Jeffrey Dwoskin, Mahadevan Gomathisankaran, Ruby Lee. "A Framework for Testing Hardware-Software Security Architectures", Princeton University Department of Electrical Engineering Technical Report CE-L2009-001, June, 2009.

A Framework for Testing Hardware-Software Security Architectures

Jeffrey S. Dwoskin
jdwoskin@princeton.edu

Mahadevan Gomathisankaran
mgomathi@princeton.edu

Ruby B. Lee
rblee@princeton.edu

Abstract

New security architectures are difficult to prototype and test at the design stage. Fine-grained monitoring of the interactions between hardware, the operating system, and applications is required. We have designed and prototyped a testing framework, using virtualization, that can emulate the behavior of new hardware mechanisms in the virtual CPU and can perform a wide range of hardware and software attacks on the system under test.

Our testing framework provides APIs for monitoring hardware and software events in the system under test, launching attacks, and observing their effects. We demonstrate its use by testing the security properties of the Secret Protection (SP) architecture [1] using a suite of attacks. Our framework enables extensive testing of hardware-software security architectures, in a realistic and flexible environment, with good performance provided by virtualization.

1 Introduction

Designers of security architectures face the challenge of testing new designs to validate the required security properties. To provide strong guarantees of protection, it is often necessary and desirable to place low-level security mechanisms in the hardware or operating system kernel, which the higher-level software layers can rely upon for a wide-range of applications. The resulting architecture is a combination of hardware, kernel, and application software components which are difficult to test together. The security of the system as a whole relies on the combination of the correct design and implementation of the low-level security features, the correct and secure use of those features by the software layers, and the security of the software components themselves (e.g., their resilience to attacks). Therefore, we need a framework that can comprehensively model the architecture and study the interactions between hardware and software components. Furthermore, the security mechanisms at each level must be robust under normal operation as well as when under active attack by an adversary.

We propose a testing framework that emulates the hardware components of a security architecture and provides a controlled environment with a full software stack, with which coordinated security attacks can be performed and observed. We have designed our testing framework with the initial goal of verifying the SP (Secret Protection) architecture [1, 2], while being generalizable to other security architectures. SP places roots of trust in the hardware which are used to protect security-critical software at the application layer, skipping over the operating system layer in the trust chain. The threat model includes attacks on software components as well as most physical attacks, with only the processor chip itself trusted. The operating system remains untrusted and essentially unmodified. The usefulness of the SP architecture depends on verifying its security properties and demonstrating that security-critical applications can be built effectively on such an architecture.

Testing architectures like SP requires that the framework models attack methods, using known and unknown penetration methods, and studies their impact on the security mechanisms provided by the hardware and software. The testing environment — including the hardware implementation, software stack, threat models, and attack mechanisms — must be as realistic as possible so that results are meaningful. As far as we know, no existing testing methods provide the ability to mount coordinated attacks on new integrated hardware-software security architectures and observe their effects, as our framework does.

Furthermore, our framework allows testing to be done during the design time; this gives confidence in the architecture before the complete system is built, at which point it is costly to make fundamental changes in response to security flaws. While we focus our analysis on SP, a hardware-software security architecture, our framework is equally useful for testing software-only architectures.

Our Approach

Our testing framework is composed of two components: a system under test (SUT) containing the new hardware architecture and software stack under consideration, and a testing system (TS) that coordinates monitoring of, and attacks on, the SUT. The SUT is attacked by the TS according to the threat model being tested; the TS itself is not attacked. The TS monitors hardware and software events that occur in the SUT using hooks provided by our framework. It also can inject attacks at all layers of the system. An attack script running in the TS coordinates events and attacks from both hardware and software components in the SUT.

Our testing framework can model real attack mechanisms using known penetration mechanisms. It can also model unknown future attacks by more powerful adversaries by enabling direct attacks on software and hardware components that go beyond known penetration methods. We do this by mapping attacks to their *impacts* on the SUT. Hence, a key strength of our system is that it allows design-time testing assuming a very powerful attacker to test the limits of the SUT, without the need to find a specific penetration path through the system.

The primary contributions of this work are:

- a new flexible framework for design-time testing of the security properties of new hardware-software architectures;
- enabling a realistic software stack, using commodity operating systems, for testing different applications using the new security mechanisms;
- a flexible, fast, and low-cost method for emulating hardware security features, using virtualization, for the purpose of design validation — without the need for costly and time-consuming fabrication of hardware prototypes;
- the ability to simulate the impact of very powerful attackers for "future" attacks;
- an improved architecture for SP's secure memory mechanism and its implementation; and
- the application of our framework toward the validation of the security properties of the SP architecture, by providing a suite of attacks on SP's trust chain.

The rest of the paper is organized as follows: Section 2 discusses hardware-software architectures and their threat models. Section 3 describes the architecture and implementation of our new testing framework. Section 4 describes the SP architecture, its emulation in the testing framework, and how trusted software can be constructed. Section 5 presents methodologies for comprehensive testing of the security properties of SP and its trusted software. We discuss related work in Section 6 and conclude in Section 7.

2 Threat Model and Assumptions

For this work, we focus on hardware-software architectures where new hardware security mechanisms are added to a general-purpose computing platform to protect security-critical software and its critical data. The hardware provides strong non-circumventable security protection, and the software provides flexibility to implement different security policies for specific applications and usage scenarios.

We assume a system with security-critical software applications running on a platform with new hardware security mechanisms added to the CPU (e.g., new instructions, registers, exceptions, and hardware mechanisms). Sometimes the OS cannot be trusted, especially if it is a large monolithic OS like Windows or Linux. Other times, an architecture might trust parts of the operating system kernel (e.g., a microkernel [3]), but not the entire operating system.

We consider three classes of attacks in our testing framework. First, malware or exploitable software vulnerabilities that can allow adversaries to take full control over the operating system to perform software attacks. They can access and modify all OS-level abstractions such as processes, virtual memory and virtual memory translations, file systems, system calls, kernel data structures, interrupt behavior, general registers, and I/O.

Second, hardware attacks, which can be performed by adversaries with physical possession of a device, such as directly accessing data on the hard disk, probing physical memory, and intercepting data on the display and I/O buses. For software-only security architectures, we can also model some software attacks as having the same impact as these physical attacks.

