Jeffrey Dwoskin, Mahadevan Gomathisankaran, Ruby Lee. "Framework for Design Validation of Security Architectures", Princeton University Department of Electrical Engineering Technical Report CE-L2008-013, Nov 17, 2008.

Framework for Design Validation of Security Architectures

Jeffrey Dwoskin Mahadevan Gomathisankaran Ruby Lee

Abstract

New security architectures are difficult to prototype and test. They require interactions between hardware, operating systems, and applications, making them hard to simulate and monitor. We have designed and prototyped a testing framework using a virtualization platform which emulates the behavior of new hardware security architecture in the virtual CPU, and performs a wide range of hardware and software attacks on the system under test. Our testing framework significantly speeds up development of the testing environment and infrastructure, and provides APIs for launching attacks and monitoring the effects of an attack on the hardware and software layers, which is especially convenient during the design and validation phases for new hardware-software architectural solutions. We have used our testing framework to test the trust chain of the SP architecture [1] as an example.

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 put low-level security mechanisms into the hardware which the software layers can rely upon for a wide-range of applications. The resulting architecture is a combination of hardware and software components which are difficult to test together. Testing must be done during the design time to give 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. To address this problem of design-time security validation, we propose a new testing framework for hardware-software security architectures. Our framework provides a controlled environment to emulate the behavior of new hardware components with a full software stack running in a virtualization environment, with which coordinated security attacks can be performed and observed.

In a new architecture, security mechanisms may be added to the processor at the lowest software-visible layers, yet the effects on system operation reach up into the operating system, middleware, and application software, and can even span networked systems. Each of these layers will use abstractions of the lower layers and make assumptions about their security properties. Validation requires that each layer be modeled and simulated together to study the interactions of the components and their effects on the overall operation of the system. This is unlike traditional computer architecture where performance and power optimizations can usually be designed and tested with layer-specific measurements that focus on the new components. Instead, our framework must emulate all hardware and software layers simultaneously, while coordinating simulated attacks by observing and controlling activity at each system layer. Furthermore, 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.

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) which coordinates monitoring of, and attacks on, the SUT. We build on top of existing virtualization technology, which allows us to run a full set of commodity software efficiently. In a virtualized system, a Virtual Machine Monitor (VMM) creates multiple virtual machines that run on a single physical host machine [2, 3, 4]. Typically the virtual machines are nearly identical to the host machine, however, we modify the VMM to emulate the new security hardware features of the SUT. The SUT and the TS are separate virtual machines, so that the environment of the SUT is as realistic as possible — it runs a commodity operating system (Linux), commodity user applications, and any new application software using the new security mechanisms.

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 virtual machine coordinates events and attacks from both hardware and software components in the SUT.

{jdwoskin, mgomathi, rblee}@princeton.edu

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 may go beyond known penetration methods. We do this by mapping attacks to their *impacts* on the SUT. Hence, a key advantage 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 new hardware-software architectures for security properties, leveraging VMM technology;
- a realistic environment using commodity operating systems for testing different applications using the new security mechanisms;
- a flexible, fast, and low-cost method for emulating HW features in the VMM for the purpose of design validation — without the need for costly and time-consuming fabrication of HW prototypes;
- the ability to simulate the impact of very powerful attackers for "future" attacks; and
- the application of our framework towards the validation of the security properties of the SP architecture [1, 5].

The rest of our 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 methodologies for its testing. Section 5 offers results and sample security attacks. We discuss related work in Section 6 and conclude in Section 7.

2 Hardware-Software Security Architectures

New security architectures can take many forms. 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 security protection which cannot be bypassed, and the software provides flexibility to implement different applications and usage scenarios, with different security policies.

Figure 1 shows a typical system with the addition of a trusted software application and new trusted hardware

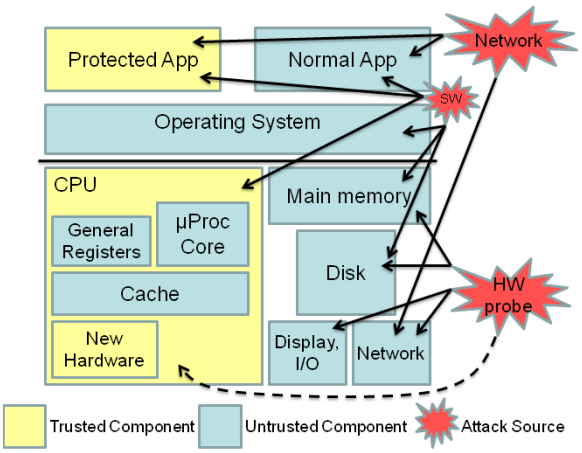

Figure 1: Threats and Attacks on Security Architectures

security mechanisms added to the CPU (e.g. new instructions, faults, registers, and hardware mechanisms). Sometimes, as shown in the figure, 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 [6]), but not the entire operating system.

