SCALABLE SECURITY ARCHITECTURE FOR TRUSTED SOFTWARE

DAVID CHAMPAGNE

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING
ADVISOR: RUBY B. LEE

JUNE 2010

À ma mère Lise et mon père Robert
Merci d’avoir toujours cru en moi

Abstract

Security-critical tasks executing on general-purpose computers require protection against software and hardware attacks to achieve their security objectives. Security services providing this protection can be offered by mechanisms rooted in processor hardware, since its storage and computing elements are typically outside the reach of attackers.

This thesis presents the Bastion architecture, a hardware-software security architecture for providing protection scalable to a large number of security-critical tasks. Protection is enabled by three sets of new mechanisms: for protecting a trusted hypervisor, for fine-grained protection of modules in application or operating system space, and for securing the input and output of Bastion-protected software modules. This thesis also presents an implementation and evaluation of Bastion, and explores alternatives for one of its core security functions: memory authentication.

The hypervisor, a layer of software dedicated to the virtualization of machine resources, is increasingly being involved in security solutions. We use it in Bastion as a manager of security-critical tasks. While past solutions protect the hypervisor from runtime software attacks, Bastion also protects the hypervisor from physical attacks, protects it from offline attacks, and provides it with a secure launch mechanism. Within this protected Bastion hypervisor, we design a second set of mechanisms that provide separate execution compartments for each security-critical task running in the virtual machines hosted by the hypervisor. These compartments are protected against both hardware attacks and software attacks originating from a potentially compromised operating system. To enable security-critical tasks to communicate with the outside world, we provide a third set of mechanisms for secure input and output to and from Bastion-protected compartments. We implement and evaluate a Bastion prototype by modifying the source code of the OpenSPARC processor and hypervisor systems. Addionally, we survey the design space of alternatives to the Bastion memory authentication mechanism, which is central to protecting critical software execution in Bastion. These contributions can improve security in the digital world by informing the design of the next generation of general-purpose computing platforms.

Acknowledgements

I would like to express my gratitude to my colleagues, friends and family. First, I wish to thank Professor Ruby Lee, my research advisor. Her generous mentoring was crucial to completing this dissertation and the work it describes. Through the years, she has shown me what it means to be a good architect and an effective communicator. She was always available to provide guidance in my work, helping me define, improve and present technical concepts and research ideas.

I wish to thank Professor Andrew Appel from Princeton University and Reiner Sailer from IBM Research for reviewing my dissertation and providing me with helpful and insightful comments and suggestions. I thank Shih Lien Lu and Keen Chan for offering me the opportunity to spend two instructive summers as an intern at Intel in Oregon.

It has been a great pleasure to work with my colleagues at the Princeton Architecture Laboratory for Multimedia and Security. I have enjoyed discussing research with Jeff Dwoskin, Yu-Yuan Chen, Jakub Szefer, Cédric Lauradoux, Yedidya Hilewitz, Zhenghong Wang, Mahadevan Gomathisankaran, Nachiketh Potlapally and Peter Kwan. In particular, I thank Reouven Elbaz for his dedication during our joint work on memory integrity, and for his strong friendship.

I also wish to thank Najwa Aaraj, Miloš Ilak, Ronny Luss and Nebojša Stanković for their support during my graduate studies and for their precious friendship. I thank my good friends back home, Jean-Philippe Beaudet, Éric Fontaine, Simon Gignac, Nicolas Lafond, David Lévesque and Frédéric Villeneuve for being there when I needed a kind ear and a good laugh.

Last but most important, I want to thank my family. I am humbled by the love, care and understanding my fiancée Sonya has given me during my years in Princeton. My sister Mélanie has always been an example of determination that continues to inspire me. And, I am thankful for my loving parents, who taught me self-reliance, hard work and perseverance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Decades of improvement in computing throughput and reduction in device prices led to today's ubiquitous presence of computers in homes, businesses and other organizations. With recent innovations in battery technology and decreases in component size and power usage, we are now seeing a massive increase in the deployment of mobile computers for personal, business and scientific use. The ubiquitous presence of computing devices, interconnected via local networks and the global internet, drives an unrelenting push for the digitization of information as well as commercial and interpersonal transactions.

Unfortunately, computer security appears to lag this movement for digitization. New attacks are regularly carried out against the confidentiality of user data, the integrity of business systems or the copyrights of media providers. In past decades, the vast majority of computer security research has focused on software solutions for protecting security-critical software. These approaches can increase security by mitigating threats like network intrusions, but they are typically unable to cope with two major threats: compromised operating system (OS) code and hardware attackers. Both have almost unlimited privileges so they can bypass or disrupt any deployed software-only security solution. This means they can snoop on or corrupt any security-sensitive operation on the target platform. Many of the attacks we see today are caused by OS compromise, enabled by software vulnerabilities in the OS code base. In addition, sophisticated attacks seen today suggest that hardware threats may soon become a widespread problem, especially as more attackers are incentivized by the increasing value of information protected by computer systems. Ignoring these new threats is not an option; it would allow an increasing number of attackers to steal or corrupt data in computer systems owned by businesses, governments, institutions and private individuals.

