Scalable architectural support for trusted software

Conference Paper · January 2010

DOI: 10.1109/HPCA.2010.5416657 · Source: DBLP

Ruby B. Lee
Princeton University

Scalable Architectural Support for Trusted Software[1]

David Champagne and Ruby B. Lee
Princeton University
{dav, rblee}@princeton.edu

Abstract

We present Bastion, a new hardware-software architecture for protecting security-critical software modules in an untrusted software stack. Our architecture is composed of enhanced microprocessor hardware and enhanced hypervisor software. Each trusted software module is provided with a secure, fine-grained memory compartment and its own secure persistent storage area. Bastion is the first architecture to provide direct hardware protection of the hypervisor from both software and physical attacks, before employing the hypervisor to provide the same protection to security-critical OS and application modules. Our implementation demonstrates the feasibility of bypassing an untrusted commodity OS to provide application security and shows better security with higher performance when compared to the Trusted Platform Module (TPM), the current industry state-of-the-art security chip. We provide a proof-of-concept implementation on the OpenSPARC platform.

1. Introduction

Many applications are involved in handling sensitive or secret information, e.g., in financial or medical transactions. Often, only certain parts of a large application require extra protection, which we call *security-critical tasks* encapsulated in *trusted software modules*. While an application writer is highly motivated to ensure the security of these trusted software modules, any application-level protection today can be undermined if the underlying operating system is compromised. Unfortunately, this is typically a commodity operating system (OS) over which the writer of a trusted application has no control. The OS is not only all-powerful over applications, but also large, complex, dynamically extendible and frequently updated, making it very hard to verify and highly vulnerable to attacks. Can the hardware provide direct alternative support for an application's trusted software modules, even in the presence of a compromised OS?

To add to the challenge, *hardware or physical attacks* are a serious new threat. Since client computers are increasingly mobile, and mobile computing devices are easily lost or stolen, attackers can get physical access to the device and launch physical attacks. Most security solutions in use today consider only software attacks in their threat model. For example, the computer industry's state-of-the-art Trusted Platform Module (TPM) chip [34] added to protect Personal Computers (PCs) assumes only software attacks in its threat model, and hence is vulnerable to physical attacks such as probing physical memory [17].

Many approaches have been suggested to increase protection of sensitive software execution in a commodity software stack, but they often exhibit shortcomings in security, scalability or functionality. Most software techniques are vulnerable to compromised OS (privileged) code and to hardware attacks. Some approaches (including AEGIS [33]) build security into the OS, which is highly desirable. However, it is not something that can be done by application writers, or hardware microprocessor vendors, who have to live with a commodity OS.

Some software solutions use hypervisors or virtual machine monitors (VMMs), e.g., VMware [32] or Xen [6], to create isolated Virtual Machines. An untrusted application is run on an untrusted commodity OS in one Virtual Machine, while a trusted application is executed on a new trusted OS in another Virtual Machine. However, VMM solutions provide coarse-grained Virtual Machine compartments while we provide fine-grained compartments (within a Virtual Machine). Our compartments are created within the VM hosting the commodity OS to avoid the high cost of a *world switch* between VMs [27, 3] when invoking

[1] Nominated for Best Paper Award. The 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India - January 9-14, 2010.

trusted software. Furthermore, the VMM itself is vulnerable to hardware attacks. We leverage virtualization techniques to override the OS where necessary for security. Unlike other architectures using hypervisors, we also propose new hardware features specifically to protect our hypervisor from hardware or physical attacks as well as software attacks.

Existing hardware techniques have limited scalability due to finite hardware resources, and constrained functionality due to the lack of visibility into the software context. In addition, they (e.g., TPM [34]) are usually still susceptible to hardware attacks. While a few proposed hardware solutions, e.g., XOM [23] and the Secret Protection (SP) architecture [22, 12] do protect against hardware attacks, they both are still susceptible to memory replay attacks. SP also allows only one trusted software module at a time, or multiple concurrent trusted software modules which belong to the same trust domain, i.e., they can all access the same set of protected information. In this paper, we allow scalability to an arbitrary number of trusted software modules, in different trust domains. Modules can be in the application space or in the OS space (e.g., loadable kernel modules). Each module has its own secure persistent storage, that is not accessible by other modules, nor even by the OS. Past work is further compared to our solution in Section 6.

Our main contributions are:

- Low-overhead architectural support for an arbitrary number of trusted software modules within an unmodified commodity software stack.
- New hardware protection for a hypervisor that is resistant to physical attacks as well as software attacks.
- New architectural mechanisms for secure module launch, protected virtual-to-physical memory mapping, secure module storage and inter-module control flow.
- Proof-of-concept FPGA implementation of our security architecture, with a modified microprocessor and a modified hypervisor running unmodified versions of Sun OpenSolaris and Ubuntu Linux.

The rest of this paper is organized as follows. Section 2 discusses our threat model. Section 3 gives an overview of the Bastion architecture while Section 4 describes its concepts and mechanisms in detail. Section 5 presents our implementation using the OpenSPARC platform. Section 6 compares Bastion to past work and Section 7 concludes.

2. Threat Model

We consider hardware or physical attacks in addition to software attacks. Unlike TPM which considers the whole computer box secure from physical attacks, we reduce our hardware security perimeter to just the microprocessor chip, as in [23, 33, 22, 12]. All other hardware like memory, buses and disks are considered vulnerable to physical attacks. Because of highly layered manufacturing technology, successful probing of complex microprocessors without destroying their functionality is extremely difficult, and hence signals within the microprocessor chip are considered secure from physical attacks. We assume the microprocessor does not contain design or implementation flaws, hardware Trojans or viruses.

Software attacks can be carried out by malicious OS or application code snooping on or corrupting security-critical software state in disks, memory, caches or registers. An adversary with physical access to the platform can also intercept its signals and tamper with its hardware (except for the microprocessor). For example, an attacker could use hardware probes to snoop on, or corrupt, values on the memory bus.

We consider both passive and active attacks on confidentiality and integrity. Adversaries can easily carry out a passive attack on data confidentiality, i.e., observe bus, memory or disk data. Similarly, attackers can affect data integrity using active attacks such as *spoofing* (illegitimate modification of data), *splicing* (illegitimate relocation of data), and *replay* (substituting stale data) attacks. We consider operational attacks, not developmental attacks. We do not consider covert or side channel attacks in this paper, nor denial of service attacks.