The figure also shows the sources of attacks that we consider in our testing framework. First, malware or exploitable software vulnerabilities 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, if adversaries get physical possession of a device, they can perform hardware attacks, such as directly accessing data on the hard disk, probing physical memory, intercepting data on the display and I/O buses. Very powerful attackers may even be able to probe parts of the processor chip.

Third, network attacks can be performed with either software or hardware access to the device, or with access to other parts of the network. 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. Our testing framework provides hooks into each relevant system component, and additionally allows information and events at each level to be correlated to

emulate the most knowledgable attacker.

3 Testing Framework

The design goals of our testing framework are to create a generic platform that can emulate and test a wide range of security architectures in a realistic environment. It must test the architecture-specific application software running on top of full commodity operating systems. The testing framework should also allow easy monitoring of software and hardware events, and allow modification of software and hardware state to launch attacks on the system.

3.1 Architecture

A virtual machine monitor (VMM) is the software which creates Virtual Machines (VMs), efficiently providing an execution environment in each VM which is almost identical to the original machine [7]. By modifying an existing VMM, we can augment the virtual machine to have the additional hardware features of our new security architecture in addition to those of the base machine. Since we are still using efficient virtualization software, we can run commodity operating systems and applications in our modified virtual machines.

Our Testing Framework is divided into two systems, as shown in Figure 2, the System Under Test (SUT) and the Testing System (TS), each running as a virtual machine on our modified VMM. The SUT machine simulates the system being designed and tested for the new security architecture, as if it were actually built and programmed. The TS machine simulates the attacker, who is trying to violate the security properties of the SUT. Through a variety of hooks in our modified VMM and guest OS, the TS has full access to the internal operations and state of the SUT.

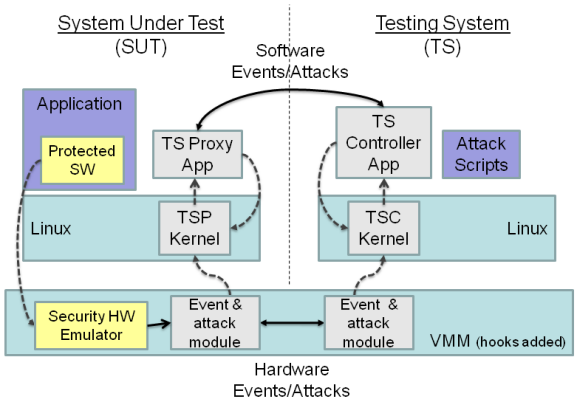

Figure 2: Testing Framework Design

In isolation, the SUT is meant to behave as closely as possible to a real system which has the new security architecture. It must behave as if it has all of the new security primitives available in hardware, 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. However, for the purposes of the testing framework, we add a software component, the TS Proxy (TSP), to the SUT to simulate the effect of a compromised operating system for launching software attacks, allowing the OS to be fully controllable by the TS — as if it has been compromised. It is still necessary to keep the TS as a separate virtual machine so that the TS Proxy in the SUT need not be invoked to launch a virtual hardware attack.

For the purposes of fully exploring the design space of new security architectures, the TS has the additional capability to be a *super-attacker*. The testing framework itself is ignorant of the threat model of the system being designed, and instead 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. For initial design and implementation of the system, the TS can act as a debugger, able to see the lowest-level micro-architectural behaviors of the hardware features and 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.

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.

3.2 Testing Framework Modules

The main components of our Testing Framework shown in Figure 2 are: the Testing System Controller (TSC), the Testing System Proxy (TSP), the Events and Attack

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 Architecture Hardware	Execution of new instructions, Triggering of new faults, Accesses to new registers.	Read/write new registers & state, Read/write protected memory plaintext.

module, the Hardware Emulation Module, the Testing Application, and the Attack Scripts. The TSC and TSP are each divided into a user-level application and a component in the OS kernel.

A System Under Test is comprised of three layers, namely: Application, Operating System, and underlying Hardware. The Testing System should be able to both monitor the events and modify the state in all three of these layers. The modified VMM monitors and controls the hardware, while the the TSP monitors and controls the applications and OS.

