proceedings of the* 6^{th} *Workshop on Hot Topics in Operating Systems*, pages 67–72, 1997.

[14] Timothy Fraser, Lee Badger, and Mark Feldman. Hardening COTS Software with Generic Software Wrappers. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, 1999.

[15] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In 10^{th} *ISOC Symposium on Network and Distributed Systems Security (SNDSS)*, February 2003.

[16] Tal Garfinkel, Mendel Rosenblum, and Dan Boneh. Flexible OS Support and Applications for Trusted Computing. In *Proceedings of the* 9^{th} *Workshop on Hot Topics in Operating Systems*, pages 145–150, May 2003.

[17] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. 2000.

[18] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 3^{rd} edition, 2003.

[19] David A. Holland, Ada T. Lim, and Margo I. Seltzer. An Architecture a Day Keeps The Hacker Away. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, October 2004.

[20] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Proceedings of the* 7^{th} *ACM International Conference on Computer and Communications Security (CCS)*, pages 190–199, November 2000.

[21] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the* 10^{th} *ACM Conference on Computer and Communications Security (CCS)*, October 2003.

[22] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure Execution Via Program Shepherding. In *Proceedings of the* 11^{th} *USENIX Security Symposium*, August 2002.

[23] Benjamin A. Kuperman, Carla E. Brodley, Hilmi Ozdoganoglu, T. N. Vijaykumar, and Ankit Jalote. Detection and Prevention of Stack Buffer Overflow Attacks. *Communications of the ACM*, 48(11):51–56, November 2005.

[24] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. In *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003), Lecture Notes in Computer Science, Springer Verlag*, March 2003.

[25] Tong Li, Carla S. Ellis, Alvin R. Lebeck, and Daniel J. Sorin. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *Proceedings of the USENIX ATC*, pages 31–44, April 2005.

[26] D. Lie, J. Mitchell, C. Thekkath, and M. Horwitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2003.

[27] D. Lie, C. Thekkath, M. Mitchell, and P. Lincoln. Architectural Support for Copy and Tamper Resistant Software. In *Proceedings of the* 9^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, 2000.

[28] David Lie, Chandramohan Thekkath, and Mark Horowitz. Implementing an Untrusted Operating System on Trusted Hardware. In 19^{th} *ACM Symposium on Operating Systems Principles (SOSP)*, October 2003.

[29] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Application Communities: Using Monoculture for Dependability. In *Proceedings of the* 1^{st} *Workshop on Hot Topics in System Dependability (HotDep-05)*, June 2005.

[30] Michael E. Locasto, Stelios Sidiroglou, and Angelos D. Keromytis. Speculative Virtual Verification: Policy-Constrained Speculative Execution. In *Proceedings of the* 14^{th} *New Security Paradigms Workshop (NSPW)*, pages 0–19, September 2005.

[31] Michael E. Locasto, Ke Wang, Angelos D. Keromytis, and Salvatore J. Stolfo. FLIPS: Hybrid Adaptive Intrusion Prevention. In *Proceedings of the* 11^{th} *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 0–19, September 2005.

[32] John P. McGregor and Ruby B. Lee. Protecting Cryptographic Keys and Computations via Virtual Secure Coprocessing. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, October 2004.

[33] Nicholas Nethercote and Julian Seward. Valgrind: A Program Supervision Framework. In *Electronic Notes in Theoretical Computer Science*, volume 89, 2003.

[34] Jeffrey Oplinger and Monica S. Lam. Enhancing Software Reliability with Speculative Threads. In *Proceedings of the* 10^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS X)*, October 2002.

[35] Richard E. Overill. How Re(Pro)active Should an IDS Be? In *Proceedings of the* 1^{st} *International Workshop on Recent Advances in Intrusion Detection (RAID)*, September 1998.

[36] H. Patil and C. N. Fischer. Efficient Turn-time Monitoring Using Shadow Processing. In *Proceedings of the* 2^{nd} *International Workshop on Automated and Algorithmic Debugging*, 1995.

[37] Nick L. Petroni, Timothy Fraser, Jesus Molina, and William A. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In 13^{th} *USENIX Security Symposium*, pages 179–194.

[38] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 207–225, August 2003.

[39] Purify. `http://www.rational.com/products/purify_unix/index.jtmpl`.

[40] RATS. `http://www.securesw.com/download_rats.htm`.

[41] James C. Reynolds, James Just, Larry Clough, and Ryan Maglich. On-Line Intrusion Detection and Attack Prevention Using Diversity, Genrate-and-Test, and Generalization. In *Proceedings of the* 36^{th} *Hawaii International Conference on System Sciences (HICSS)*, 2003.

[42] M. Rinard, C. Cadar, D. Dumitran, D. Roy, T. Leu, and Jr. W Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings* 6^{th} *Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[43] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In 13^{th} *USENIX Security Symposium*, pages 223–238.

[44] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and Dan Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the* 11^{th} *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[45] Stelios Sidiroglou, John Ioannidis, Angelos D. Keromytis, and Salvatore J. Stolfo. An Email Worm Vaccine Architecture. In *Proceedings of the* 1^{st} *Information Security Practice and Experience Conference (ISPEC)*, April 2005.

[46] Stelios Sidiroglou and Angelos D. Keromytis. A Network Worm Vaccine Architecture. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WET-ICE), Workshop on Enterprise Security*, pages 220–225, June 2003.

[47] Stelios Sidiroglou, Michael E. Locasto, Stephen W. Boyd, and Angelos D. Keromytis. Building a Reactive Immune System for Software Services. In *Proceedings of the USENIX Annual Technical Conference*, pages 149–161, April 2005.

[48] Stelios Sidiroglou, Michael E. Locasto, and Angelos D. Keromytis. Hardware Support For Self-Healing Software Services. In *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, pages 42–47, October 2004.

[49] Stelios Sidiroglou, Michael E. Locasto, and Angelos D. Keromytis. Hardware Support For Self-Healing Software Services. *ACM SIGARCH Computer Architecture News*, 33(1):42–47, March 2005.

[50] Christopher Small and Margo Seltzer. MiSFIT: A Tool for Constructing Safe Extensible C++ Systems. *IEEE Concurrency*, 6(3):33–41, 1998.

[51] A. Smirnov and T. Chiueh. DIRA: Automatic Detection, Identification, and Repair of Control-Hijacking Attacks. In *The* 12^{th} *Annual Network and Distributed System Security Symposium*, February 2005.

[52] A. Somayaji and S. Forrest. Automated Response Using System-Call Delays. In *Proceedings of the* 9^{th} *USENIX Security Symposium*, August 2000.

[53] Splint. `http://www.splint.org`.

[54] G. Edward Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the* 11^{th} *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XI)*, October 2004.

[55] Dean Turner and Stephen Entwisle. Symantec Internet Security Threat Report. `http://enterprisesecurity.symantec.com/content.cfm?articleid=1539`, September 2004.

[56] Nicholas Wang, Michael Fertig, and Sanjay J. Patel. Y-Branches: When You Come to a Fork in the Road, Take It. In *Proceedings of the* 12^{th} *International Conference on Parallel Architectures and Compilation Techniques*, September 2003.

[57] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the* 22^{nd} *International Symposium on Reliable Distributed Systems (SRDS)*, 2003.