3. Bastion Overview

The goal of Bastion is to protect the execution and storage of trusted software modules within untrusted commodity software stacks. First, the Bastion microprocessor protects the storage and runtime memory state of our enhanced hypervisor against both software and hardware attacks. Then, the hypervisor uses its hardware-protected environment to protect an arbitrary number of trusted software modules. Each of these modules is provided with a secure execution environment and its own secure storage area, both protected against software and hardware attacks. The hypervisor also protects the invocation of trusted modules and their preemption on interrupts to ensure all control flow in and out of trusted modules is secure. Figure 1 depicts an example application of Bastion, where three OS and application modules (A, B, C) run on or within commodity operating systems. Each module is associated with its own secure storage area on the untrusted disk.

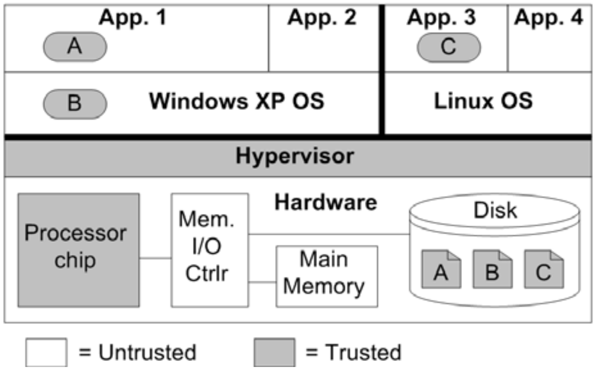

Figure 1. Application of Bastion for three trusted software modules A, B and C

We assume that the *baseline CPU* has virtualization support (as in [18, 2]). Conceptually, this means it has at least three hierarchical privilege levels, or protection rings. In addition to the usual user (PL=3) and supervisor (PL=0) privilege levels, the hardware also provides new hypervisor or Virtual Machine Monitor (VMM) privilege level(s). The hardware ensures that software at other privilege levels cannot access code or data at the hypervisor privilege level (sometimes called ring -1). The hypervisor gives its Virtual Machines (VMs) the illusion they each have unrestricted access to all hardware resources, while retaining ultimate control on how the resources are used.

The platform supports virtualized software stacks, where *guest operating systems* run in separate VMs, monitored by a hypervisor. We use the term *machine memory* space to denote the actual physical memory available in the hardware. The hypervisor virtualizes the machine memory space to create *guest physical memory* spaces for the guest operating systems it hosts.

When an OS builds a page table for an application, it maps virtual pages to page frames in its guest physical memory space. To map guest virtual memory to machine memory, hypervisors use either *shadow page tables* [1] or *nested paging* [3], the two main techniques for memory virtualization. Mainstream virtualization-enabled microprocessors support both techniques [18, 2]. In the former, the hypervisor maintains for each application a shadow page table translating virtual addresses to machine addresses. In the latter, the hypervisor only keeps track of guest-physical-to-machine memory translations in nested page tables. Virtual-to-machine address translations are provided to the processor on Translation Lookaside Buffer (TLB) misses, either by a hardware page table walker or by the hypervisor itself. On a miss, shadow page tables already contain up-to-date versions of these translations. Nested page tables must be looked up in parallel with guest page tables to construct the missing translation on-the-fly. In both approaches, the hypervisor, not the guest OS, retains ultimate control on translations inserted in the TLB.

We also assume there are two levels of on-chip cache (L1 and L2), as is the case in many general-purpose processors. More than two levels of on-chip caches can be supported—we just use L1 to refer to that closest to the processor and L2 to refer to the last level of on-chip cache. For simplicity, this paper only considers the case of a uniprocessor. Applying Bastion to multi-threaded, multi-core and multi-chip processor systems is future work.

3.1. Protecting the Hypervisor

New Bastion hardware features in the processor directly protect our security-enhanced hypervisor. We describe the secure launch of the hypervisor (without needing a TPM chip), and its secure storage. Secure runtime memory is provided to the hypervisor and trusted software modules via a common mechanism described in Section 4.3.

3.1.1. Secure Launch. The boot sequence of a Bastion platform is similar to that of a regular computing platform. The reset vector of the Bastion processor points to untrusted BIOS code, like for a traditional processor. This code sets up a basic execution environment and then relinquishes CPU control to an untrusted hypervisor loader. The loader fetches the hypervisor binary image from persistent storage (e.g., disk), loads it into memory and jumps to the hypervisor's initialization routines. The Bastion secure hypervisor must then invoke a new `secure_launch` instruction for the Bastion processor to begin runtime protection of hypervisor memory and activate hypervisor secure storage capability.

The `secure_launch` instruction transfers control to an internal routine that executes out of an on-chip memory mapped to a reserved segment of machine address space. This software routine computes a cryptographic hash over the state of the hypervisor and stores the resulting hash value in a new Bastion register (*hypervisor_hash*). This value is the *identity* of the loaded hypervisor, to be used in binding a hypervisor to its secure storage area.

Security Analysis. If the untrusted loader skips `secure_launch` or loads a corrupted hypervisor, the *hypervisor_hash* value will be different, or will not get computed at all. The hypervisor's secure storage area then remains locked, since it is tied to the good *hypervisor_hash* value. This ensures that no other software can read or modify the information in the hypervisor's secure storage. Secure storage is further discussed in Section 4.5.

The `secure_launch` routine also generates a new key for each power-on event, used to protect hypervisor memory during runtime from hardware and software attacks. Any hypervisor data is automatically encrypted by a dedicated on-chip crypto engine whenever it gets evicted from on-chip caches back to main memory. Hypervisor cache lines are also hashed before being evicted to memory, as part of a memory integrity tree mechanism (described in Section 4.3).

3.2. Protecting Trusted Software Modules

Once the hypervisor is up and running within its protected memory space, it can spawn any number of Virtual Machines (VMs). For each VM, it allocates a chunk of memory and other resources. It copies or maps boot code (e.g. OpenBoot or BIOS code) into the partition's memory space and then jumps to that boot code. The boot code initializes the VM and eventually loads an operating system, which in turn loads any number of applications. At any point during runtime, a security-critical OS or application module may invoke a new SECURE_LAUNCH hypercall to request a secure execution compartment from the hypervisor. This invocation is slightly different from the secure hypervisor launch done with the `secure_launch` instruction. A module's SECURE_LAUNCH hypercall has an argument pointing to a new *security segment* data structure (Figure 2). This defines the module's private code and data pages, authorized entry points and authorized shared memory interfaces. The hypervisor's SECURE_LAUNCH routine parses this definition to set up our new *Shadow Access Control* mechanism (Section 4.2), which creates a memory compartment for the module's memory pages and enforces strict separation of this compartment from the rest of the software stack during runtime. The SECURE_LAUNCH hypercall also activates runtime memory protection for all module pages and computes the *module identity*, a hash of the module's initial state and its security segment.