The TS and SUT components communicate to exchange information about events occurring in the SUT and to implement attacks from the TS by modifying system state in the SUT. The user level components communicate with the kernel components through *system calls*, and the kernel components use *signals* to communicate asynchronously with the user level components. The VMM communicates with the guest OS kernel asynchronously through a new virtual hardware interrupt which we have defined.

Testing System Controller The TS Controller, running on the TS, is the aggregation point that receives events from all three layers in the SUT. It receives OS and Application level (software) events from the TS Proxy via a network channel and receives hardware events from the VMM through a channel from its kernel-level component. It provides APIs to the Attack Scripts which can monitor or wait for specific events and adaptively mount a coordinated attack on the SUT.

Testing System Proxy This module acts as a proxy for the TSC running on the SUT itself. It controls the application to be tested, and uses its corresponding kernel-level component to control and monitor OS behavior and the OS-level abstractions used by the testing application, including system calls, virtual memory, file systems, sockets, etc. The TSC communicates with the TSP over TCP/IP on the virtual network which connects the VMs. Note that the TSP is used to simulate the impact of a compromised OS in the SUT, whereas in the framework the OS itself is not actually compromised.

Event and Attack Module This module provides hooks to both observe and control the internal machine state of the SUT from inside the VMM. The TS Controller registers with this module for the set of events to be monitored, and uses the APIs provided by this module to observe or modify (i.e., attack) the state. Events that can be watched include: interrupts/faults specific to the secure architecture, execution of new instructions, execution of new behaviors (e.g. modified hardware interrupt handling), regular x86 interrupts, etc. Machine state that is exposed includes CPU registers, physical memory, and the new registers for the secure architecture. Note that unlike a normal attacker, this model is very powerful and can stop the machine precisely for observation or modification before or after any specific instruction execution or other detected

hardware event. In Figure 2, we show this module split within the VMM into two separate components for each VM to accommodate the split-VMM structure present in some virtualization software.

Hardware Emulation Module This component, added to the VMM, emulates the behavior of the new security architecture to be tested. It enhances the base micro-architecture with new instructions, functional units, registers, interrupts/fault behaviors and memory access behavior. New instructions are exported to the software layers through hypercalls, which allow application or OS layers to directly invoke the VMM.

Testing Application The Testing Application is the actual implementation of the application software that would be run on the secure architecture. It should demonstrate the features of the architecture and possibly interact with a local user to provide access to secrets and data which the architecture is protecting. The testing framework has hooks that allow the application to report any internal events such as function calls or decision points. The usage of these hooks are optional and is required only if the application level events need to be monitored for the attack script.

Attack Scripts The Attack Scripts reside on the TS and specify how particular attacks are executed on the SUT. They provide step-by-step instructions for monitoring events and dynamically responding to them in order to successfully launch attacks, or detect that an attack was prevented by the security architecture. The scripts act like a state-machine, acting on hardware and software events which are aggregated by the TS. Scripts can be written to form a library of generic attacks, that can be used to attack any application. Alternatively they can be specific to the behavior of the testing application, written by the user of the testing framework. The TS Controller reads and executes these scripts and implements the communication mechanisms and control of the SUT as needed.

In addition to these components, the TS hosts and emulates any other trusted third parties required by the system, possibly intercepting and modifying their network traffic as an additional source of events and attacks.

By using two separate virtual machines to host the components of our testing framework, we are able to keep the SUT as close as possible to the real security architecture being tested, including all of its software components, both trusted and untrusted. We do not need to trust the SUT's guest OS for the sake of correctness of the emulation of new hardware security features. In fact, the SUT's guest OS can launch real attacks on the system (as controlled by the TS).

We also have the ability to use the separate VM to launch hardware attacks asynchronously from the TS; the TS can pause the execution of the entire SUT VM and continue running the attack scripts to inject attacks at any instruction boundary.

3.3 Events and Attacks

The testing framework is designed to expose events from the three layers of the SUT (application software, OS, and hardware) to the TS, and to allow the TS access to the state of the SUT to launch attacks. Table 1 lists various events and attacks for each layer of the system. The hardware layer is further classified into events and attacks from the base hardware (x86 architecture in our work) and the new emulated security architecture.

The hardware layer is accessed by the TS Controller through the event and attack module in the VMM, which communicates events across an inter-VM channel inside the VMM to relay events and attacks between the SUT and TS. This channel is used to communicate with the TS during execution of a single instruction or HW operation in the SUT, possibly changing the result of that operation.