Third, network attacks that can be performed with either software or hardware access to the device, or with access to the network itself. Some network attack mechanisms act like software attacks (e.g., remote exploits on software), while others attack the network itself (e.g., eavesdropping attacks) or application-specific network protocols (e.g., modification attacks and man-in-the-middle attacks).

In order to adequately test a new security architecture, all of these attack mechanisms must be considered and tested, according to the threat model of the particular architecture. Our testing framework provides hooks into each relevant system component, and allows information and events at each level to be correlated to emulate the most knowledgable attacker.

Overall, we consider the functional correctness of the new hardware security mechanisms and the security-critical software components, as well as the interaction between these hardware and software components. We do not consider timing or other side-channel attacks.

3 Testing Framework

The design goal of our testing framework is to create a generic platform that can emulate and test a wide range of security architectures in a realistic environment. It must test the application software, which uses new secure hardware features, running on top of a full commodity operating system. The testing framework should allow easy monitoring of software and hardware events, and allow modification of software and hardware state to mimic attacks on the system.

3.1 Architecture

We build our testing framework on top of existing virtualization technology, which allows us to run a full set of commodity software efficiently. A virtual machine monitor (VMM) is the software that creates and isolates Virtual Machines (VMs), efficiently providing an execution environment in each VM which is almost identical to the original machine [4, 5]. By modifying an existing VMM's hardware virtualization, we can augment the virtual machine to have the additional hardware features of a new security architecture in addition to those of the base machine. Using virtualization allows the unmodified hardware and software components to run at near-native speed, while permitting our framework to intercept events and system state as needed.

Our Testing Framework is divided into two systems, as shown in Figure 1, the System Under Test (SUT) and the Testing System (TS), each running as a virtual machine on our modified VMM. The SUT is meant to behave as closely as possible to a real system which has the new security architecture. It has all of the new hardware security primitives, along with the corresponding protected software for that architecture. In our current system, the SUT runs a full commodity operating system (Linux) as its guest OS, which is vulnerable to attack and is untrusted.

The TS machine simulates the attacker, who is trying to violate the security properties of the SUT. It is kept as a separate virtual machine so that the TS Controller can be outside of the SUT to launch hardware attacks. The virtualization isolates all testing activity and networking from the host machine.

The testing framework itself is independent of the threat model of the system being tested, and hence enables full controllability and observation of the SUT in both hardware and software. This makes it suitable for many purposes during the design phase of a new architecture. During the initial design and implementation of the system, the TS can act as a debugger, able to see the low level behaviors in hardware, all code behavior, and data in the software stack. When testing the supposedly correct system, the TS is the attacker, constrained by a threat model to certain attack vectors.

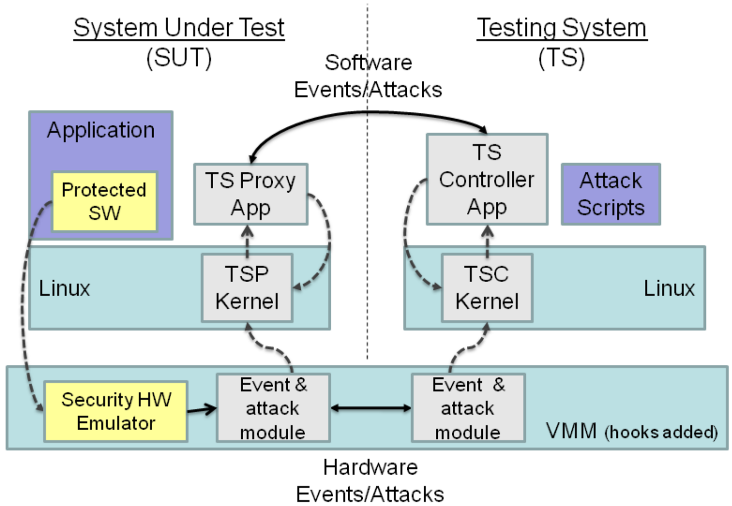

Figure 1: Testing Framework Design

A particular point of elegance of our framework is that the threat model can be easily changed, and the set of attack tools given to the attacker adjusted for each test. The framework can be used for any combination of mechanisms: access to internal CPU state of the virtual processor, "physical" attacks on the virtual machine hardware (e.g. hardware probes on the buses, memory, or disk), software attacks on the operating system (e.g. a rootkit installed in the OS kernel), and network attacks (e.g. interception and modification of network packets and abuse of network protocols and application data). For example, in some cases, it might be desirable to perform black-box testing of a new design using only the network to gain access to the SUT, while in other cases, white-box testing will allow the attacker knowledge about the system's activities, such as precise timing of attacks with hardware interrupts or breakpoints into the application code, or observation of data structures in memory.

Table 1: Example Events and Attacks

Layer	**Events Monitored**	**Impact of Attack**
Protected Application	API function entry/exit, Library calls, User authentication, Network messages, Other application-specific events.	Read/write application data structures, Trigger application API calls, Intercept/modify network messages, Other application-specific attacks.
OS	Memory access watchpoints, Virtual memory paging, File system access, System calls, Process scheduling, Instruction breakpoints, Device driver access, Network socket access, Interrupt handler invocation, etc.	Read/write virtual memory, Read/write kernel data structures, Read/write file system, Intercept/modify syscall parameters or return values, Read/write suspended process state, Modify process scheduling, Intercept/modify network data, Modify virtual memory translations.
Base Hardware (x86)	Privileged instruction execution, Triggering of page faults and other interrupts, Execution of an instruction pointer.	Read/write general registers, Read/write physical memory, Trigger interrupts, Intercept device I/O (e.g. raw network & disk accesses).
Secure Hardware	Execution of new instructions, Triggering of new faults, Accesses to new registers.	Read/write new registers & state, Read/write protected memory plaintext.

3.2 Testing Framework Components

The main components of our Testing Framework are shown in Figure 1. The framework detects events in the SUT and provides the TS with access to the full system state using both hardware and software channels. The TS Controller (TSC), running in the TS, is the aggregation point that receives events from both channels. It receives OS and Application level (software) events from the SUT via a network channel and receives hardware events from the VMM.

The TS Proxy (TSP) is added to the SUT to communicate with the TS Controller to receive commands and send events back. It simulates the effect of a compromised operating system for launching software attacks, allowing the OS to be fully controllable by the TS. The TS Controller and TS Proxy are each divided into user-level and kernel-level components. Additional trusted entities of the security architecture that are not under test, such as network servers, may be hosted in the TS and report their activity directly to the TS Controller.