In this thesis[1], we present Bastion, a hardware-software architecture protecting the launch and runtime of security-critical software against both compromised OS code and hardware attackers. Security-critical tasks are encapsulated in what we call *trusted software modules*. Bastion protects the integrity and confidentiality of the execution, storage, input and output of these trusted software modules, within commodity software stacks. These security services are provided by the Bastion Trusted Computing Base, formed of enhanced microprocessor hardware and a thin layer of hypervisor software,

[1] This work was supported in part by NSF CCF-0917134 and NSF CNS-0430487 (co-sponsored with DARPA)

running commodity operating systems. The Bastion microprocessor launches the hypervisor and protects its runtime. This trusted hypervisor can in turn leverage thehardware-protected environment to scale up protection to an arbitrary number of trusted OS and application modules in the virtual machines it manages.

1.1 New Threats

Traditionally, application writers have regarded the operating system as the trusted manager of all system resources, from memory, to disk storage and Input/Output (I/O) devices. It is assumed to share resources amongst applications, without interfering with their execution. With commodity OS code base sizes now reaching in the millions of lines of code [Ozment and Schechter 2006], the assumption is being revisited by a number of security researchers, e.g., [Lie et al. 2000], [Lee et al. 2005], [Suh 2005]. These large, complex, arbitrarily extensible and frequently updated software systems are very hard to inspect for correctness and security, as evidenced by the large number of patches released by OS vendors, due to security exploits compromising the OS code base. Across software vendors, OS code is released with large numbers of bugs, including security vulnerabilities, and it remains buggy and vulnerable long after its release [Ozment and Schechter 2006]. Writers of malicious software are increasingly quick at exploiting these OS vulnerabilities to install malicious software on targeted platforms, to snoop on or corrupt OS and application state [Levy 2004]. Attackers will then often hide their traces using so-called *rootkit* software to prevent detection [Hoglund and Butler 2005].

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 can thus launch hardware or physical attacks. For example, attackers with very low sophistication can easily recover a lost device or steal one and physically swap hard drives to extract the data it contains on another computer [Garfinkel and Shelat 2003]. Sophisticated attackers can also gain physical access to fixed-location computers such as workstations and servers by corrupting human assets in targeted organizations or by physically breaking into the building hosting the computer devices. Once an attacker has physical access, he or she can steal hard drives, substitute chips or peripherals with malicious ones (e.g., [Kuhn 1998]) or install probing devices on external buses (e.g., [Huang 2003]). Such attacks can allow the adversary to steal secrets, create fake data or execute malicious programs.

With these new threats in mind, how can a user get any assurance that secrets he provides to an application are safe from attack? Can a remote party entrust software running on the local platform with security-critical data? How are participants in a distributed computation supposed to verify one another’s contribution to the common task? The Bastion architecture presented in this thesis attempts to solve these problems by defining a secure computing platform that can implement different security policies.

1.2 Past Approaches

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. Some approaches 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. The goal of this thesis is to provide a solution that can be implemented by microprocessor vendors and distributed with the chip. We believe security capabilities should not be optional, added as an afterthought. Our trusted computing base is anchored in immutable hardware, hence it cannot be bypassed.

Some software solutions use hypervisors or Virtual Machine Monitors (VMMs), e.g., VMware [Sugerman et al. 2001] or Xen [Barham et al. 2003], to create isolated Virtual Machines. An untrusted application is run on top of the untrusted commodity OS in one Virtual Machine, while a trusted application is executed on top of a new trusted OS in a new trusted Virtual Machine. However, VMM solutions provide coarse-grain Virtual Machine compartments while we provide fine-grained secure compartments (within a virtual machine) to protect trusted software modules, similarly to [Chen et al. 2008]. Our compartments are created within the VM hosting the commodity OS to avoid the high cost of a world switch between VMs [Menon et al. 2005] when invoking trusted software. Since the VMM itself is vulnerable to hardware attacks, we go one step further than existing VMM-based security solutions and provide our hypervisor layer with strong cryptographic isolation. We leverage virtualization techniques to override the OS where necessary for security, and protect our hypervisor from hardware attacks as well as software attacks. As previous work on software security shows, the software architecture of a hypervisor can be structured to make the hypervisor trusted and allow for security verification [Karger et al. 1991].

Existing hardware techniques have limited scalability due to finite hardware resources, and constrained functionality due to lack of visibility into the software context. In addition, they (e.g., TPM [TCG 2006]) are usually still susceptible to hardware attacks. While a few proposed hardware solutions, e.g., XOM [Lie et al. 2000] and the Secret Protection (SP) architecture [Lee et al. 2005, Dwoskin and Lee 2007] 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 security domain, i.e., they can all access the same set of protected information. In this thesis, we present a scalable solution to efficiently supporting protection of large number of trusted software modules, in different security domains. Bastion mechanisms allow each trusted software module to establish its own secure input and output channels, inaccessible to other modules and even to the OS.

1.3 Thesis Contributions

The first contribution this thesis is the design of a set of hardware mechanisms in the microprocessor to enable the secure launch and runtime protection of a virtualization layer, i.e., a hypervisor or virtual machine monitor. Virtualization is quickly becoming a standard feature on mainstream computing systems and is increasingly used for security applications [Garfinkel et al. 2003, Sailer et al. 2005, Chen et al. 2008]. Our design of direct hardware mechanisms protecting this security-critical layer of software against new threats, presented in Chapter 4, is therefore a significant contribution to the field of secure computing.

Our second contribution is at the core of the Bastion architecture: a co-designed hardware-software Trusted Computing Base (TCB) for the protection of trusted OS and application modules within an untrusted, unmodified commodity software stack. TCB protection, described in Chapter 5, is requested by the modules themselves via a simple, but flexible *Trusted Programming Interface*. Our TCB offers a novel security service for software modules within a trust domain: secure inter-module collaboration, which allows for cross-authentication of callers and callees, and protected data exchange, despite untrusted surrounding software and hardware attackers.