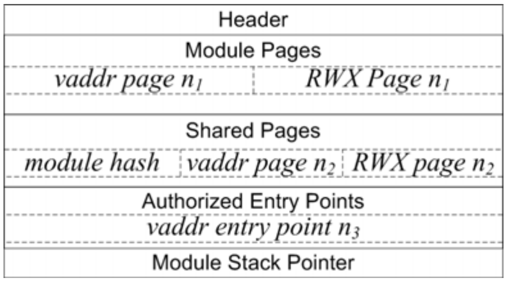

Figure 2. Security Segment

4. Bastion Architecture

We describe key components in the Bastion architecture below. New registers, state, routines and mechanisms introduced are summarized in Figure 3.

4.1. Security Segment, Module State Table and Module Identity

Each trusted software module has a security segment (Figure 2). This must be made available to the software module before it invokes SECURE_LAUNCH. It can be compiled into the data space of the application or OS containing the module, or it can be read from a separate data file. Its internal structure is fixed and known to the Bastion hypervisor. The module definition contained in the security segment is derived from instructions given by the programmer, e.g., via code annotations. The main part of this definition is a set of *Shadow Access Control rules*, each formed by the following triple: 1) a *module hash* identifying a module, computed over the module's initial code and data, 2) the virtual address of a page associated with

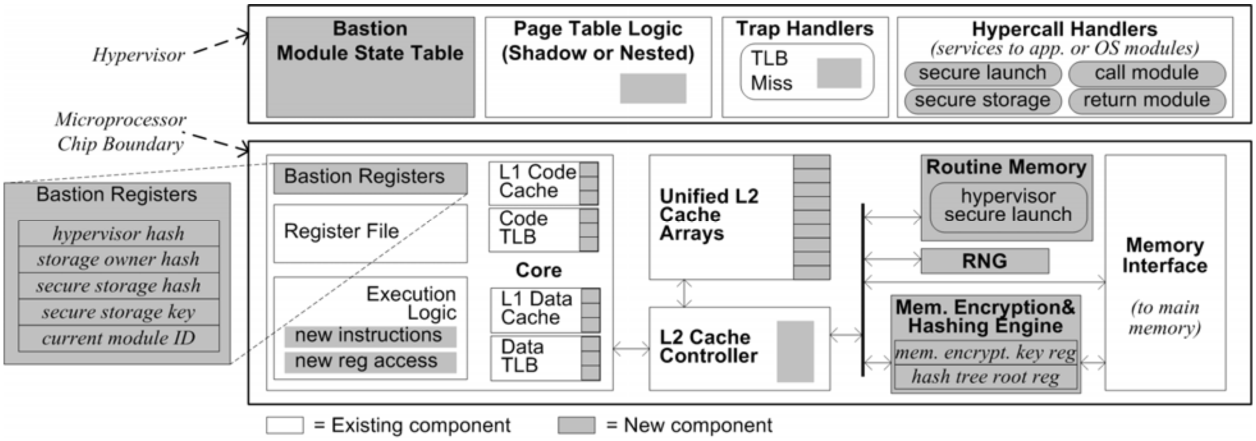

Figure 3. New Bastion Features in Microprocessor and Hypervisor

the identified module, and 3) the Read/Write/eXecute (RWX) access rights given to this module for that page. The *Module Pages* section describes the pages belonging specifically to the module being defined in this security segment. The Shadow Access Control rules it lists thus have an implicit module hash. Shared memory interfaces are defined in the *Shared Pages* section of the segment, where Shadow Access Control rules identify sharing modules explicitly. An *Authorized Entry Points* section lists the virtual addresses of authorized entry points into module code. Finally, *Module Stack Pointer* specifies the top of the module's private stack. Memory for this stack is reserved within the pages listed in *Module Pages*.

SECURE_LAUNCH parses a security segment to extract Shadow Access Control rules and then checks their compatibility with rules requested by modules processed in previous invocations of SECURE_LAUNCH. The hypervisor checks that: 1) there is no aliasing between modules' private virtual pages, nor between the corresponding machine pages, and 2) all modules sharing a memory page agree on one another's identity. Validated rules are added to a hypervisor data structure called the *Module State Table*. To speed up verification of aliasing between machine pages, the Module State Table also contains a table mapping each machine page to the set of modules allowed access to the page. These mappings are also updated as validated rules are committed to the Module State Table. Within these data structures, the hypervisor identifies modules using their `module_id`, a shorthand for their full identity hash. The `module_id` is a unique identifier assigned to each module by SECURE_LAUNCH and valid until the next platform reset. It is used in hypervisor software and in hardware tags, so its bit width must be much smaller than that of a full hash (which is typically 128 or 160 bits), but also large enough to accommodate a large number of concurrent modules; between 8 and 20 bits should be sufficient.

Bastion supports arbitrary module sizes, defined at page granularity. For example, a module could be composed of the few pages containing the code and private data of a security-critical function, or the code and data space of a library, or an OS loadable module, or an entire trusted application or even the whole memory space of a trusted OS. Ideally, a module should encapsulate only the code and data necessary to fulfill its security objectives, making it small and easy to verify for correctness, possibly using formal methods. However, it may not always be possible to perform such fine-grain partitioning of an application or OS, especially in legacy software. Hence Bastion also supports large modules to provide monolithic pieces of software with physical attack protection and secure storage. Methodologies for partitioning software systems into trusted and untrusted components are studied elsewhere, e.g. [10].

4.2. Enforcing Virtual Memory Compartments

At runtime, the compartments defined by Shadow Access Control rules are enforced by the processor hardware. Rules are available in the instruction and data Translation Lookaside Buffers (TLBs) and enforcement is done during the TLB lookup preceding every memory access. Infringement is signaled by a new type of TLB miss causing the hypervisor to intervene. To express Shadow Access Control rules, TLB entries are extended with a `module_id` tag identifying a module allowed access to the page. To enable enforcement, a new *current_module_id* register specifies the `module_id` of the module currently executing. Memory accesses are allowed to go forward only if the TLB `module_id` of the page accessed is equal to the one in *current_module_id*. All software that is not in a Bastion compartment is considered part of a generic *module zero*, with pages tagged using the reserved `module_id` 0. Therefore, modules (including the untrusted module zero) can only access instructions and data tagged with their `module_id`.