Attack Scripts specify how particular attacks are executed on the SUT, making use of the TS Controller through an API. The Controller passes hardware and software events to an attack script, which can respond by accessing and modifying the state of the SUT. The script monitors events and dynamically responds to them in order to successfully launch attacks, or to detect that an attack has been prevented by the security architecture.

To capture events and access system state in the SUT, the modified VMM monitors and controls the hardware with the Event & Attack Module providing hooks into the virtual CPU and virtual devices, as well as into the new Security Hardware Emulator for new hardware not present in the base CPU. The TS Proxy monitors and controls the applications and OS. Communication of events and data between the SUT and TS occurs asynchronously through a network channel for software events/attacks and through a custom channel within the VMM[1] for hardware events/attacks. When synchronization is necessary, either the application or the entire SUT machine can be frozen to preserve state, while the TS and attack scripts continue to execute.

3.3 Events and Attacks

Table 1 lists various events and attacks exposed by the framework for each layer of the system. The hardware layer is further classified into events and attacks for the base hardware (x86 architecture in our work) and the new emulated security architecture.

Hardware events are monitored through the VMM hooks during execution and are as fine-grained as the execution of a single instruction or hardware operation in the SUT. The VMM freezes the SUT as it communicates each event over the inter-VM channel, allowing the TS to possibly change the result of that operation before it completes. Software events and attacks rely on hooks from the TS Proxy into the OS kernel through its kernel module, and to the testing application using its user-mode component. The TS Proxy can also function as a debugger tool reading the application's memory and accessing its symbol table to map variable and function names to virtual addresses. The application can optionally be instrumented to access its state and events. The TS Proxy suspends scheduling of the application's process while it is communicating an event to the TS Controller or performing an attack.

Table 2 lists the API which the TS Controller exports to the attack scripts, combining the hardware and software channels. First are commands used to launch and control the execution of the application under test on the SUT. The second group of commands control event handling[2], and the last group provides access to SUT state.

The attack mechanisms we provide are designed around the *impact* of attacks on the SUT's state. This is preferred since (1) new attack penetration methods are frequently discovered after a system is deployed and often are not foreseen by the designer, (2) most real attacks result in or can be modeled by the impact of attacks which we provide in Table 1, and (3) the attack scripts themselves can be restricted to model specific penetration methods when testing for a more limited attacker.

[1]The hardware channel is implemented over shared memory between each VM's Event & Attack Module.

[2]The watch list can wait for any of the event types in Table 1. Event parameters and data are either passed to the TS directly or are accessible via pointers with *Access_Mem*.

Table 2: TS Controller API for Attack Scripts

Function	Description
`h ← INIT()`	Initialize the Controller and return a handle *h* to access resources.
`EXECUTE(h,app,params)`	Execute the application *app* on SUT with the given parameters *params*.
`INTERRUPT(h,num)`	Trigger an immediate virtual hardware interrupt number *num* on the SUT.
`BREAKPOINT(h,addr)`	Setup a breakpoint to interrupt the SUT at an address (addr).
`EVENTADD(h,eventType)`	Add the *eventType* to watch-list.
`EVENTDEL(h,eventType)`	Delete the *eventType* from the watch-list.
`event ← WAIT(h)`	Blocking call that *waits* for any event in the watch-list to occur in the SUT. Once an event is triggered, the SUT is paused and the TS continues running the attack script. An application exit in the SUT always causes a return from WAIT().
`event ← WAITFOR(h,eventType)`	Similar to WAIT() but waits for the specified event (or application exit), regardless of the watch-list.
`CONT(h)`	Execution of the SUT is resumed, after an event or interrupt.
`ACCESS_GENREG(h,r/w,buf)` `ACCESS_SPREG(h,r/w,buf)`	Reads/writes (*r/w*) the general registers or SP registers of the SUT to/from *buf*.
`ACCESS_MEM(h,v/p,r/w,addr,sz,buf)` `ACCESS_SPMEM(...)`	Reads/writes (*r/w*) *sz* bytes from virtual or physical memory (*v/p*) of the SUT at address *addr* to/from the buffer *buf*. Can access memory regularly or as an SP secure region (accessing the plaintext of encrypted memory).

3.4 Implementation

We implemented our testing framework on VMware's virtualization platform [6], including all of the components in Figure 1, and events and attacks at each system layer. The Security HW Emulator, VMM Event & Attack Module, and inter-VM communication channel required modifying the source code of the VMware VMM. The kernel components of the TS Proxy and TS Controller are implemented as Linux kernel modules. The TS Proxy application is implemented as a Linux user process and controls the execution of the Testing Application. The TS Controller application is implemented as a static library which is called by the Attack Scripts.

As a sample security architecture, we implement the SP architecture, described in Section 4. The Security Hardware Emulator emulates the SP architecture including its hardware roots of trust, secure memory, and interrupt protection. We have also implemented a library of protected software for SP, which is used for a remote key-management application as described in Section 4.5. Our Testing Application uses this library to exercise the software, and in turn, the SP hardware.

Our framework, by using existing virtualization technology, enables reasonable performance while allowing our SUT to provide a realistic software stack and emulate new hardware. Other virtualization environments, like Xen [7], can also be used. Other simulation and emulation environments available, such as Bochs [8] and QEMU [9], could be used in place of virtualization to implement our framework as designed and described in this paper. We choose a virtualization environment for performance reasons, because only parts of the hardware and protected software need to be emulated, while the OS and other non-protected software can run virtualized. VMware provided an excellent development environment, under the VMAP program.

4 SP Architecture and Emulation

We use the Secret Protection (SP) architecture [2, 1] to demonstrate the application of our framework. SP skips software layers in the conventional trust chain by using hardware to directly protect an application without trusting the underlying operating system. SP protects the confidentiality and integrity of cryptographic keys in its persistent storage which in turn protect sensitive user data. These security properties provided by SP need to be validated. Furthermore, it is important to write and test many secure software applications for SP in a realistic environment, where the untrusted OS can be a source of attacks.