The third contribution of this thesis, described in Chapter 6, is the design of two services, Tailored Attestation and Secure Persistent Storage, assisting with secure I/O within an untrusted commodity software stack. Our TCB can tailor attestation reports to attest solely to the integrity of our hypervisor and to the integrity of those trusted software modules needed for a given security-critical task. This greatly simplifies the task of the attestation report recipients, who need to determine whether the reported software can be trusted. Secure storage is bound to, and accessible by only the associated trusted software module so they can protect the integrity and confidentiality of long-lived secrets across platform reboots, despite a potentially compromised OS. Leveraging this module-specific attestation and secure storage, Bastion-protected modules can establish secure I/O channels with local or remote entities.

Our fourth contribution consists in the implementation (Chapter 7) and evaluation (Chapter 8) of a Bastion prototype on the OpenSPARC project [Sun 2008]. We modified SPARC processor hardware to implement new Bastion logic and synthesized the resulting design for an FPGA target. We also modified the SPARC hypervisor, formed of less than 50,000 lines of code, to implement new Bastion features and loaded the resulting hypervisor on the FPGA. With the TCB thus implemented, we ran applications with Bastion-protected modules on two unmodified commodity operating systems: Linux Ubuntu and Sun OpenSolaris. On Ubuntu, we also ran the vi text editor, enhanced with two trusted software modules implementing an ORCON (Originator CONtrol) security policy [Chen and Lee 2009]. The level of realism achieved by our implementation is greater than that of past hardware security architecture proposals, as it is based on real, high-performance general-purpose processor hardware, and commercial hypervisor software running unmodified commodity operating systems. This is an improvement on previous research proposals implemented only in software simulators or using simple embedded system hardware and software as a baseline.

Additionally, we also present a survey of existing memory authentication techniques and engines in Appendix A, with an emphasis on two proposals developed by the author of this thesis.

1.4 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides background for the subsequent chapters. It presents our security definitions, surveys the literature on past approaches to securing trusted software in adversarial environments, describes concrete attacks on real-world security architectures and presents the baseline processor we are assuming for the rest of this thesis. Chapter 3 provides an overview of the Bastion architecture, including the required processor and hypervisor modifications, and describes new classes of security applications enabled by our TCB. Chapter 4 provides background on virtualization and describes our approach to securing the launch and runtime state of the Bastion hypervisor. Chapter 5 describes the Bastion mechanisms for the establishment and runtime protection of the secure execution compartments hosting our trusted software modules. This chapter also presents how modules within a given trust domain can collaborate towards a common security-critical task despite being surrounded by untrusted software.

Chapter 6 defines the nature and scope of the secure I/O problem we are tackling in this thesis. It then describes the secure storage and attestation primitives and how they can be used to achieve secure user, disk and network I/O. Chapter 7 describes our implementation of Bastion on the OpenSPARC hardware and software, including the required changes to applications hosting trusted software modules. Chapter 8 evaluates the security, complexity and performance of Bastion, and provides a detailed application example demonstrating the functionality of Bastion. Chapter 9 concludes this thesis and summarizes opportunities for future work. Appendix A presents our survey of existing memory protection schemes, many of which can be substituted for Bastion's basic scheme, to improve its performance and complexity.

Chapter 2

Background

In this chapter, we provide some background on the notions necessary to understand the Bastion architecture and its contributions to field of computer security. We first present definitions for the multiple aspects of security discussed in this paper, including a trust model, a threat model and a list of Bastion security objectives. The chapter then continues with a discussion of past approaches to solving the problem of securing trusted software in an untrusted software stack. To make our threat model more concrete, we then provide examples for the attacks we are considering. Finally, we describe the baseline software and hardware platform we are considering in this thesis, upon which the Bastion architecture is to be built.

2.1 Definition of Security

In this section, we present our definition of security for the Bastion architecture. While the term security can be defined simply as "a state in which no bad things can happen", it is harder to concisely define "bad things". No one system can protect against the set of all possible "bad things" or *threats*. The set of threats under consideration must be restricted to a well-defined subset, justified with respect to the intended usage of the system. For example, a PDA intended to browse the web securely should not be concerned with the threat posed by a possible nuclear war, which could affect the availability of the device, the user and the web services of interest. Our *threat model* defines and details the full array of "bad things" that we consider in this thesis. To allow for certain "bad things" to be left out of the picture, we make and justify a set of assumptions in our *trust model* about which components of our system are trusted. Trusted components are axiomatically assumed to be outside the reach of attackers, natural events and other threats that could negatively affect the security of a Bastion system. We first spell out the Bastion security objectives.

2.1.1 Security Objectives

The security objectives of the Bastion architecture can be summarized into three security functions:

- Protect the Bastion hypervisor

- Protect trusted software modules
- Provide trusted software with secure I/O capabilities

Protecting of the Bastion hypervisor, trusted software modules and their I/O means either preventing or detecting the attacks listed in this section's threat model, given the assumptions made in this section's trust model. Each security objective is given its own chapter in this thesis, viz., chapters 4, 5 and 6 respectively.