The software layers are accessed through the TS Proxy, which hooks into the OS kernel through its kernel module, and to the testing application using its user-mode component. The TSP can 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 also 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 or attack to the TS Controller.

The SUT modules in both the hardware and software layers capture their relevant events and signal the TS Controller. The controller, following its attack scripts, can then attack the SUT based on those events. Table 2 lists the API which the TS Controller exports to the attack scripts for event handling and the basic attack mechanisms.

The attack mechanisms we provide are designed around the *impact* that attacks have on the state of the SUT. The security properties and attacks considered in the threat model of a given security architecture may instead focus on the method of penetration used to access that state. For our testing framework, we enable a powerful attacker without regard to how penetration occurs. 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. Thus an architecture can be tested

Table 2: TS Controller API for Attack Scripts

Function	Description
`h ← INIT()`	Initializes the TSC and returns a handle *h* to access the resources.
`FREE(h)`	Free the resources for the handle *h*.
`EXECUTE(h,app,params)`	Execute the application *app* on SUT with the given parameters *params*.
`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.
`ACCESSREG(h,type,r/w,buf)`	Reads/writes (*r/w*) the general registers or SP registers (*type*) of the SUT to/from *buf*.
`ACCESSMEM(h,v/p,r/w,addr,sz,buf)`	Reads/writes (*r/w*) *sz* bytes from virtual or physical memory (*v/p*) of the SUT at address *addr* to/from the buffer *buf*.
`INTERRUPT(h,num)`	Trigger a virtual hardware interrupt number *num* on the SUT.

against unknown attacks as well as known methods of penetration by considering the most powerful attacker.

3.4 Implementation

We implemented our testing framework on VMware's virtualization platform [8], including all of the modules in Section 3.2 and events and attacks at each system layer. The Hardware Emulation Module and VMM Event & Attack Module required modifying the source code of the VMware VMM. The kernel components of the TSP and TSC are implemented as Linux kernel modules. The TSP application is implemented as a Linux user process and controls the execution of the Testing Application. The TSC application is implemented as a static library which is called by the Attack Scripts. Upon initialization, the TSC connects with the TSP over the virtual network, and the TSP registers with the Event and Attack Module via a hypercall to the VMM.

As a sample security architecture, we implement the SP architecture, described in Section 4. Hence, the Hardware Emulation Module emulates the SP architecture including its master secrets, 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 4.4. Our Testing Application uses this library to exercise the software, and in turn, the SP hardware.

4 SP Architecture and Emulation

For the rest of this paper, we will use the Secret Protection (SP) architecture [1][5] to demonstrate our framework. We have chosen SP because it was specifically designed to be as simple as possible to facilitate verification, while providing robust security utilizing both hardware and software components. At the same time, it uses a somewhat controversial and unproven design, skipping layers in the trust chain by using hardware to directly protect an application without trusting the underlying operating system. To be accepted by the security community, SP's security properties need to be validated, demonstrating that it can protect the confidentiality and integrity of cryptographic keys in its persistent storage which in turn protect sensitive user data. Furthermore, it is important to write and test many secure software applications to run on SP in a realistic environment, where the untrusted OS can be a source of attacks. To the best of our knowledge no single tool exists to test hardware-software security architectures like SP.

Our testing framework first emulates SP's hardware features using modifications to the VMM. This allows us to directly test the primitives and guarantees made by the SP hardware itself and how those primitives interact with software components. Additionally, we can consider the hardware emulation as a constant and test a

range of software implementations which take advantage of the SP hardware. We wish to test how SP hardware protects its software, how trusted software can be written to take advantage of the hardware guarantees, and how trusted software interacts with untrusted software on an SP device.

4.1 Secret Protection 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 the equivalent of the trusted computing base (TCB) for the application. Rather than protecting an entire application, only the security-critical parts of an application 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.

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 maintains its state using new processor registers. It 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 3, SP uses two on-chip master secrets: the Device Root Key (DRK) and the Storage Root Hash (SRH). For code integrity, the DRK 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. 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 written 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. 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 will encrypt the contents of the general registers in place, and keep a hash of the registers on-chip in the interrupt registers to verify before decryption when the TSM is resumed.