Our testing framework emulates SP's hardware features using modifications to the VMM. While SP hardware primitives have already undergone a detailed security analysis on paper, the framework can test the robustness of the design and its implementation, as well as discover any potential flaws. Additionally, we modify SP's secure memory mechanisms and then show how our framework can be used to demonstrate that these new hardware features are also resilient to attack.

The framework also serves as a platform to develop trusted software as part of the SP architecture. Any such software must be tested to be sure that it correctly uses the SP hardware mechanisms to protect itself and its data. It must correctly enforce policies and prevent leaking keys and data under physical attacks, attacks by the operating system, and interactions with untrusted software.

4.1 Secret Protection (SP) Architecture

In the Secret Protection (SP) architecture, the hardware primarily protects a Trusted Software Module (TSM), which protects the sensitive or confidential data of an application. Hence, a TSM plus hardware SP mechanisms form a trust chain for the application. Rather than protecting an entire application, only the security-critical parts are made into a TSM, while the rest of the application can remain untrusted. Furthermore the operating system is not trusted; the hardware directly protects the TSM's execution and data. (See Figure 2.)

Protecting the TSM's execution requires ensuring the integrity of its code and the confidentiality and integrity of its intermediate data. Code must be protected from the time it is stored on disk until execution in the processor. Data must be protected any time when the operating system or other software can access it. This includes storage on disk, in main memory, and in general registers when the TSM is interrupted.

To provide this protection, SP provides new hardware mechanisms. *Roots of Trust:* SP maintains its state using new processor registers; the threat model of SP assumes the processor chip to be the security boundary, safe from physical attacks which are very costly to mount on modern processors. As shown in Figure 2, SP uses two on-chip roots of trust: the Device Root Key and the Storage Root Hash. *Code Integrity:* The Device Root Key is used to sign a MAC (a keyed cryptographic hash) of each block of TSM code on disk. When a TSM is executing, the processor enters a protected mode called Concealed Execution Mode (CEM). As the code is loaded into the processor for execution in the protected mode, the processor hardware verifies the MAC before executing each instruction. *Data Protection:* For the TSM's intermediate data, while in protected mode, the TSM can designate certain memory accesses as "secure", which will cause the data to be encrypted and hashed before being evicted from on-chip caches to main memory. This secure data is verified and decrypted when it is loaded back into the processor from secure memory. Secure data and code are tracked with tag bits added to the on-chip caches. *Interrupt Protection:* Additionally, the SP hardware intercepts all faults and interrupts that occur while in the protected mode before the OS gets control of the processor. SP encrypts the contents of the general registers in place, and keeps a hash of the registers on-chip in the interrupt registers; When the TSM is resumed, the hash is verified before decryption of the registers.

The TSM protects secret data belonging to the application in persistent storage. SP allows a TSM (and no other software) to derive new keys from the Device Root Key using a new hardware instruction, *DRK_DeriveKey*. These derived keys are used by the TSM to protect the confidentiality of its persistent data. Furthermore, the TSM is the only software that can read and write the Storage Root Hash register, using it as the root of a hash tree to protect the integrity of this persistent secure data.

Hence, to emulate SP hardware we require the following components: new processor registers (including the protected mode and roots of trust); new instructions; hardware mechanisms for code integrity checking, secure memory, and interrupt protection; and new hardware faults which these mechanisms generate.

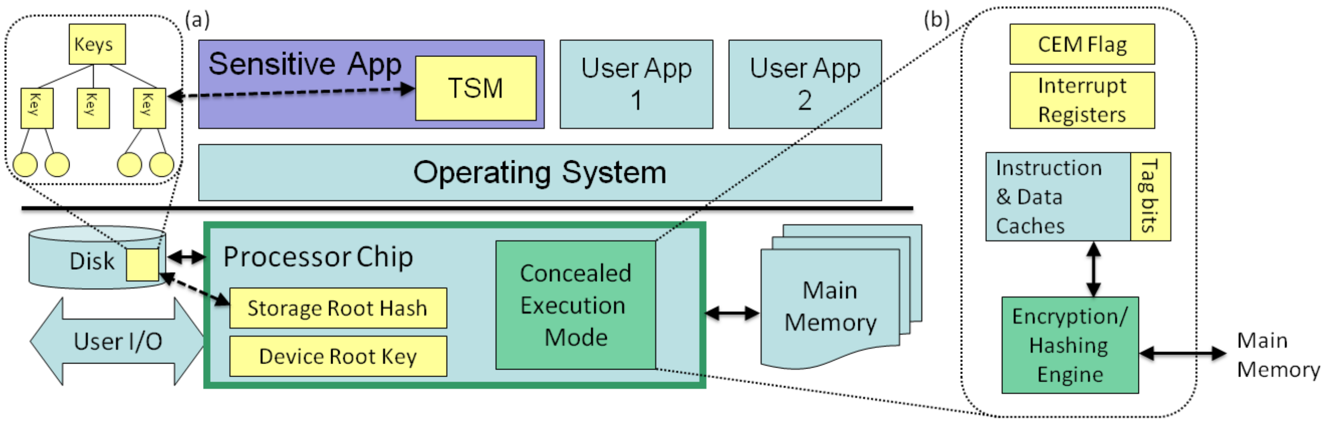

Figure 2: Secret Protection (SP) Architecture. Enlargements show (a) the application secrets protected by the TSM, and (b) the Concealed Execution Mode (CEM) hardware.

4.2 Emulation of the SP Architecture

A traditional VMM provides a virtual machine which is (nearly) identical to the physical machine, matching the instruction set and behavior of the real CPU. It does this by trapping or translating privileged code, while ignoring hardware effects that are transparent to software, such as cache memory. Most of the time, the VMM runs code on the physical hardware and only emulates the components that are virtualized. In order to implement and emulate new hardware architecture features, we take advantage of the VMM's virtualization methods. For example, the VMM maintains data structures for the virtual CPU state, which we expand to store new security registers. The VMM then emulates accesses that are made to those new registers. Other useful VMM behaviors include: interception of all hardware interrupts, dynamic binary translation of code, mapping of virtual memory translations, and virtualization of hardware devices.

To emulate the SP architecture, the Security Hardware Emulator Module implements the following:

Protected Mode SP requires new registers to be added to the virtual CPU, which include those for Concealed Execution Mode [2] — new mode bits and interrupt handling registers — and the roots of trust. New SP instructions are modeled as hypercalls, where the TSM running in the SUT is able to directly invoke the emulation module without going through the guest OS.