2.1.2 Trust Model

In traditional computer systems, the entire computer box is trusted, including hardware and software. The processor, the motherboard, memory chips, extension cards, the disk and other peripheral hardware devices are assumed not to be malicious, to behave as defined in the manufacturer's specification. The operating system software is assumed to be correctly written, free from functional bugs or exploitable vulnerabilities. Application software is not fully trusted, as evidenced by the existence of privilege levels to isolate the OS from applications, and virtual address spaces to isolate applications from one another. However, each application is usually entrusted with a set of resources (e.g., files) without having to undergo formal cryptographic identification before authorization is given. A physical adversary overwriting an application program on the disk with its own malicious program could thus use the latter to hijack the resources assigned to the former.

In the Bastion architecture presented in this thesis, the microprocessor chip is the root of all trust. Upon platform reset, it is the only trusted component. We assume the microprocessor does not contain design or implementation flaws, hardware Trojans or viruses. We are considering a modern general-purpose microprocessor chip manufactured with multiple layers of metal and silicon. We assume the small feature size and the high complexity of the manufacturing process makes it impossible for an attacker to successfully probe any storage, routing or processing element within the microprocessor chip. Attackers are assumed to be unable to interfere with chip activities without destroying the chip. These assumptions are deemed reasonable, especially since contemporary research can thwart very sophisticated attackers with access to expensive equipment either by obfuscating certain circuit elements [Ishai et al. 2003] or by adding chip-level physical tamper detection mechanisms [Weingart 2000]. However, we do note that there is no silver bullet against all forms of invasive attacks; full-blown hardware tamper-resistance is still an open research problem [Anderson and Kuhn 1996]. We consider the memory chip, the disk and other peripherals to be untrusted, vulnerable to attacks (see Figure 3.1 in Chapter 3 for an illustration of our trust model). All software and data loaded from outside the microprocessor chip are untrusted.

As in most work on hardware-software security architectures (e.g., [Lie et al. 2000, Lee et al. 2005, Suh 2005, Dwoskin and Lee 2007]), side channel attacks are outside the scope of this thesis. We thus assume that our trusted microprocessor chip is immune to leakage and interference via side channels such as power consumption and electromagnetic radiation. Again, this is a reasonable assumption given that several circuit design techniques have been proposed to reduce the information that can be gathered via side channel attacks [Chari et al. 1999, Tiri and Verbauwhede 2005, Gebotys 2006]. Application of such techniques is orthogonal to the Bastion design, which is not tied to any one circuit implementation. We also do not consider information leakage via the address bus and memory access patterns in general. The theory underlying this problem has been studied [Goldreich 1987] and some practical solutions for general-purpose microprocessors have also been suggested for protection against address bus leakage [Zhuang et al. 2004, Duc and Keryell 2006] and cache-based side channels [Osvik et al. 2005, Wang and Lee 2007, Wang and Lee 2008]. Integration of such countermeasures to the Bastion architecture is left to future work.

As detailed in Chapter 4, Bastion security features rely on a trusted hypervisor. We do not require that there be a universally trusted hypervisor—i.e., a single hypervisor trusted by all parties. Rather, we assume that each party wishing to interact with a Bastion platform trusts at least one hypervisor. Different parties may trust different hypervisors. Parties could delegate the decision of whether to trust a hypervisor to a third party. This could be done, for example, by having a trusted online authority like Verisign [Verisign 2009] certify certain hypervisors to be trustworthy. The definition of a Public Key Infrastructure (PKI) is outside the scope of this thesis: when a PKI is needed, we assume it is already deployed and trusted by the concerned parties. In any case, it is up to the parties interacting with a Bastion platform to verify that it is running a trusted hypervisor. Trusted hypervisors are assumed to be free of bugs, so they are expected to be functionally correct (e.g., provide full virtual machine isolation) and to respect the Bastion security objectives described in Section 2.1.1. The combination of the Bastion microprocessor and the trusted hypervisor is the Bastion *Trusted Computing Base* (TCB).

As we discuss in Chapter 5, the Bastion TCB protects security-critical software modules in both the application and operating system layers. Again, there is no set of universally trusted software modules. Each remote party wishing to interact with a Bastion platform is assumed to have a precise definition of the modules it considers trustworthy. The set of trusted modules can vary from one party to another and the determination of a module's trustworthiness can be delegated to third parties. The Bastion platform allows all software to execute. The parties interacting with the platform are responsible for checking whether it is running the software modules they trust. Trusted software modules are assumed to be correctly written, free of bugs, including security vulnerabilities. Once designated as trusted by a remote party, a software module is expected to carry out its functionality while respecting the security objectives of that party.

The core idea behind this thesis is to allow for a general-purpose, highly flexible security architecture with a trusted code base as small as possible. Trusted software modules can be arbitrarily small and fully isolated from one another. Given the very high cost per line of code for high assurance verification of software systems [NICTA 2010], small, isolated software modules are more likely to be verified with high assurance than traditional software stacks, formed of large, monolithic pieces of software. The definition of a methodology for inspection of hypervisors and trusted software modules is outside the scope of this thesis. The goal of the Bastion architecture is to simplify code verification regardless of the code inspection methodology employed.

Strongly-typed languages (e.g., Java [Gosling et al. 2000], Python [Lutz 2006] and Lisp [Allen 1978]) and secure design methodologies [Kocher et al. 2004, Verbauwhede and Schaumont 2007] can be used to reduce the risk of unexpected code behavior that could lead to exploitable software vulnerabilities. While these methods can improve security of software systems, they do not provide a formal proof of correctness. Theorem-proving software [Kammller 2008] and special-purpose programming languages [Nipkow et al. 2002] can be used as part of a special development methodology that allows for automated checking of the final software product. The result of this approach is a formal proof that certain properties can be upheld by the verified piece of software. However, these methodologies require a significant overhead in development time for checking simple properties like pointer safety. Current methods do not lend themselves to checking complex security properties like "secure application launch" or "secure networking". In addition, these proofs are based on several layers of assumptions, including an idealized model of the underlying hardware, and an assumption that specific source code constructs map perfectly to their mathematical counterparts in the formal model used by the theorem-proving software. Therefore, we consider that proving the correctness and security of software components is still an open research problem. Designing an infrastructure for distribution and revocation of trusted software certificates is also outside the scope of this thesis.