The method for getting the right `module_id` to a TLB entry on a TLB miss is hypervisor-specific: it depends on the technique used for virtualizing machine memory. When shadow page tables are used, the hypervisor extends shadow page table entries with the `module_id`. The transfer of a shadow page table entry from hypervisor memory to the TLB thus automatically transfers the `module_id`. To handle shared virtual pages, the hypervisor must replicate shadow page table entries for these pages and assign

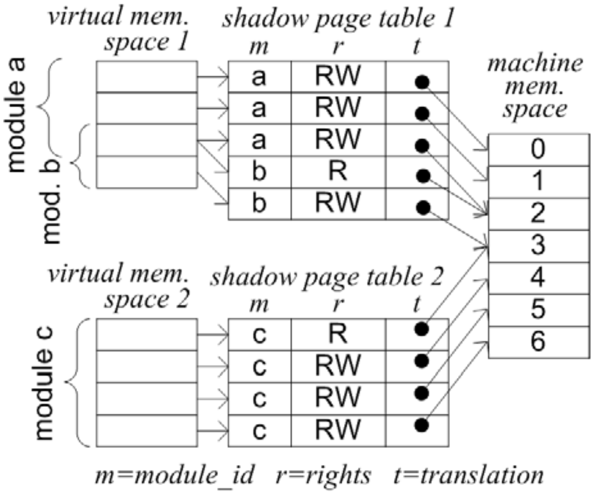

Figure 4. Shadow Access Control with Shadow Page Tables

each entry the module_id of a module sharing the page. The hypervisor checks that replicated entries map to the same machine page to ensure the correctness of this form of aliasing. Hardware page-table-walkers, if used, must be modified to handle shared virtual pages. Figure 4 depicts module a and b sharing machine page 2 within virtual address space 1 and module b and c share machine page 3 across virtual address spaces 1 and 2. It also shows each module's entry may have different access rights, e.g., a producer-consumer buffer between the two modules.

Hypervisors using nested paging rather than shadow page tables only track guest-physical-to-machine memory mappings; they do not maintain copies of all virtual-to-machine memory mappings as is done in shadow page tables. In this case, missing TLB entries—the virtual-to-machine address translations with access rights—must be constructed on-the-fly by the hypervisor or a hardware nested page table walker instead of simply fetched from a shadow page table. To add the module_id to these TLB entries, we add an extra step to the TLB miss handling procedure: once the entry is constructed, the hypervisor checks that the entry respects the rules in the Module State Table, adds the appropriate module_id and writes the extended entry to the TLB. When a hardware nested page table walker is used, a new hypervisor trap is triggered at the end of a table walk. Hypervisor software then inspects and extends the new TLB entry.

When the hypervisor determines a memory access request violates Shadow Access Control rules, it either restricts or denies the request, depending on the requestor. A guest OS is given access to an encrypted and tamper-evident version of the cache line, while an application is simply denied access. Restricted access is given to guest operating systems to ensure they can perform paging or relocate pages in physical memory. Encryption and tamper-evidence is provided by the Secure Physical Memory mechanisms described next.

4.3. Secure Physical Memory

Bastion protects hypervisor and module memory state against hardware adversaries that might snoop on or corrupt data on buses and memory chips. Such memory confidentiality and integrity is provided by two new hardware cryptographic engines located between the L2 cache and main memory. One engine encrypts and decrypts protected cache lines while the other verifies their integrity using a hash tree. Using the on-chip RNG, the memory encryption key is generated anew by the on-chip secure_launch routine upon every platform reset to thwart (known-ciphertext) attacks that could leak information across reboots if the same key was reused.

The integrity hash tree detects memory corruption attacks by checking that what is read from main memory at a given address is what the processor last wrote at that address. This is based on the Merkle tree [28], adopted in [33]. The Merkle tree technique recursively computes cryptographic hashes on the protected memory blocks until a single hash of the entire memory space, the tree root, is obtained. The tree root is kept on-chip in the microprocessor, while the rest of the tree can be stored off-chip. Authenticating a block read from main memory requires fetching and verifying the integrity of all hashes on the tree branch starting at the block of interest and ending with the root. Similarly, a legitimate update to a block triggers updates to hashes on the block's branch, including the root. Tree nodes can be cached [16] to speed up tree operations and the simple hashing primitive can be substituted for one with better performance characteristics [30].

During platform boot-up, the on-chip secure_launch routine initializes the tree to fingerprint the loaded hypervisor. Two new L2 cache tag bits (i_bit and c_bit) identify cache lines requiring encryption or hashing. The values of these bits come from the TLB, where each entry is also extended with an i_bit and a c_bit. Bastion runtime memory protection mechanisms operate at a page granularity: all L2 cache lines in a page are tagged with the i_bit and c_bit read from that page's TLB entry. The bits in this entry are set by the hypervisor on a TLB miss.

To protect trusted module memory, the SECURE_LAUNCH hypercall encrypts the pages of each module being launched, and adds its pages to the tree's coverage. During runtime, the hypervisor sets the TLB i_bit and c_bit for module pages. Our tree also protects module pages swapped from memory to an on-disk page file and back, similarly to [30].

Bastion's hash tree is slightly different from traditional Merkle hash trees. We selectively cover only hypervisor and trusted module pages, while they cover the entire physical memory space. We cover pages swapped out to disk (as in [30]), while they do not cover data outside the physical memory space.

To efficiently support its functionality, our integrity tree is formed of two parts: a *top cone* and several *page trees*. A page tree is a Merkle hash tree computed over a page of data or code, with a root called the *page root*. The top cone is a Merkle hash tree rooted in a register of the hashing engine (*hash_tree_root_reg* in

Figure 3) and computed over the page roots, to protect their integrity. Our tree is shown in Figure 5. Upon platform reset, there are no pages to protect so the top cone is initialized on a set of null page roots. When a page is added to the tree, its page tree is computed and its page root replaces a null page root.

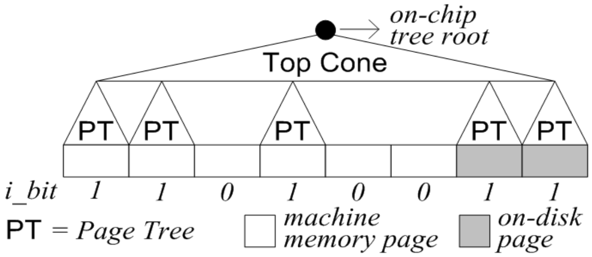

Figure 5. The Bastion Integrity Tree

For simplicity, we assume the components of the tree are stored in dedicated machine memory regions and that the initial top cone covers enough null page roots to satisfy the memory needs of the software stack. Dynamic expansion and shrinkage of integrity tree coverage is an orthogonal research problem; it can be addressed using techniques such as in [8].