Interrupts and SP Faults The SP architecture changes the hardware interrupt behavior when in protected mode. Since the VMM already emulates interrupt behavior, we simply detect that an interrupt has occurred during the protected mode and emulate the effect on the CPU, which includes suspending the protected mode and encrypting and hashing the general registers. To detect returning from an interrupt, the VMM inserts a breakpoint at the current instruction pointer where the interrupt occurs, so that it is invoked to emulate the return-from-interrupt behavior of SP. Additionally, when the emulated hardware generates a new fault, it first reports to the TS Controller and then translates the fault into a real x86 fault, such as a general protection fault, which is raised in the SUT causing the OS to detect the failure of the TSM.

Secure Memory We change the SP abstraction of secure memory, as described below. Further, we use block sizes of virtual memory pages rather than individual cache lines, since the VMM does not intercept cache memory accesses. While this limits the ability to model a few low-level attacks on SP (such as the behavior of cache tags), the majority of the security properties of the hardware and all those of the software can still be tested.

Code Integrity The TSM's code is signed with a keyed hash over each cache-line of code and the virtual address of that line, and is checked as each cache line is loaded into the processor during execution. We model this using the VMM's binary translator to execute the TSM code. Verified instructions are tagged as secure code fragments in the dynamic binary translator cache.

4.3 Secure Memory

The original SP architecture uses two new instructions for a TSM to access secure memory: *Secure Load* and *Secure Store*. With these, any virtual address can be accessed as secure memory, where cache lines are tagged as secure (accessible only to a TSM) and are encrypted and MACed upon eviction from cache. We introduce a new secure memory model, called *Secure Areas*, to replace *Secure Load/Store*.

There are a few drawbacks to the *Secure Load/Store* approach. First, while most new SP instructions can be used as inline-assembly, the compiler must be modified to emit the secure memory instructions whenever accessing protected data structures or the TSM's stack. This further requires programmers to annotate their code to indicate which data structures and variables to protect, and which code and functions are part of a TSM. Second, while a RISC architecture need only supplement a few *Load* and *Store* instructions with their secure counterparts, a CISC architecture has many more instructions which access memory rather than general registers and need to support secure memory access. Third, while SP provides confidentiality and integrity for its secure memory, replay protection is also required to prevent manipulation of the TSM's behavior, but was not explicitly described. Rather, SP assumes a memory integrity tree [10, 11, 12] spanning the entire memory space, requiring significant overhead in on-chip storage and performance when only small amounts of memory need protection.

Secure Areas address these concerns by allowing the TSM to define certain regions of memory which are always treated as secure when accessed by a TSM. The programmer specifies the address range to protect explicitly, allowing the compiler to use regular memory instructions without modification. This is especially useful for our framework since the new architectural features can be tested during design-time without modifying the existing compilation toolchain. It also no longer requires duplicating all instructions in the instruction set which touch memory, a benefit for implementing SP on x86. Finally, it confines the secure memory to a few small regions which are more easily protected from memory replay attacks with less overhead.

Table 3: New SP Instructions for Secure Areas (only available to TSM)

Instruction	**Description**
`SecureArea_Add Rs1,Rs2,`*num* Rs1 = start_addr Rs2 = size *(must be aligned to block size)*	Initialize the specified Secure Area (region *num*). On-chip hashes for the region are cleared. All TSM memory accesses for *addr* will be treated as secure if: (start_addr) $\leq$ *addr* $<$ (start_addr + size).
`SecureArea_Relocate Rs1,`*num* Rs1 = start_addr	Change the starting address of the specified Secure Area region. The size remains unchanged. When TSM code in multiple process contexts share memory containing a Secure Area, each may access it at a different address in their virtual address space; this is used to relocate the region.
`SecureArea_Remove` *num*	Disables and clears the specified Secure Area region. On-chip hashes for the region are cleared and secure-tagged cache entries in its address range are invalidated, making any data in the region permanently inaccessible in plaintext.
`SecureArea_CheckAddr Rd,`*num* `SecureArea_CheckSize Rd,`*num*	Retrieves the parameters of the specified Secure Area region. Used to verify whether or not a region is setup for secure memory and where it is located.

Table 3 shows the new instructions added to SP to support Secure Areas, replacing *Secure Load* and *Secure Store*. The SP hardware offers a limited number of Secure Area regions, which the TSM can define using these instructions. Each region specifies an address range which is always treated as secure memory when accessed by the TSM, and is encrypted when accessed by any other software or hardware devices. At least two regions should be provided so a TSM can separately protect ranges of memory for its secure data and for its stack.

On-chip storage is needed for each region to track the start and end address, and to store hashes for each block within the region. The block size for hashing can range from one cache line to one virtual memory page, and is determined by the hardware implementation. Upon defining a new region, the corresponding on-chip hashes are cleared. As secure data is written, it is tagged as secure in cache; when it is evicted from cache the contents are encrypted and a hash is computed and stored in the on-chip storage for that block. It must be verified when the data is read back in from off-chip memory. Since the regions can be small relative to total memory (only tens to hundreds of kilobytes are needed for our prototype TSMs), only small amounts of on-chip storage are required. Alternatively, other memory integrity tree methods [11, 13, 10] can be integrated to store some hashes off-chip to permit replay protection of larger regions of secure memory.

We used our framework to test and debug these interfaces, to implement TSMs that use them, and to test those TSMs to verify that data is not leaked from secure data structures or from the stack (which we protect without modifying the compiler).

4.4 Other Architectures

While this paper focused on testing the hardware and software mechanisms of the SP architecture, our testing framework is by no means limited to this architecture. Other hardware security architectures such as XOM [14], AEGIS [15] and Arc3D [16] modify hardware in similar ways but have somewhat different goals and assumptions from SP. However they combine hardware and software in ways that also make them suitable for validation in our framework. Similarly, TPM [17] adds hardware to protect all software layers and provide cryptographic services. Rather than utilizing changes to the processor itself, TPM adds a separate hardware chip that integrates with the system board. This is still compatible with our testing framework, simply requiring a different set of modifications to the VMM to implement a virtual TPM device. Furthermore, software-only security architectures can benefit from analysis under attack in our framework, both during development and for security validation. Access to existing hardware state provides insight into attack impacts and possible flaws, and provides an additional vector for injecting attacks.