To be useful, the Bastion TCB must communicate with the outside world. In Chapter 6, we define secure I/O mechanisms that allow a Bastion platform to securely carry out such communication. But if everything in the outside world is untrusted, there is no point trying to establish secure communications: everything sent or received by an outside entity could be leaked or corrupted by the entity itself. Even if cryptographic protection is applied to data exchanged, no outside world entity could ever be trusted to protect the underlying cryptographic key materials. Therefore, we need to assume there are some other trusted entities in the outside world to allow for meaningful applications. While some of the entities in this outside world could be Bastion platforms as well, we do not want to assume that the only trustworthy entities in the universe are Bastion platforms. We also want to avoid having to precisely define the set of hardware and software entities that Bastion platforms should trust. Therefore, we define the generic concept of a *trusted I/O endpoint*.

A trusted I/O endpoint is a hardware entity, possibly running some firmware or software, which can be trusted to correctly process data that is either sent to a Bastion platform as an input or received as an output from a Bastion platform. Trusted I/O endpoints are islands of trust in a sea of untrusted entities. They are immune to all forms of attacks: the software they run is correct, and they are protected against hardware adversaries. Trusted I/O endpoints are also capable of producing and verifying attestation reports, where a software and hardware platform attests to its identity. A Bastion platform wishing to interact with a trusted I/O endpoint only has to authenticate it, and establish a secure communication channel with it. This simplifying assumption is necessary to enable the simplest of security applications. For example, Bastion security mechanisms could be used to protect secure ping software, which creates unforgeable ping replies for a remote monitor computer. The remote monitor authenticates incoming ping packets and reports ping successes or failures to a local user. In this case, the remote monitor must be a trusted I/O endpoint, otherwise it could itself forge ping replies and lie to the user. Practical examples of trusted I/O endpoints are given in Chapter 6, where we discuss secure disk, user and network I/O.

2.1.3 Threat Model

Software Attacks: In this thesis, we consider both software and hardware attacks on the Bastion platform. All code and data is considered vulnerable to software attacks, which can be carried out by malicious OS or application code snooping on or corrupting security-critical software state in disks, memory, caches or registers. Software attacks can also be carried out by any piece of code that executes prior to the loading of the operating system, e.g., the Basic Input/Output System (BIOS) or the OS loader. Our definition of a software attack is limited to the software that runs on the Bastion processor of interest. Attacks carried out by software or firmware running on a peripheral hardware device, or on a special-purpose attacking device are considered to be hardware attacks. For example, corruption of sensitive disk data by malicious firmware running on a disk drive controller is a hardware attack. Corruption of memory bus data using a special-purpose hardware device—e.g., a programmable mod chip—controlled by attacking software running on a laptop is also considered to be a hardware attack. However, network-based attacks are considered software attacks, even though they are carried out by remote hardware. For example, a format string exploit in which a remote computer snoops on or corrupts data on a Bastion platform is considered a software attack. In this case, the leak or corruption is effectively carried out by compromised local software, which was vulnerable to format string exploits.

We group software attacks into two categories: *internal* attacks and *external* attacks. For a given software entity—e.g., an application or a loadable OS module—an internal attack is a corruption or a leak carried out by the entity itself. The attack is due to a bug triggered when the affected code construct is executed under specific conditions. These conditions usually happen rarely; otherwise, the bug would have been discovered during development, by common-case testing. These conditions can occur without any malicious intent by local software and remote parties. They may also be forced to occur by a clever, malicious remote party or local software entity, whose intent is to activate the bug by feeding it oddly formed inputs, to trigger the corruption or leak. In both cases, the leak or corruption occurs because of a bug within the target entity's code base, not because an outside entity directly reads or overwrites the target entity's state. This is why we call these events internal software attacks.

External software attacks are carried out by local software, which directly reads or overwrites the state of a target software entity—e.g. a malicious OS snoops on an application's data space. These attacks occur even if no bugs are present in the target entity's code base. In this thesis, trusted software is assumed to be bug-free and thus is not subject to internal attacks. However, it is subject to external software attacks.

Untrusted software running on a Bastion platform is subject to both internal and external software attacks. We consider that the binaries of the hypervisor and trusted software modules do not contain any secrets. In other words, Bastion is not concerned with the confidentiality of the code and initial data space of software. Bastion only needs to protect secrets generated by trusted software modules at runtime, and possibly stored to persistent storage or sent to a remote party. Secure distribution of secret software binaries—e.g., for the protection of the Intellectual Property (IP) contained in a code base—is considered an orthogonal research problem—e.g., see the XOM architecture [Lie and al. 2000].