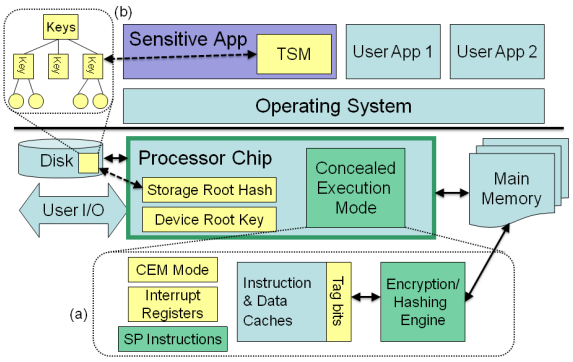

Figure 3: Secret Protection (SP) Architecture. Enlargements show (a) the additional CEM hardware and (b) the application secrets protected by the TSM.

Secret data belonging to the application is also protected in persistent storage by encryption using keys protected by the TSM. SP allows a TSM to derive new keys from the DRK using a new hardware instruction, *DRK_DeriveKey*. These derived keys are used by the TSM to protect the confidentiality and integrity of its persistent data. Other software, including the OS, may not use the *DRK_DeriveKey* instruction. The TSM is also the only software that can read and write the SRH 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 master secrets); new instructions; hardware mechanisms for code integrity checking, secure memory, and interrupt protection; and new hardware faults which these mechanisms generate.

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 microarchitecture effects that do not affect program correctness, 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 can 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, binary translation of code, mapping of virtual memory translations, and virtualization

of hardware devices.

To emulate the SP architecture, a number of components must be added to the virtual machine as part of the Hardware Emulation Module:

Protected Mode SP's Concealed Execution Mode [5] requires new registers which are added to the virtual CPU, including the registers holding SP's master secrets (DRK and SRH), mode bits, and interrupt handling registers. New SP instructions are currently modeled as hypercalls, where the TSM running in the SUT is able to directly invoke the emulation module without going through the guest OS. Any trapping instruction would serve this purpose. Alternatively, binary translation can be used by the VMM for TSM code, which can be written with new unused opcodes to implement new instructions.

Interrupts and SP Faults The SP architecture changes the hardware behavior when interrupts occur when in protected mode. Since the VMM already needs to emulate interrupt behavior, we can simply detect that an interrupt has occured 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, which includes verifying the hash and decrypting the general registers before resuming the TSM. Additionally, new SP faults can be triggered by both hardware events and new SP instructions. When the emulated hardware generates a new fault, it first reports to the TSC, 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 to work on virtual memory pages instead of cache lines, since the VMM does not intercept cache memory behavior. 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.

We also change how the hardware determines when to access secure memory, so that the VMM need not trap every individual memory access, causing performance degradation. Rather than using new instructions to access secure memory and regular memory, we allow the TSM to define regions of memory which are always accessed securely — an abstraction which fits the embedded SP model [9]. This simplifies writing example TSMs since an entire data structure or stack segment can be protected, and the compiler need not be modified to emit new instructions for memory accesses. The TSM simply declares which memory ranges should be protected upon application launch.

Code Integrity SP verifies the integrity of the TSM code dynamically during execution. 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. We model this using the VMM's binary translator to execute the TSM code. The implementation tags secure instructions as code fragments in the binary translator cache. SP's instruction signing (with keyed hashes) is similarly generated and stored across larger regions of code (for emulation efficiency) and saved in a separate file.

4.3 Other Architectures

While this paper focussed on testing the hardware and software mechanisms of the SP architecture, our testing framework is by no means limited to this architecture. Other security architectures such as XOM [10], AEGIS [11, 12, 13] and Arc3D [14, 15] 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 [16] modifies hardware to protect all software layers and provide cryptographic services. However, 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 however requiring a different set of modifications to the VMM to implement a virtual TPM device.

4.4 Remote Key-management TSM

SP's Concealed Execution Mode provides code integrity, a secure execution state, and secure memory for the TSM. The TSM in turn must be written to support applications that use sensitive data. In software, the TSM uses the master secrets (in the DRK and SRH registers) to protect the confidentiality and integrity of persistent data, and to offer interfaces to unprotected applications that need to use this data. It must also manage secure communication with the user (including authentication) and with third parties over the network.

A TSM can be written for many different usage scenarios. As an example we look at the remote key-management scenario from Authority-mode SP [1]. We use this TSM to test various aspects of the SP architecture, from the hardware features in the virtual processor (which are now protecting a real TSM), to the software and network interfaces. We demonstrate

how the security-critical components of the remote key-management software can be partitioned off into a TSM which can be used by other software. Using the framework, we then subject our TSM to a suite of attacks to see if the desired security properties are preserved.

For remote key-management, we consider a central 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 data, consisting of secrets and cryptographic keys that protect these 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.