4.5 Remote Key-management TSM

We test an application TSM and its SP hardware protection using a remote key-management example, suitable for many applications, e.g., for emergency response [1, 18]. We subject our TSM to a suite of attacks to see if the desired security properties are preserved. We attack the robustness of the TSM's memory usage, persistent storage, network protocols, and software interfaces.

For remote key-management, we consider a trusted authority which owns multiple SP devices and wants to distribute sensitive data to them. The authority installs its remote key-management TSM on each device as well as the protected sensitive data, consisting of secrets and the cryptographic keys that protect those secrets. It also stores policies for each key which dictate how it may be used by the local user. During operation, the TSM will accept signed and encrypted messages from the authority to manage its stored keys, policies, and data. It also provides an interface to the application through which the local user can request access to data according to the policies attached to the keys. The TSM must authenticate the user, check the policy, and then decrypt and display the data as necessary.

5 Testing of SP

Testing a security architecture requires testing all components that are involved in establishing the trust chain. For the SP architecture, we use the framework to demonstrate testing the SP hardware mechanisms, the proper usage of these mechanisms by a TSM, and the proper enforcement of access control for the Remote Key-management TSM. In this section we elaborate on how we use the Testing Framework to achieve these goals, testing the system's security properties while it is under attack. In addition, we validate that the emulation of the SP mechanisms is correct and secure according to the design, as it forms the basis for the other tests.

Table 4 lists various attacks on the system's security properties. Data confidentiality is the primary purpose of the SP architecture. The attack generally checks to see if any sensitive data that should be protected by a TSM is ever leaked. We eavesdrop on the unprotected memory and check whether any known keys generated by the TSM, in addition to the Device Root Key (DRK) and any DRK-derived keys, are found. This is similar to the cold boot attack [19] which looks for sensitive keys left in physical memory. If the TSM properly uses secure memory for its intermediate data, and protects its persistent data, then no keys should ever leak.

The second section sets up a series of attacks on the basic mechanisms of SP, such as controlling access to the master secrets (e.g., Device Root Key), code integrity checking, and encryption of secure data in protected mode. These tests verify that the emulation is correct and also validate the original security analysis. For example, we attack SP's Concealed Execution Mode by attempting to modify registers during an interrupt. A non-TSM application's registers can be modified by a corrupted OS without detection, causing changes in the application's behavior. However, a TSM will have its registers encrypted and hashed by the SP hardware upon any interrupt, such that SP detects the modification when resuming the TSM.

We also test how TSMs use SP mechanisms to protect intermediate and persistent data. We test our new secure memory implementation to verify that a TSM compiled with GCC can adequately protect its intermediate data on the stack. Our example TSM in Figure 3 derives a new key from the Device Root Key and uses it for encryption. The attack script freezes the SUT shortly after the key is derived and scans physical memory. It finds that the key has been leaked via parameter passing on the stack, violating data confidentiality. As a result, we have instrumented a new software mechanism to swap the TSM's stack to use memory in a designated Secure Area. The same attack script then verifies that this modified TSM correctly protects the confidentiality of the key when passed as a parameter. This demonstrates how a secure hardware mechanism (e.g., for secure memory) can be used incorrectly by a TSM, often inadvertently, leading to vulnerabilities.

The next section in Table 4 shows generic attacks on a TSM, which test security properties common to many TSMs (e.g., control flow, entry points). These attacks consider that a basic goal of many TSMs (and indeed of the SP architecture) is to provide confidentiality and integrity to any sensitive information and enforce access control.

We develop tests of the robustness of the TSM against future unknown vulnerabilities that might arise in the hardware or TSM code. Since the penetration mechanism is unknown, we instead model the effects of the attack. For example, the control flow of the TSM could be attacked in many different ways. When the TSM makes branching decisions, the jump targets and the input data for the branch conditions should be protected. If either is not stored in secure memory, or if secure data can be modified or replayed, then arbitrary changes to the TSM's control flow would be possible. We verify that a TSM only bases control flow decisions on data in its secure memory, and test how control flow violations could cause data to leak.

As another example, we consider control flow attacks that allow arbitrary entry points into a TSM. Since instructions to enter protected mode (*Begin_TSM*) are not signed, *Begin_TSM* could be injected into the TSM to create an entry point. We implement this as an attack script, crafting a case where the TS overwrites instructions and tries to enter in the middle of a TSM function without detection, bypassing access control checks.[3] To prevent this, we add a new security requirement to SP that it must distinguish entry points in TSM code from blocks of code that are not entry points. This can be achieved by adding an extra bit to the calculation of the signature of each block of TSM code, indicating whether or not it is an entry point.

The attack on TSM page mappings demonstrates a system-level attack, where rather than attack the TSM directly, the OS manipulates the system behavior to indirectly affect how the TSM executes. The OS can manipulate process scheduling, intercept all I/O operations, and in this case, modify how virtual addresses map to physical addresses.

The last section in Table 4 shows application-specific attacks for a particular TSM — in this case our Remote Key-management TSM. This TSM stores cryptographic keys, security policies, and secure data in its persistent secure storage, which it protects using SP's underlying hardware mechanisms. We test the confidentiality and integrity of the storage itself, the TSM's use of the storage to protect keys and key-chains, and its enforcement of the policies on accesses to data that the keys protect. We also test the protocols the TSM uses to communicate with a remote authority, managing the keychains.

Our system implements the SP hardware mechanisms, a full TSM providing an API to the testing application, and a suite of attacks that test both the software and hardware components using our new testing framework. This

[3] In some cases, this attack would be detected by SP — if the injected instruction is not correctly aligned to the start of a block of signed code, or if later in execution the TSM jumps back to code before the injection site. A carefully crafted attack succeeds.