Through its monitoring of (shadow or nested) page table changes, the hypervisor can detect a guest OS is sending a page out to disk. If the remapping respects Shadow Access Control rules, the hypervisor moves the page's tree so it becomes rooted by one of the page roots reserved for disk pages. As a result, the tree root fingerprints module pages in main memory and on the disk. When the guest OS moves a page from the disk back to physical memory (or simply relocates a page in physical memory), the hypervisor moves the page tree to reflect the page's new position. If the OS or a physical attacker corrupts the page while it resides on the disk, the integrity checking engine detects tampering as soon as the affected cache lines are accessed by the module. Therefore, critical module pages remain tamper-evident and confidential as they are moved by the guest OS between memory and disk.

4.4. Secure Inter-Module Control Flow

Entering and leaving protected modules securely is essential for security-critical software. Bastion addresses this need for secure control flow with mechanisms enabling secure invocation and preemption of modules.

4.4.1. Module Invocation. VM software always starts executing in the unprotected module zero, where the bootstrap procedure takes place. The hypervisor thus starts a VM with the *current_module_id* register set to zero. It remains set to zero until a protected software module is called via the new CALL_MODULE hypercall. This hypercall takes as parameters the module hash of the callee and the virtual address of the desired entry point. The hypervisor services CALL_MODULE by first checking the validity of this entry point against the list of authorized entry points provided upon SECURE_LAUNCH for the callee. When the entry point is valid, the hypervisor registers the desired transition in the Module State Table and returns from the hypercall, but it does not change the value in the *current_module_id* register yet.

Fetching the first callee instruction triggers a TLB miss due to a module_id mismatch—the instruction is tagged with the callee's module_id while the *current_module_id* register still contains the caller's module_id. This TLB miss is handled by the hypervisor. It checks whether the transition that just occurred was legitimate and previously registered in the Module State Table. If so, it allows the transition by setting *current_module_id* to the callee's module_id. It also saves the address of the call site so that it can later enforce a correct return into caller's code. Execution of the callee module then resumes.

Callee modules can only return to their callers via the RETURN_MODULE hypercall. As for CALL_MODULE, this hypercall registers the requested transition, but it does not modify the value in the *current_module_id* register. This means that when the callee executes the return instruction to jump back to the caller's code, an instruction TLB miss occurs due to a module_id mismatch. The hypervisor intervenes to first check that this transition was previously requested. It also checks that the return address into the caller's code corresponds to the instruction following the caller's call site, previously saved by the hypervisor. When all checks pass, the hypervisor allows the return transition back into the caller by setting the *current_module_id* register to the caller's module_id.

4.4.2. Module Preemption. When a protected module gets preempted by the guest OS (e.g. to service a timer or device interrupt), the hypervisor first intervenes to save the module's register state in the Module State Table. It also wipes out any remaining register state so that malicious OS code is unable to observe or modify critical module state via its registers. When the OS resumes module execution, the hypervisor intervenes again to restore the register state of the preempted module, including the program counter. Hypervisor intervention upon module preemption or resumption is triggered by a module_id mismatch between module code and OS interrupt handler code.

4.5. Scalable Secure Persistent Storage

4.5.1. Hypervisor Secure Storage. The hypervisor secure storage is a non-volatile area, located in a flash memory chip or disk, where the hypervisor stores long-lived secrets and other persistent sensitive data. Its protection is rooted in new Bastion hardware registers. The hypervisor creating such an area protects its confidentiality and integrity by encrypting it with a symmetric encryption key and fingerprinting it with a cryptographic hash, both stored in dedicated microprocessor registers (*secure_storage_key* and *secure_storage_hash*). The values in these non-volatile registers exist across reboots and are only accessible to the hypervisor that created the storage area, identified by the hash value contained in the new non-volatile *storage_owner_hash* register. When the values in *storage_owner_hash* and *hypervisor_hash* match, the `secure_launch` routine "unlocks" the existing secure storage area by giving the hypervisor access to the *secure_storage_hash* and *secure_storage_key* registers. The contents of these registers allow the loaded hypervisor to decrypt and hash verify the secure storage area it fetches from disk or flash, which may have been tampered with by a physical attacker since the last power down. When the values in *storage_owner_hash* and *hypervisor_hash* do not match, the `secure_launch` routine wipes out the contents of the *secure_storage_hash* and *secure_storage_key* registers before they become accessible to the loaded hypervisor. This prevents different hypervisors from accessing one another's secure storage areas and locks corrupted hypervisors out of a good hypervisor's secure storage.

Mechanisms for backup and migration of hypervisor secure storage should also be devised to avoid loosing stored data upon deployment of a new hypervisor or following the load of a corrupted hypervisor. Similarly to TPM [34], we suggest using trusted authorities from which the hypervisor can request migration and backup services. The hypervisor can establish a secure communication channel to such an authority by authenticating the authority using a public key embedded in its data space.

4.5.2. Module Secure Storage. The hypervisor leverages its hardware-rooted secure storage to provide trusted software modules with their own secure storage in a scalable way. The secure storage area of a module is kept confidential with a symmetric encryption key and made tamper-evident with a cryptographic hash computed over its contents. The hypervisor labels this (key, hash) pair with the identity of the module and stores the labeled pair in the hypervisor's secure storage area, protected by the dedicated hardware registers. This method, depicted in Figure 6, can scale up to an arbitrary number of module storage areas. Hence, only two hardware registers (to protect the hypervisor's storage) are needed to protect any amount of data (for any number of trusted modules).