Testing SP requires testing TSM software in combination with SP hardware mechanisms. The TSM provides protection for the application's critical data and is itself protected by the SP hardware. We can attack the robustness of the TSM's memory usage, persistent storage, network protocols, and software interfaces. We thereby use our testing framework as a testbed for secure software development in addition to the secure hardware — both the software TSM and the hardware SP features are required to provide SP's confidentiality and integrity guarantees.

4.5 Testing Example

We now give an example of how the TSC can test an application and TSM running on the SUT. Table 3 shows, on the left, the protected application using a TSM (`TSMapp`) and a corresponding unprotected application (`unsafeApp`). The attack script is shown on the right. The attack simulates a compromised OS trying to access confidential data currently being used by a TSM (in this case, a secret key used for encrypting sensitive data) by interrupting the TSM during its execution. The `TSMapp` uses SP instructions to enter and exit protected mode (*BEGIN_TSM* and *END_TSM*) and to generate a key (*DRK_DeriveKey*). The `unsafeApp` is instrumented to explicitly notify the TSP (with *TSP_Notify*)when it generates a key.

The application first generates a new key, for *TTP_KeyID*, which it uses to encrypt data. It then sends the encrypted data over the network to the remote Trusted Third Party (TTP). In the first line of the attack script, the TSC launches the testing application, either using the `TSMapp` (where we expect the attacks to fail) or the `unsafeApp` (where the attacks should succeed). The script then watches for a particular event to occur in the application, in this case the generation of a key. The application executes normally until it triggers this event which the attack script is monitoring.

Steps 1-5 in figure 4, show the steps taken by the testing framework to detect an event. In step 1, the TSM uses an SP instruction to generate a new derived key from the DRK master secret. This requires making a hypercall down to the SP HW Emulation Module to emulate the instruction and produce results into the general registers. In step 2, the SP HW notifies the Event & Attack Module of this HW event (the execution of an SP instruction). In step 3, the event is passed up to the TSC kernel module via a virtual HW interrupt/IRQ, and the VMM freezes execution of the entire SUT. In step 4, the TSC is notified of the event via a Linux signal. In step 5, the TSC returns the event to the attack script which was waiting for notification of this key generation event. It can now continue executing the script to perform any hardware attacks while the SUT is frozen. These steps are repeated for every hardware event monitored by the TSC.

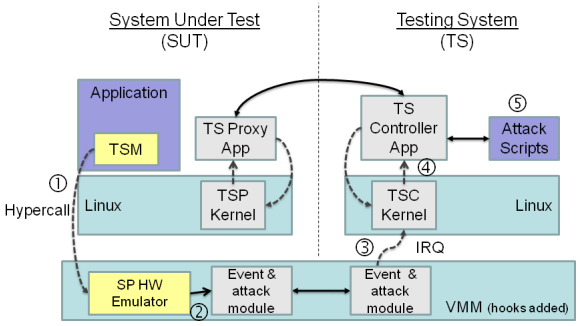

Figure 4: Example Attack Sequence

The script resumes the TSM and then attacks by instructing the TSC to generate an interrupt during the TSM's execution. This triggers the SP HW's interrupt protection mechanism, encrypting the general registers and secure memory, and suspending protected mode for the TSM. Next the script, acting like a corrupted operating system's interrupt handler, attempts to read and modify the TSM's general registers. It overwrites the generated key with a known key. With SP protection, it can only read an encrypted version of the registers, and upon resuming from the interrupt, the incorrect general register values will cause an SP fault. If the same attack is performed using the unprotected application without SP and its TSM, the attacker would obtain the generated key and the user of the SUT would remain unaware of the modification to its registers, encrypting with a key known to the attacker.

Table 3: Example Application and Attack Script for Eavesdropping and Spoofing Attack using Interrupt

Application with TSM (`TSMapp`)

```
BEGIN_TSM
...
Reg1 ← DRK_DeriveKey(TTP_KeyID)
Ciphertext ← Encrypt(Reg1, &SecureMem, sz)
END_TSM
Network_Send(TTP, Ciphertext, sz)
...
```

Application without TSM (`UnsafeApp`)

```
// No SP protection
...
Reg1 ← App_Generate_Key(TTP_KeyID)
TSP_Notify(App_Keygen, Reg1)
Ciphertext ← Encrypt(Reg1, Data, sz)
Network_Send(TTP, Ciphertext, sz)
...
```