Table 4: Example Attacks on the SP Architecture Using the Testing Framework

Security Property	Attack
Data Confidentiality	Scan physical memory for leaks of Device Root Key, DRK-derived keys, and TSM's other sensitive information.
Securing General Registers on Interrupts	Attack the general registers during an interrupt of a TSM through eavesdropping, spoofing, splicing, and replay.
Code Integrity	Attack TSM code during execution through spoofing and splicing; attack TSM code on disk.
Secure Memory	Attack intermediate data of TSM through eavesdropping, spoofing, splicing and replay; attack the use of secure memory for TSM's data structures or stack.
Secure Storage	Attack the TSM's secure storage for persistent data (splicing, spoofing & replay).
Control Flow Integrity	Attack TSM's indirect jump targets that are derived from unprotected memory. Arbitrarily modify jump targets within the TSM.
	Attack the input data for branch conditions in the TSM from unprotected memory. Replay secure data to cause incorrect branch decisions.
	Attack TSM entry points by entering CEM at arbitrary points in the code, skipping access control checks or initialization of secure memory.
TSM Page Mappings	Remap TSM code pages and data pages, as a means to attack secure memory or control flow.
Key-chain management	Spoof key add/delete message; replay key-add message after it is deleted; corrupt a key-management message in transit.
Access control on keys	Exceed usage limits/expiration of keys; attempt to use a key that was deleted; attempt to perform a disallowed operation with a key.

is a major step towards the complete validation of the design of the SP architecture together with its applications. Furthermore, we demonstrate that TSMs must be carefully written to avoid serious security flaws, and that a security architecture can benefit from testing with many different applications. Our framework provides a platform for this necessary testing.

Testing Example

Figure 3 shows a sample TSM on the left, and a corresponding attack script using the TS Controller API (Table 2). This demonstrates the interactions between the TS and SUT for event detection and modification of SUT state. The simple attack shown here verifies that secure data (here the derived AES key), placed on the stack by the TSM as a function parameter, is not leaked in physical memory where the OS could read it. This attack demonstrates precise coordination of software events (injected breakpoints) with access to the hardware (physical memory state), while the SUT is frozen to prevent clearing or overwriting of any data in memory. The script also requires access to the internal state of the SP hardware from the TS to verify the results of the attack.

Attack scripts are typically longer and can involve many additional steps and interactions, along with a complete TSM and its corresponding application. The full range of events and attack mechanisms described in Table 1 are available to the attack scripts, with the TS in full control over the applications, OS, and hardware running in the SUT.

Application with TSM (`TSMapp`)	Attack Script (pseudocode)
BEGIN_TSM	EXECUTE(TSMapp, params)
...	*// Wait for key generation* EVENTADD(DRK_DeriveKey) EVT ← WAIT() *// Read the generated key* ACCESS_SPREG(r, SPRegs) SPKey ← SPRegs.DerivedKey
Reg1 ← DRK_DeriveKey(TTP_KeyID) SecureMem.AESkey ← Reg1	
// Breakpoint injected at start of Encrypt function Ciphertext ← Encrypt(SecureMem.AESkey, &SecureMem.data, sz) END_TSM	*// Inject breakpoint and wait for its interrupt* BREAKPOINT("&Encrypt"); CONT() EVT ← WAITFOR(Interrupt)

// Send encrypted file on network or store on disk
Network_Send(TTP, Ciphertext, sz)
...
	// Scan physical memory for leaked key
	for $addr = 0$ to $256M$ **do**
	ACCESS_MEM(PHYS, r, addr, SIZE, buf)
	if strstr (buf, SPKey) **then**
	return "Derived Key Leaked in Memory"
	print "Derived Key Not Found in Memory"

Figure 3: Example Application and Attack Script for Detecting Leaked Keys

6 Related Work

Our testing framework emulates new security architectures and runs full system software. Thus it can be compared against hardware simulators and full system emulators.

Micro-architectural simulators like Simplescalar [20] are cycle-accurate and hence can be very useful in estimating performance metrics, but they can not simulate a realistic software system with a full commodity OS. Thus it is impossible to test the security critical interactions of a software-hardware security solution with such a simulation architecture.

Chow, et. al. [21] use system emulation to passively trace data leaks in applications. However, our framework performs active attacks as well. Chow's work also does not consider violations other than data leaks, while we consider more security breaches, such as violations of data integrity, policy enforcement, and control flow. Furthermore, we are looking for flaws in both the TSM code and hardware mechanisms that are specifically designed to protect security, unlike Chow where the applications are tested for properties they were not designed for.

Virtual machine introspection [22, 23, 24] techniques provide access to VM-state in similar ways to our framework, however, they focus mostly on observability of software configurations or low-level hardware behavior. Examples include intrusion detection and virus-scanning from non-vulnerable host systems, preventing execution of malware, and tracing memory accesses. In contrast, we strive to combine observability of the full-system state with controllability of those same components, actively during operation, to attack software thought to be secure. Also, while previous work focused on techniques for security monitoring of production machines, we focus on design-time testing of new architectures or of new software systems to evaluate their potential vulnerabilities and flaws. Where some of these techniques provide improved hooks into the VMM [25], the hooks could be integrated into our framework to make the attack scripts more robust.

The efforts by IBM [26], Intel [27] and others [28] provide the functionality of a virtual TPM device to software, even when the physical device is not present. In contrast, we not only emulate new hardware but also hook into a virtual device to observe and control its behavior for testing purposes, and study the interaction with other hardware and software components.

Another related area of research is the formal verification of both hardware and software, in which formal methods are used to write specifications for computer hardware or software, and proof techniques are used to determine the validity of such specifications. The complexity of formal verification problems range from NP-hard to undecidable [29, 30, 31, 32]. This complexity led to the use of hybrid techniques [33] which use some formal as well as informal methods. Some formal methods of verification are those using theorem provers (ACL2 [34], Isabelle/HOL [35]), model checkers, satisfiability solvers, etc. Some other techniques used in practice are control circuit exploration, directed functional test generation, automatic test program generation, heuristic-based traversal, etc. The important distinction between our approach and these formal verification techniques is that we try to verify the complete system while these formal techniques try to verify each component piece by piece. The complexity of both specification and verification explodes exponentially with the addition of more and more pieces to be tested. In our approach we model both the security critical hardware and software together, thus making the verification problem solvable in an informal but systematic and efficient way.

7 Conclusion

We have designed and implemented a virtualization-based framework for validation of new security architectures. This framework can realistically model and test a new system during the design phase, and draw useful conclusions about the operation of the new architecture and its software interactions. It also enables testing of various software applications using new security primitives in the hardware or in the OS kernel.

Our framework serves as a rapid functional prototyping vehicle for black-box or white-box testing of security properties. It can utilize and *integrate* multiple event sources and attack mechanisms from the hardware and software layers of the system under test. These mechanisms can test both low-level components and high-level application behavior. As a result, a comprehensive set of attacks are realizable on the hardware, operating system, and applications.