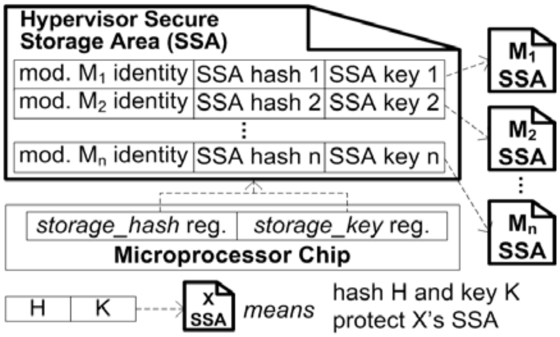

Figure 6. Scalable Secure Storage

Modules access and manage their secure storage areas via new hypercalls. To create a new area, a protected module 1) generates a key, 2) encrypts the data to be stored in the area with the key, 3) computes a hash over the contents of the encrypted data and 4) stores the encrypted data to disk or flash memory. Because it is encrypted and hashed, this data can safely exit the module's compartment to be written to disk via an untrusted file system manager using an untrusted disk device driver. To enable subsequent access to the data across platform reboots, the module invokes the WRITE_STORAGE_KEY and WRITE_STORAGE_HASH hypercalls with, respectively, the encryption key and the computed hash as arguments. These hypercalls bind the module's identity to the key and hash pair protecting the new storage area, and store the resulting (key, hash, identity) tuple in the hypervisor's secure storage area. To recover this key and hash following a platform reboot, the module invokes the READ_STORAGE_KEY and READ_STORAGE_HASH hypercalls. The hypervisor services these calls by returning the key and hash only if the identity of the invoking module (computed during a SECURE_LAUNCH invocation) is identical to the identity of the module that created the secure storage area. When a module modifies the contents of an existing area, it invokes the WRITE_STORAGE_HASH to commit the updated state to the hypervisor's secure storage. Modules can detect the malicious replay of an old version of their storage since the hypervisor always returns the hash reflecting the latest update. Our approach is similar to that in the SP architecture [12], except that we provide secure storage to multiple modules from different trust domains rather than a single one.

5. Implementation

5.1. Baseline Architecture

We implemented Bastion by modifying the Sun Microsystems UltraSPARC T1 (codename *Niagara*) microprocessor [21] and the UltraSPARC hypervisor. The Verilog description of the microprocessor and the source code of the hypervisor are both publicly available as part of the OpenSPARC project [29]. OpenSPARC also provides developers with a full-system implementation, in a Xilinx Embedded Developer's Kit (EDK) project, of the T1 chip—i.e. the UltraSPARC core, crossbar, L2 cache (emulated by a Xilinx MicroBlaze processor running firmware), UART and on-chip Ethernet and memory controllers. This implementation can be synthesized for a Xilinx Virtex 5 Field-Programmable Gate Array (FPGA) and executed on the Xilinx ML505 development board. This board includes a Virtex 5 FPGA chip, a 256MB DRAM DIMM, a Flash memory card controller, a Network Interface Card and other I/O peripherals.

As a proof-of-concept implementation, we modified the single-thread UltraSPARC microprocessor with new Bastion hardware, and modified the hypervisor with new Bastion features. The resulting FPGA board successfully hosted our modified Bastion hypervisor, protecting trusted software modules in an application running on an unmodified commodity OpenSolaris operating system. All the functionality discussed in this paper was implemented either in hardware, firmware or software, except for the Random Number Generator, restricted OS access to protected pages and shadow page table logic (the hypervisor uses nested paging). We also implemented hypervisor and module attestation capabilities, not discussed in this paper for lack of space. While we present results for OpenSolaris, we also compiled and ran our application on an unmodified Linux Ubuntu OS, without changing a single line of code in our Bastion hypervisor.

5.2. Implementation Strategy

5.2.1. Microprocessor Core. To add the new Bastion registers (Figure 3) to the T1 microprocessor, we wrote a top-level Verilog unit containing storage flip-flops and address decoders for 32 new 64-bit registers. These 2048 bits of register storage hold the hypervisor hash, the secure storage key, hash and owner hash, and a private attestation key not discussed in this paper. We could not make any of these registers non-volatile since the Virtex 5 does not provide on-chip non-volatile storage capabilities.

Instruction and data TLBs were extended with a 5-bit `module_id` in each entry and a 5-bit *current_module_id* register was added. The TLB lookup logic was also enhanced to raise a TLB miss on a mismatch between the `module_id` tagging a "matching" TLB entry and the value in the *current_module_id* register. Although we designed Bastion to support much more than 32 modules, 5 bits was a convenient width for this implementation since it corresponds to an unused part of existing TLB lines.

Finally, we implemented internal microprocessor routines (hypervisor secure launch and attestation) as compiled SPARC code executing from a reserved range in machine address space. Due to a shortage in FPGA RAM blocks, this range currently gets mapped to off-chip rather than on-chip memory. The routine code is responsible for identifying the hypervisor so it cannot rely on the hypervisor. We developed it as a self-contained software stack with a tiny, statically-linked C library as runtime support for the secure routines. To invoke an internal microprocessor routine, the hypervisor simply jumps to its entry point within the reserved range of machine address space.

5.2.2. Firmware and Crypto Hardware. Unfortunately, the EDK project does not implement L2 cache arrays and controllers in hardware. MicroBlaze firmware emulates the L2 cache without allocating actual L2 storage. We implemented our cache-related mechanisms in the MicroBlaze firmware, adding a hardware AES encryption-decryption engine as a Fast Simplex Link (FSL) MicroBlaze peripheral. Our extended firmware checks whether the `c_bit` of each cache line accessed by the core is set and if so, sends the line to the hardware AES engine to be decrypted or encrypted on, respectively, a read from or write to external memory. Similarly, the firmware checks the `i_bit` and uses the AES engine as a CBC-MAC (Cipher Block Chaining - Message Authentication Code) hash function to perform integrity hash tree verifications or updates. Finally, our extended firmware detects and routes accesses to the machine memory range reserved for secure on-chip routines.

5.2.3. Hypervisor Software. To add Bastion mechanisms to the UltraSPARC hypervisor, we modified the instruction and data TLB miss handlers and added new hypercall handlers. To speed up implementation, many of the functions added to the hypervisor were coded in C rather than the SPARC assembly used throughout the existing hypervisor (except for initialization routines written in C). This required providing the hypervisor with its own stack and with routines to make enough register windows available to the C routines. Using C code during runtime in a thin hypervisor layer is clearly sub-optimal, but still provides us with a proof-of-concept implementation that can boot a full commodity operating system and its applications in an acceptable amount of time.

5.2.4. Trusted Module Software. The application modules we implemented have their own private data and code space, and they share a single page as a memory interface. To support dynamic memory allocation, a set of pages is assigned to each module to be used as their private heaps and stacks. The hypervisor routines for secure module control flow are responsible for switching stacks upon entering or leaving a module. To support functions such as heap memory allocation, each module is assigned a separate copy of a statically-linked, minimalist C library.