Attack Script

```
EXECUTE(TSMapp, params) // or UnsafeApp

// Wait for key generation
EVENTADD(DRK_DeriveKey)
EVENTADD(App_Keygen)
EVT ← WAIT()
ACCESSREG(SP, r, SPRegs)
SPKey ← SPRegs.DerivedKey
AppKey ← EVT.param1

// Trigger an interrupt
INTERRUPT(03); CONT()
EVT ← WAITFOR(Interrupt)

// Attack confidentiality of general registers (GRs)
ACCESSREG(GR, r, GRegs)
if SPKey = GRegs.R1 OR AppKey = GRegs.R1 then
  print "Register Confidentiality Attack Succeeded"
else
  print "Register Confidentiality Attack Failed"
end if

// Attack integrity of general registers (GRs)
ACCESSREG (GR, w, KnownKey); CONT()
EVT ← WAITFOR(SPFault_All)
if EVT = SPFault_Reg_Integrity then
  print "Register Integrity Attack Failed"
else
  print "Register Integrity Attack Succeeded"
end if
```

5 Testing of SP

As we have shown in the previous section, it is possible to use our new testing framework to perform a wide range of tests and attacks on the SUT and the security architecture it emulates. We now look more closely at how we can use the framework to validate the security properties of the SP architecture in particular through a range of tests.

SP claims to provide a number of security properties, of particular importance are the confidentiality and integrity of: master secrets (DRK & SRH), TSM execution, secure data, and usage policies. We must demonstrate that each security property is enforced through the combination of trusted and verified TSM software and trusted SP hardware.

The main goal of the remote key-management TSM is to enforce policies on secure data, as dictated by the Authority, in an SP device. The following steps establish the trust chain of the architecture:

1. The authority binds the TSM code to the SP device (and its master secrets, which cannot leave the device) and provides secret data to the TSM on that device.
2. The TSM executes in Concealed Execution Mode.
3. The TSM binds security policies and secure data to the keys derived from SP's master secrets on the device.

We need to verify each of these steps by launching attacks to test the SP's trust chain.

In Table 4 we show a list of attacks which test these aspects of the architecture, using our framework, the emulated SP hardware, and our TSM. Each row in the table indicates a specific attack we have implemented, where 'PASS' indicates that the architecture successfully protected against the attack.

When the authority provides secret data, it needs to be sure that it can only be used on the designated device and only using the TSM it provided which is trusted to enforce its security policies. Therefore it must not be possible to replace that TSM with another, even

when an adversary has physical possession of the SP device. Therefore, for step 1, we attempt to replace the TSM by installing a new DRK which signs different TSM code, however it then cannot access the authority's data which was protected by the original DRK. Since the DRK never leaves the processor chip and cannot be extracted, there is no other way to get access to the original DRK.

In step 2, the integrity of a TSM must be protected both before and during execution. This involves testing SP's hardware protection of the confidentiality and integrity of all intermediate data that it generates. This intermediate data can be attacked while in general registers during a processor interrupt or in secure memory at any time while the TSM is not running. Additionally, the SP master secrets themselves must only be used by the TSM itself and not other software. The code integrity of the TSM must be protected both on disk and during execution.

Finally, step 3 covers the proper design and implementation of the TSM itself. The TSM must store cryptographic keys, security policies, and secure data in its persistent secure storage, which it protects using SP's underlying hardware mechanisms (DRK & SRH). We test the confidentiality and integrity of the storage itself, plus the TSM's use of the storage and its enforcement of the policies on accesses to the data. Adversaries might attack the TSM's persistent data offline when stored on disk or after the TSM has loaded it into secure memory.

Our system implements the SP hardware mechanisms, a full TSM providing an API to the testing application, and a suite of attacks which test both the software and hardware components using our new testing framework. This is a first step towards the complete validation of the design of the SP architecture together with its applications, and provides a framework for further testing the architecture with other software and attacks.

6 Related Work

Our testing-framework of a system architecture solution has multiple related fields of research. Our emulation of security architecture can be compared with hardware simulation research, our validation mechanisms can be compared with formal and hybrid verification methodologies, and our virtualization architecture can be compared with systems virtualization solutions.

Micro-architectural simulators like Simplescalar [17] are cycle-accurate and hence can be very useful in estimating performance metrics, but they can not simulate a realistic software system with full commodity OS. Thus it is impossible to test the security critical interactions of a software-hardware security solution with such a simulation architecture. Our emulation architecture, in using the existing virtualization technology, enables reasonable performance while allowing our SUT to provide a realistic software stack. Other simulation and emulation environments are available, such as Bochs [18], Qemu [19], and Xen [20]. Where these environments provide sufficient software performance and granularity of hardware emulation, they could be used in place of VMware to implement our framework as designed and described in this paper.