Hardware Attacks: Hardware attacks group all attacks that are not software attacks. They can be carried out by malicious versions of hardware components composing a normal computing platform: memory chip, disk, I/O controller, etc. They can also be carried out by new components added to the platform solely to perform an attack: bus probes, attack chips (e.g., a mod chip on a gaming console), etc. These attacks aim to directly snoop on or corrupt the state of security-critical software running on the Bastion platform. They may also attempt to intercept and modify secure I/O communication between a Bastion platform and a trusted I/O endpoint. This means that malicious parties aiming to snoop on network I/O—e.g., an attacker with physical access to networking hardware, or a rogue technician at an Internet Service Provider (ISP)—are considered to be hardware attackers.

Attacks Types: We consider both passive and active attacks on confidentiality and integrity. In a passive attack, the adversary only observes platform state, while in an active attack, the adversary modifies platform state to reach his goals. Adversaries can easily carry out a passive attack on data confidentiality, i.e., observe bus, memory or hard drive data. They can also combine active and passive attacks to force software to leak unencrypted secrets—e.g., by corrupting the code base of security-critical software (an active attack), an adversary can get the software to write secret data to an unencrypted memory region, where the adversary can observe them (passive attack). Attackers can affect data integrity using any combination of spoofing (illegitimate modification of data), splicing (illegitimate relocation of data), and replay (substituting stale data) attacks. All of these involve an active attack modifying platform state. Some attacks also require observing platform state (passive attack) before waging the attack, e.g., a replay attack consists in modifying platform state so it corresponds to a previously observed (and recorded) state.

This thesis focuses on providing confidentiality and integrity to the data handled by security-critical software. It does not address the problem of privacy (anonymity of platforms and users) or availability. Within the Bastion threat model, availability is practically impossible to achieve, as a physical adversary could simply smash the processor chip with a hammer to deny service to trusted software. Even without physical destruction, a hardware adversary could render a Bastion processor useless in practice by systematically corrupting every memory transaction it sees on the processor-to-memory bus. Availability is thus easier to address within threat models where physical adversaries have less power. Within these models, the availability problem often becomes akin to the problem of reliability, which has been addressed thoroughly in the literature for faulttolerant systems [Musa 2004]. In most cases, reliability or availability of a hardware component is ensured with redundant instances of this component. Availability of software is often addressed by redesigning operating system software to keep track of resources consumed by applications more precisely, and limit resource usage to ensure critical applications have a guaranteed share of available resources. Applying such methods to the Bastion security mechanisms is an orthogonal problem which we do not address in this thesis.

2.2 Past Approaches

In this section, we examine existing solutions[3] to securing critical software in a commodity software stack with a potentially compromised OS. We distinguish between three approaches:

- add security features to the OS,
- measure and verify an unmodified OS or
- bypass the OS altogether.

2.2.1 Secure OS

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, and is responsible for scheduling and other forms of resource management [Ames et al. 1983, Karger and Schnell 2002]. OS 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. [Loscocco and Smalley 2001, Wright et al. 2002, Suh 2005, Zeldovich et al. 2006, Bratus et al. 2008]. 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 [Bratus et al. 2008] but only to enable detection of tampering with static code and data, not dynamic data (e.g. stack, heap). In the vast majority of cases, these are software-only approaches so they remain vulnerable to attackers with physical access to the device. For example, augmented file system access control policies in [Wright et al. 2002] could be circumvented by snooping on or corrupting file data through probing unprotected I/O buses or direct access to the disk.

Linux Security Modules (LSM) [Wright et al. 2002] is a project that enables software developers to create their own security module and load it into the Linux kernel. The core

[3] We first introduced the classification of approaches presented here in a prior publication [Champagne and Lee 2010]

idea behind LSM is to support arbitrary access control policies on system resources by allowing user-created security modules to mediate access to system resources. LSM modifies the commodity Linux kernel to insert hooks at critical points in system operations. User modules are allowed to register as security modules and request callbacks on a specific set of hooks. Prior to the execution of a hooked up operation, the kernel calls the user module, which can then apply an arbitrary access control policy to the operation about to execute. This mediation allows a user module to deny or restrict any resource access or system operation which violates the security policy implemented by the module. LSM also offers a new `security` system call for security-aware applications, to allow them to directly invoke special security operations implemented in a kernel security module. A variety of hooks are implemented by LSM. Task hooks allow security modules to mediate privileged operations carried out by monitored processes. Program loading hooks can verify the privileges given to programs being loaded during processing of the `execve` system call. File system hooks allow for enhanced access control on super block, inode and file constructs, while network hooks enable monitoring of socket-related operations. Finally, module and system hooks allow, respectively, for the monitoring of kernel module operations and generic system operations, e.g., `setuid`.

SELinux [McCarty 2004] is a major secure operating system project that exploits the infrastructure set up by LSM. It aims to make the Linux operating system more secure by allowing for a flexible Mandatory Access Control (MAC) policy to be implemented in addition to the traditional Discretionary Access Control (DAC) policy used in Linux. DAC policies are judged too coarse-grained in the assignment of privileges. Moreover, DAC is vulnerable to malicious or buggy software. SELinux is implemented as an LSM. This LSM allows platform administrators to specify, via a configuration file, a customized mandatory access control policy on platform resources. The SELinux framework implements a combination of Role-Based Access Control, type enforcement and (optional) multi-level security. For type enforcement, the configuration file allows the platform administrator to define domains of processes and types of objects, which are to be subject to runtime checks—e.g., controlled entry points into domains, controlled transitions between domains, binding of domains to a set of object types, etc.