5.3. Complexity Costs

Tables 1 and 2 show, respectively, the hardware and software complexity of the Bastion mechanisms. Table 1 shows that our modifications to the microprocessor core increase resource usage by less than 10%. Table 2 shows increases in the size of the hypervisor and firmware code base are between 6 and 8%. The new AES crypto core causes the biggest increase in resource consumption. We note that these numbers could be significantly reduced if the Bastion logic was implemented and optimized by experienced engineers.

5.4. Performance Impact

In Table 3, we provide preliminary evidence that the performance overheads caused by Bastion protection are reasonable given the high throughput of modern microprocessors. We present instruction counts rather than cycle counts since the absence of an L2 cache, combined with our rudimentary memory crypto engine (single engine for both encryption and hashing, clocked at less than a quarter of the memory frequency, with no caching of tree nodes) currently causes extremely high load latencies. Our next step in developing this prototype is to provide it with memory crypto engines that can match memory bandwidth and cache frequently accessed metadata, as suggested in the literature on the subject [16, 30].

Table 1. Hardware Complexity

	Original System	Bastion Additions *absolute (change)*
SPARC CPU Core	*T1 core*	*new regs, TLB tags, new instructions*
Slice Registers	19.6K	1.6K (+8.2%)
Slice LUTs	30.9K	1.1K (+3.6%)
BRAMs	98	0 (+0%)
Non-core HW	*crossbar, L2 ctrlr,etc*	*AES crypto core, μBlaze interface*
Slice Registers	28.7K	5.9K (+20.6%)
Slice LUTs	39.5K	6.5K (+16.5%)
BRAMs	114	15 (+13.2%)

Table 2. Software Complexity (lines of code)

	Original System	Bastion Additions *absolute (change)*
OpenSolaris OS	9.06M	0 (+0%)
Hypervisor	38.1K	2.9K (+7.6%)
Cache Firmware	12.1K	0.8K (+6.6%)
CPU Routine	0	1.5K (N/A)

The table shows the number of instructions that need to be executed to handle our various hypercalls and to enter or leave modules (depicted as “module call/return” and “module preempt/resume”). The SECURE_LAUNCH creating our compartments requires less than 15K instructions, which compares advantageously to the 468K instructions required to initialize an UltraSPARC virtual machine on the same platform. Our module transition mechanisms can enter and leave Bastion compartments in less than 2,000 instructions. With an operating system giving our modules 10ms quanta of CPU time, these overheads correspond to a performance hit of less than one tenth of a percent. On a 2GHz CPU with enough L2 cache storage to support a throughput of 0.5 instruction per cycle, secure storage hypercalls execute in under 1μs, several orders of magnitude faster than the sealed storage operations on a TPM chip, reported to require hundreds of milliseconds to complete [26].

Table 3. Performance overhead of Bastion operations

	Bastion Performance *# of instructions executed*		Notes / Comparison
module SECURE_LAUNCH	11,881 *(40kB module)*	13,781 *(96kB module)*	Initializing SPARC VM: 468,537 instructions
module call/return	1913	1902	< 0.1% runtime overhead*
module preempt/resume	1241	1203	
READ/WRITE_STORAGE_KEY	719	719	orders of magnitude faster than equivalent TPM operations*
READ/WRITE_STORAGE_HASH	909	941	

* *These are rough approximations assuming a 2GHz processor running at a throughput of 0.5 instructions per cycle, with an OS assigning CPU time quanta of 10ms*

6. Related Work

We distinguish between three approaches for protecting critical software in a commodity stack: 1) add security features to the OS, 2) measure and verify an unmodified OS or 3) bypass the OS altogether.

Security mechanisms built into an operating system benefit from significant visibility into the software they are protecting since OS code has access to memory and I/O mapping information. Such mechanisms can mediate system calls, I/O requests and even memory accesses to provide security-critical tasks with isolated execution or some form of secured I/O, e.g. [24, 35, 33, 36, 7]. However, secure operating systems are either custom built and hence are unlikely to replace well-established commodity operating systems, or they are based on a mainstream OS and are exposed to software vulnerabilities in the large privileged code base they are built upon. Modifications to the microprocessor have been suggested to limit the privileges of vulnerable OS code [7] but only to enable detection of tampering with static code and data, not dynamic data (e.g. stack, heap). In most cases, these are software-only approaches that remain vulnerable to attackers with physical access to the device. Bastion, however, provides protection from physical attacks.

Rather than add security to an OS, other approaches measure and verify unmodified operating systems prior to providing the security-critical tasks they run with sensitive data to process [31, 25, 37]. These techniques compute a cryptographic hash to fingerprint the initial data and code state of the OS; the hash is then used as an identity in an attestation report sent to a remote party. The party is expected to determine whether the identified OS is trustworthy, and if so, reply with sensitive data sealed to the identity. These secure storage and attestation capabilities are provided by a hardware Trusted Platform Module (TPM) [34] chip. Although TPM keys can be used to protect disk data while the device is powered off, TPM-based systems remain vulnerable to physical attacks during runtime [19, 17]. Even without physical attacks, these approaches are likely to remain vulnerable to software-based runtime attacks since remote parties are typically unable to correctly assess the trustworthiness of a large commodity software stack [7].

Finally, some architectures bypass the commodity operating system altogether to protect a security-critical application module using either an enhanced microprocessor or a trusted hypervisor. These tend to focus on memory compartmentalization, so most do not offer secure storage capabilities as Bastion does. When they do, these services are either restricted to a single trust domain as in SP [12], or they require the use of a slow TPM chip [15, 14, 26] or an expensive PCI-based coprocessor peripheral [13]. There are two broad categories of techniques for bypassing a commodity OS: 1) move security-critical tasks to a trusted OS running concurrently to the commodity OS, within a separate execution environment [15, 5, 20, 4, 26] or 2) isolate security-critical tasks running on the commodity OS [23, 22, 12, 9, 11].

By provisioning additional execution environments, typically extra virtual machines, the techniques in the first category can enforce strict isolation between security-critical tasks and the commodity software stack. For such standalone environments, however, the cost of initializing, context switching and interfacing with the commodity software stack can be high [27] and prevent scalability. In the second category, software techniques [9, 11] secure critical tasks within the commodity software stack without creating new environments, but cannot protect against attackers with physical presence. While providing better security against physical attackers, the hardware-based techniques [23, 22, 12] remain vulnerable to memory replay attacks on dynamic data and are restricted in their scalability by limited hardware resources. They focus on achieving security objectives such as secure storage or confidential and tamper-evident execution despite a potentially compromised OS.