We implement the SP architecture in our framework and test its security mechanisms thoroughly, studying the interactions of trusted software with the hardware protection mechanisms. We also improve the design of SP's secure memory for both emulation and hardware implementation. We implement a TSM for remote key-management to demonstrate a real application of the SP architecture. Using a suite of attacks on each layer of the architecture, we thoroughly test each component of SP's trust chain.

Future work includes testing more applications and their partitioning into Trusted Software Modules in SP, random attack generation, and devising additional cross-layer system attacks. We hope to have provided a framework to facilitate more systematic design-time testing and reasoning about security properties.

References

[1] J. S. Dwoskin and R. B. Lee, "Hardware-rooted Trust for Secure Key Management and Transient Trust," in *ACM Conference on Computer and Communications Security (CCS)*, (Alexandria, VA), pp. 389–400, October 2007.

[2] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. S. Dwoskin, and Z. Wang, "Architecture for Protecting Critical Secrets in Microprocessors," in *ISCA*, pp. 2–13, 2005.

[3] "OKL4 Microkernel." Open Kernel Labs, http://www.ok-labs.com.

[4] G. Popek and R. P. Goldberg, "Formal Requirements for Virtualizable 3rd Generation Architectures," *Communications of the A.C.M.*, vol. 17, no. 7, pp. 412–421, 1974.

[5] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," *IEEE Computer*, vol. 38, no. 5, pp. 39–47, 2005.

[6] "VMware Workstation." VMware Inc., http://www.vmware.com.

[7] 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 *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, (New York, NY, USA), pp. 164–177, ACM, 2003.

[8] D. Mihocka and S. Shwartsman, "Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure," in 1^{st} *Workshop on Architectural and Microarchitectural Support for Binary Translation in ISCA-35*, (Beijing), June 2008.

[9] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX Annual Technical Conference, FREENIX Track*, pp. 41–46, 2005.

[10] G. E. Suh, D. Clarke, B. Gassend, M. v. Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, (Washington, DC, USA), p. 339, IEEE Computer Society, 2003.

[11] R. Elbaz, D. Champagne, R. B. Lee, L. Torres, G. Sassatelli, and P. Guillemin, "Tec-tree: A low-cost, parallelizable tree for efficient defense against memory replay attacks," in *CHES*, pp. 289–302, 2007.

[12] D. Champagne, R. Elbaz, and R. B. Lee, "The reduced address space (ras) for application memory authentication," in *ISC*, pp. 47–63, 2008.

[13] W. E. Hall and C. S. Jutla, "Parallelizable authentication trees," in *Selected Areas in Cryptography*, pp. 95–109, 2005.

[14] D. Lie, C. A. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. C. Mitchell, and M. Horowitz, "Architectural Support for Copy and Tamper Resistant Software," in *ASPLOS*, pp. 168–177, 2000.

[15] G. E. Suh, D. E. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-evident and Tamper-resistant Processing," in *Intl. Conference on Supercomputing (ICS)*, pp. 160–171, 2003.

[16] M. Gomathisankaran and A. Tyagi, "Architecture Support for 3D Obfuscation," *IEEE Trans. Computers*, vol. 55, no. 5, pp. 497–507, 2006.

[17] Truscted Computing Group, *Trusted Platform Module Specification Version 1.2 Revision 103*, July 2007. https://www.trustedcomputinggroup.org/.

[18] T. E. Levin, J. S. Dwoskin, G. Bhaskara, T. D. Nguyen, P. C. Clark, R. B. Lee, C. E. Irvine, and T. Benzel, "Securing the dissemination of emergency response data with an integrated hardware-software architecture," in *TRUST*, pp. 133–152, 2009.

[19] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold boot attacks on encryption keys," in *USENIX Security Symposium*, pp. 45–60, 2008.

[20] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer*, vol. 35, pp. 59–67, February 2002.

[21] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security Symposium*, pp. 321–336, 2004.

[22] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection," in *In Proc. Network and Distributed Systems Security Symposium*, pp. 191–206, 2003.

[23] B. Payne, M. Carbone, M. Sharif, and W. Lee, "Lares: An architecture for secure active monitoring using virtualization," pp. 233–247, May 2008.

[24] L. Litty, H. A. Lagar-Cavilla, and D. Lie, "Hypervisor support for identifying covertly executing binaries," in *SS'08: Proceedings of the 17th conference on Security symposium*, (Berkeley, CA, USA), pp. 243–258, USENIX Association, 2008.

[25] B. Payne, M. de Carbone, and W. Lee, "Secure and flexible monitoring of virtual machines," pp. 385–397, Dec. 2007.

[26] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," in 15^{th} *USENIX Security Symposium*, (Vancouver, Canada), July 2006.

[27] V. Scarlata, C. Rozas, M. Wiseman, D. Grawrock, and C. Vishik, *Trusted Computing*, ch. TPM Virtualization: Building a General Framework, pp. 43–56. Springer, 2008.

[28] M. Strasser, H. Stamer, and J. Molina, "Software-based TPM Emulator." http://tpm-emulator.berlios.de.

[29] W. A. Hunt, "Mechanical Mathematical Methods for Microprocessor Verification," in *Intl. Conference on Computer Aided Verification (CAV)*, pp. 523–533, 2004.

[30] S. Ray and W. A. Hunt, "Deductive Verification of Pipelined Machines Using First-Order Quantification," in *Intl. Conference on Computer Aided Verification (CAV)*, pp. 31–43, 2004.

[31] H. Iwashita, S. Kowatari, T. Nakata, and F. Hirose, "Automatic test program generation for pipelined processors," in *Intl. Conference on Computer Aided Design (ICCAD)*, pp. 580–583, 1994.

[32] R. C. Ho, C. H. Yang, M. Horowitz, and D. L. Dill, "Architecture Validation for Processors," in *ISCA*, pp. 404–413, 1995.

[33] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray, "A Survey of Hybrid Techniques for Functional Verification," *IEEE Design & Test of Computers*, vol. 24, no. 2, pp. 112–122, 2007.

[34] S. S. Moore, "Symbolic Simulation: An ACL2 Approach," in *Formal Methods in Computer-Aided Design*, pp. 334–350, 1998.

[35] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle's Logics: HOL*.