The HiStar operating system presented in [Zeldovich et al. 2006] is built from scratch with security in mind. The main goal of HiStar is to reduce the amount of code that must be trusted from the entire OS code base to the kernel only. Kernel abstractions are built upon six basic object types: threads, address spaces, segments, gates, containers, and devices. Each object is linked to a unique identifier, a label used for information flow tracking, a storage quota, some user-defined metadata and flags. The kernel is designed to enforce the following property: "The contents of object *A* can only affect object *B* if, for every category *c* in which *A* is more tainted than *B*, a thread owning *c* takes part in the process." This ensures that very strict information flow restrictions are systematically enforced, without the need for application developers to understand how their objects interact with the rest of the system. The resulting kernel is less than 20,000 lines of code, a very small kernel when compared to commodity operating systems. Unix-like abstractions are implemented on top of the HiStar kernel, in user space. The bulk of modifications occur in a HiStar compatibility layer implemented below LibC, with about

10,000 lines of code. With this ported LibC, file system, processes, gate calls, signals and networking can be implemented to work with HiStar abstractions.

AEGIS [Suh 2005] is a hardware-software security architecture which focuses on providing memory compartments and attestation to security-critical application software. A new security kernel in the OS and AEGIS mechanisms in the hardware collaborate to protect these compartments against both software and hardware attacks. Dedicated hardware registers managed by the operating system define ranges of physical memory that are either unencrypted, encrypted, or encrypted and authenticated. Memory protection is provided by on-chip engines for encryption and hashing (as part of a Merkle Tree memory integrity tree scheme [Gassend et al. 2003]). In software, memory compartments are enforced by the operating system, which reserves certain virtual and physical address space segments to security-critical applications that request them. AEGIS includes an attestation mechanism which produces a report, signed by the processor's private key, identifying the current security kernel and the software running in one of the application compartments.

2.2.2 Verify OS

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 [Sailer et al. 2004, Marchesini et al. 2004, Cihula 2007]. These techniques compute a cryptographic hash to fingerprint the initial data and code state of the operating system; 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. This sealed storage as well as the attestation capability are provided by a hardware Trusted Platform Module (TPM) [TCG 2006] chip containing immutable keys. Although TPM keys can be used to protect disk data while the device is powered off, TPM-based systems remain vulnerable to attackers with physical presence during runtime [Kauer 2007, Halderman et al. 2008]. Even without physical attackers, 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 [Bratus et al. 2008].

The TPM [TCG 2006] is the most widely distributed security co-processor chip for general-purpose computing platforms. It is used primarily to enable measurement, reporting and verification of software stacks, including the operating system at the request of remote parties. The TPM for Personal Computers is a low cost, hence low-resource, standalone chip designed to be attached to the motherboard, on a bus accessible to software running on the microprocessor via standard I/O operations, typically the Low-Pin Count (LPC) bus. The TPM includes some hardware crypto accelerators, volatile and non-volatile registers, as well as some firmware and an execution engine. Only asymmetric cryptography and hashing functionalities are implemented in hardware, due to export restrictions on devices capable of performing bulk symmetric key encryption. The main role of the TPM chip is to securely store cryptographic measurements of the identities of components forming the software stack running on the microprocessor. TPM allows for two distinct ways to measure a software stack.

With the *Static Root of Trust* method, the Basic I/O System (BIOS) boot block, the first piece of code to be executed upon a processor reset, is responsible for carrying out the first software measurement. This measurement is a cryptographic hash taken over the state of the next layer of software to run, i.e., the rest of the BIOS, and is stored by the BIOS boot block in the TPM chip using a TPM command. The boot block then jumps to the measured BIOS, which is responsible for measuring the next layer of software to execute, typically the OS boot loader, responsible for loading the operating system kernel. Again, this measurement is stored in the TPM via an I/O transaction. This chain of measurements continues until the entire software stack is loaded, and identified in secure TPM registers. The idea is that if any component in the software stack is corrupted, it can be detected by looking up its measurement in the TPM registers. Once a measurement has been registered in the TPM, its presence cannot be erased until the platform is reset.

A reset causes the entire software stack to be wiped out, and the measurement procedure to restart from the BIOS boot block. Since each software layer other than the trusted BIOS boot block is measured before it is executed, a corrupted layer cannot alter its own identity in the TPM, hence it is detectible. When the number of measurements is larger than the number of TPM measurement registers, software can reuse registers by having the TPM create compounded measurements. A compounded measurement M_C is computed by the TPM over the existing state S of a TPM measurement register and a new measurement M using a compressing cryptographic hash function: $M_C = H(S \| M)$. M_C then becomes the new state of the measurement register[4]. Because of the properties of cryptographic hash functions, this method of compounding measurements maintains a non-malleable trace of measured software layers, even if a corrupted layer produces its own fake measurements. To recover the set of measurements used to create a compounded measurement, the software stack must keep a repository of full-length measurements. This repository does not require any special protection since its content is made tamper-evident by the compounded measurements, securely stored on the TPM chip.

While the Static Root of Trust method only required a change to the BIOS firmware, the *Dynamic Root of Trust* method requires changes to the microprocessor hardware. In this method, a regular software stack can be deployed without any measurement sent to the TPM. It is only after a special microprocessor instruction is executed, e.g., GETSEC[SENTER] on Intel processors with Trusted eXecution Technology [Intel 2007], that software stack measurements are sent to the processor. This new instruction starts the loading and measurement of a new software stack that is to be isolated from the existing, unmeasured, software stack. The main motivation is to allow for the launch, at any point in time, of a trusted execution environment (the measured stack) for security-critical software, even if the platform was booted with an untrusted software stack (the

[4] In practice, a single measurement is also stored as a compounded measurement, using the initial zeroed state of the register.

unmeasured stack). To allow for the two software stacks to co-exist, designers of TXT had initially envisioned encapsulating on-the-fly the unmeasured stack in a Virtual Machine, slipping a trusted Virtual Machine Monitor (VMM) under it, and loading the trusted, measured stack in a different Virtual Machine. The TPM was to receive the measurements of the VMM and all components of the trusted software stack. In practice, however, Dynamic Root of Trust technology has not yet been used to dynamically encapsulate a software stack running on bare hardware into a virtual machine running on a VMM. Instead, the GETSEC[SENTER] instruction is used to launch a trusted VMM, following platform set up by an untrusted, unmeasured BIOS.