The Bastion architecture adopts the strategy of bypassing the commodity OS to provide to security-critical tasks isolated execution compartments with secure storage. It differs from past work in that it can maintain an arbitrary number of compartments, either in the operating system or application layer, and can defend against hardware attacks, including memory or persistent storage replay attacks. It is also the first security solution to provide strong protection of the virtualization layer's execution and storage.

7. Conclusion

This paper introduced the Bastion architecture, formed of processor hardware and a thin hypervisor, both enhanced to provide scalable secure execution and storage to critical modules within an untrusted commodity software stack. Mechanisms in the processor hardware securely launch the hypervisor and provide runtime cryptographic protection of the hypervisor memory state against physical attacks. Extended memory management mechanisms in hardware and software allow the Bastion hypervisor to define and protect an arbitrary number of low-overhead execution compartments for trusted software modules. Each module is provided with its own secure storage area, rooted in hypervisor secure storage. Our implementation demonstrates the feasibility of skipping an unmodified commodity OS to provide application-level security and shows an acceptable complexity overhead. As opposed to past approaches where the operating system must be trusted, Bastion only needs to trust and protect a thin layer of hypervisor software which can be two orders of magnitude smaller than a commodity OS. Protection mechanisms against hardware attacks provide more security than TPM-based platforms, which cannot defend against probing attacks on buses and memories. Integration of our security mechanisms in the processor hardware also ensures Bastion is much faster than platforms using a slow TPM chip.

8. References

[1] K. Adams et al. A comparison of software and hardware techniques for x86 virtualization, in Proc. of ASPLOS06, Oct. 2006.

[2] AMD, Industry Leading Virtualization Platform Efficiency. www.amd.com/virtualization, 2008.

[3] AMD, AMD-V Nested Paging. AMD Whitepaper Revision 1.0, July 2008.

[4] M.J. Anderson et al. Towards Trustworthy Virtualisation Environments. Technical Report HPL-2007-69, Hewlett-Packard Development Company, L.P., April 2007.

[5] T. Alves et al. Trustzone: Integrated hardware and software security. ARM white paper, July 2004.

[6] P. Barham et al., Xen and the Art of Virtualization, In Proc. of Symposium on OS Principles (SOSP), Oct 2003.

[7] S. Bratus et al. TOCTOU, Traps, and Trusted Computing. In Proc. of the TRUST 2008 Conference, March 2008.

[8] D. Champagne et al., The Reduced Address Space (RAS) for Application Memory Authentication, In Proc. of the 11th ISC'08, Sept. 2008.

[9] X. Chen et al. Overshadow: A Virtualization-Based Approach to Retrofitting Protection in Commodity Operating Systems, In Proc. of ASPLOS08, March 2008.

[10] Y. Chen and R.B. Lee, Hardware-Assisted Application-Level Access Control, In Proc. of Information Security Conference, Sep. 2009.

[11] P. Dewan et al. A Hypervisor-Based System for Protecting Software Runtime Memory and Persistent Storage, SSSS'08, April 2008.

[12] J. Dwoskin and R.B. Lee. Hardware-rooted Trust for Secure Key Management and Transient Trust, Proc. of ACM CCS'07, Oct. 2007.

[13] J.G. Dyer et al. Building the IBM 4758 Secure Coprocessor, Computer, v.34 n.10, p.57-66, October 2001.

[14] P. England et al. A Trusted Open Platform, Computer, v.36 n.7, p.55-62, July 2003

[15] T. Garfinkel et al., Terra: A virtual machine-based platform for trusted computing, in Proc. of SOSP, Oct. 2003.

[16] B. Gassend et al., Caches and Merkle Trees for Efficient Memory Authentication, Proc. of HPCA 2003, Feb. 2003.

[17] J. A. Halderman et al. Lest We Remember: Cold Boot Attacks on Encryption Keys. In Proc. of USENIX Security, July/August 2008.

[18] Intel, Intel Virtualization Technology: Hardware Support for Efficient Virtualization, Intel Technology Journal, Aug. 2006.

[19] B. Kauer, OSLO: Improving the Security of Trusted Computing, in Proc. of USENIX Security, Aug. 2007.

[20] P. Kwan et al. Vault: Practical Uses of Virtual Machines for Protection of Sensitive User Data. In Proc. of ISPEC 2007, May 2007.

[21] J. Laudon. UltraSPARC T1: Architecture and Physical Design of a 32-threaded General Purpose CPU, In Proc. of IEEE ISSCC, Feb. 2006.

[22] R. B. Lee et al. “Architecture for Protecting Critical Secrets in Microprocessors,” Proc. of ISCA 2005, June 2005.

[23] D. Lie et al. Architectural Support for Copy and Tamper Resistant Software, Proc. of ASPLOS IX, 2000.

[24] P. Loscocco et al. Integrating Flexible Support for Security Policies into the Linux Operating System, Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, June 2001.

[25] J. Marchesini et al., Open-Source Applications of TCPA Hardware, Proc. of ACSAC'04, December 2004

[26] J.M. McCune et al., Flicker: An Execution Infrastructure for TCB Minimization, In Proc. of EuroSys2008, March 2008.

[27] A. Menon et al., Diagnosing performance overheads in the Xen virtual machine environment. Proc. of the 1st ACM/USENIX international conference on Virtual execution environments, June 11-12, 2005.

[28] R.C. Merkle, “Protocols for Public Key Cryptosystems,” IEEE Symposium on Security and Privacy, 1980.

[29] Sun Microsystems, http://www.opensparc.net, 2008.

[30] B. Rogers et al., Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly, In Proc. of Int'l Symp. on Microarchitecture (MICRO2007), Dec. 2007.

[31] R. Sailer et al., Design and implementation of a TCG-based integrity measurement architecture, In Proc. of USENIX Security, 2004.

[32] J. Sugerman et al. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor In Proc. of 2002 USENIX Annual Technical Conference, June 2001.

[33] G. E. Suh. AEGIS: A Single-Chip Secure Processor. PhD thesis, Massachusetts Institute of Technology, 2005.

[34] Trusted Computing Group, “Trusted Platform Module (TPM) Main – Part 1 Design Principles,” Spec. v1.2, Revision 94, March 2006.

[35] C. Wright et al. Linux Security Modules: General Security Support for the Linux Kernel, Proc. of USENIX Security, August 2002

[36] N. Zeldovich. Making information flow explicit in HiStar, Proceedings of Operating systems design and implementation, Nov. 2006

[37] J. Cihula, Trusted Boot: Trusted Boot: Verifying the Xen Launch, Xen Summit 07 Fall.