The efforts by IBM [21], Intel [22] and others [23] provide the functionality of a virtual TPM device to software, even when the physical device is not present. In contrast, we not only emulate the new hardware but also hook into the 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 mathematics is used to write specifications for computer hardware and 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 [24, 25, 26, 27]. The complexity of these formal verification mechanisms led to the use of hybrid techniques [28] which uses some formal as well as informal techniques. Some formal methods of verification are methods using theorem provers (ACL2 [29], Isabelle/HOL [30]), model checkers, satisfiability solvers, etc. Some informal techniques used in practice are control circuit exploration, directed functional test generation, automatic test program generation, heuristic-based traversal, etc. The formal and hybrid techniques try to verify the hardware and software separately, unlike our holistic verification of a software-hardware system.

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 piece by piece. The complexity of both specification and verification explodes exponentially with addition of more and more pieces to be tested. In our approach we model both the security critical hardware and software as a single entity, 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 a new system during the design phase, and draw useful conclusions about the operation of the new architecture and

Table 4: Verifications and Attacks

#	Security Property	Attack	Mechanism	Results
1	Binding TSM to Secrets	Attack the persistent storage protected by one TSM from another TSM when the DRK is changed.	Physical possession of SP device	PASS
2	Confidentiality & Integrity of Secure Memory	Attacks secure memory outside protected mode through eavesdropping, spoofing, and splicing.	Virtual/physical memory access to TSM pages	PASS
	Confidentiality & Integrity of General Registers	Attacks on registers during an interrupt of a TSM through eavesdropping, spoofing, splicing, and replay.	OS access to saved process context	PASS
	Access HW Master Secrets	Attack the SP master secrets (DRK & SRH) from outside protected mode	Unprotected application execution	PASS
	Integrity of TSM Code	Attacks on TSM code before and during execution through spoofing and splicing.	Disk access or access to text pages in memory	Not Yet Tested
3	Confidentiality & Integrity of Persistent Secure Storage	Attack the TSM's secure storage, created using DRK and SRH, on disk or when loading into memory.	Disk access or access to data pages in memory	PASS
	Binding Security Policies to Keys	Attack the TSM's enforcement of security policies on the use of keys in its persistent storage, and the secrets those keys protect.	Disk access or access to data pages in memory	PASS

its software interactions. It also enables testing various software applications using the new security hardware, unlike a new hardware prototype which does not have an OS running on it.

Our framework serves as a rapid 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 mechanisms and high-level application behavior. As a result, a full range of attacks are realizable on the hardware, operating system, and applications.

Finally, we implement the SP architecture in our framework and test its security mechanisms thoroughly. We demonstrate that the full trust chain of the architecture holds up under attack, using a real TSM for remote key-management.

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. P. Goldberg, *Architectural Principles for Virtual Computer Systems*. PhD thesis, Harvard University, 1972.

[3] R. Goldberg, "Survey of Virtual Machine Research," *IEEE Computer*, pp. 34–45, June 1974.

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

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

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

[7] 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.

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

[9] J. Dwoskin, D. Xu, J. Huang, M. Chiang, and R. Lee, "Secure Key Management Architecture Against Sensor-node Fabrication Attacks," in *Globecom*, 2007.

[10] 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 *Intl. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 168–177, 2000.

[11] 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.

[12] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas, "Design and Implementation of the AEGIS Single-Chip Secure Processor Using Physical Random Functions," in *Intl. Conference on Computer Architecture (ISCA)*, pp. 25–36, 2005.

[13] G. E. Suh, C. W. O'Donnell, and S. Devadas, "Aegis: A Single-Chip Secure Processor," *IEEE Design & Test of Computers*, vol. 24, no. 6, pp. 570–580, 2007.

[14] M. Gomathisankaran and A. Tyagi, "Arc3D : A 3D Obfuscation Architecture," in *High Performance Embedded Architectures and Compilers (HiPEAC)*, pp. 184–199, Springer, 2005.

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

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

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

[18] 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.

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

[20] 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.

[21] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn, "vTPM: Virtualizing the Trusted Platform Module," Tech. Rep. RC23879, IBM, February 2006.

[22] 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.

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

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

[25] 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.

[26] 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.

[27] R. C. Ho, C. H. Yang, M. Horowitz, and D. L. Dill, "Architecture Validation for Processors," in *Int. Symposium on Computer Architecture (ISCA)*, pp. 404–413, 1995.

[28] 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.

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

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