To allow for measurement of the trusted VMM, the instruction fetches and authenticates a secure initialization routine which executes on-chip, in a locked down part of the cache. The routine measures a VMM loader and sends the measurement to the TPM. The measured VMM loader then measures the VMM itself. The measured VMM can then initiate the measurement of the trusted software stack. The TPM specification does not define a methodology for measuring the hundreds of software components and configuration files involved in the construction of a commodity software stack. To ensure that every component is properly measured as it is loaded, one needs to modify the operating system to perform timely measurements, store them in the TPM in compounded form and log them in full-length in a measurement repository. The Integrity Measurement Architecture (IMA) [Sailer et al. 2004] was developed to provide such an infrastructure, within the Linux operating system.

To protect the trusted VM from DMA transactions initiated by the untrusted VM, technologies such as Intel TXT provide special hardware to separate the I/O transactions of different VMs. Since this separation is similar to what is being done by the Memory Management Unit (MMU) for separating virtual address spaces, this new hardware is often called an I/O MMU.

TPM provides secure storage and attestation using the values in its measurement registers, regardless of the measurement method used. Software can request that an encryption key protecting the confidentiality of a secure storage area be bound by the TPM to the current measurements in TPM registers. With a key protected by its Storage Root Key, the TPM then encrypts the key with metadata specifying the measurements the key is bound to. This binding operation is called *sealing*. To unseal an encryption key, software must present the encrypted blob to the TPM for decryption. The TPM will only decrypt the blob if the measurements of the software requesting the unsealing of the blob are the same as the sealer’s measurements. This guarantees that a secure storage area created by a good software stack cannot be decrypted by a corrupted software stack.

To protect the integrity of a secure storage area, software can compute a cryptographic hash over the area and request that the TPM store this hash in its non-volatile storage area[5]. Like the key, the hash can be bound to a specific set of software measurements to ensure that it cannot be accessed and altered by a compromised

[5] The following procedures for secure storage integrity are not mandated by the TPM specifications. It is up to each to TPM user to decide whether or not to implement storage integrity protection.

software stack. To attest to its identity, a software stack requests an attestation *quote* from the TPM. The TPM handles this request by compiling a report containing a nonce and the various measurements identifying the components of the software stack. It then signs the report with a private key that is only accessible to the TPM itself. This report is returned to the software stack, which can then forward it, along with the list of full-length measurements, to a remote party.

The TPM chip is meant to be an inexpensive security anchor that can easily be integrated to any general-purpose computer. As a result, it was designed to cost only a few cents by providing it with only little computing power and execution memory. These low resources being insufficient to communicate on complex high-speed buses, the TPM chip is thus traditionally added to computers via the slow, Low-Pin Count (LPC) bus. A common problem with all TPM-based approaches is thus that they are unable to carry out security operations at a high frequency, due to their reliance on the external TPM chip.

2.2.3 Bypass OS

Finally, some architectures bypass the commodity operating system altogether to protect a security-critical application component using either an enhanced microprocessor or a trusted hypervisor. These tend to focus on memory compartmentalization so most do not offer sealed storage or attestation capabilities. When they do, these services are either restricted to a single trusted software module or multiple modules belonging to one security domain [Dwoskin and Lee 2007] or they require the use of a slow TPM chip [Garfinkel et al. 2003, England et al. 2003, McCune et al. 2008] or an expensive[6] PCI-based coprocessor peripheral [Dyer et al. 2001]. There are two broad categories of techniques for bypassing a commodity OS: 1) move security-critical tasks to a trusted operating system running concurrently to the commodity OS, within a separate execution environment [Garfinkel et al. 2003, Alves and Felton 2004, Kwan and Durfee 2007, Anderson et al. 2007, McCune et al. 2008] or 2) isolate security-critical tasks running on top of the commodity OS [Lie et al. 2000, Lee et al. 2005, Dwoskin and Lee 2007, Chen et al. 2008, Dewan et al. 2008].

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 [Menon et al. 2005] and prevent scalability. In addition, the trusted OS hosting the critical tasks imposes a tradeoff between security and functionality: a very small, hence easy to verify, OS cannot offer many functionalities to the security-critical tasks it hosts, while the complexity a large and fully-functional OS may make it as vulnerable to software bugs as the commodity OS. In the second category, software techniques [Chen et al. 2008, Dewan et al. 2008] secure critical tasks within the commodity software stack without creating new environments, but cannot

[6] We use the term expensive to compare the TPM chip (less than a dollar) to the PCI-based coprocessor (thousands of dollars). We note, however, that in certain applications, thousands of dollars can represent an insignificant cost.

protect against attackers with physical presence. While providing better security against physical attackers, the hardware-based techniques [Lie et al. 2000, Lee et al. 2005, Dwoskin and Lee 2007] in this category remain vulnerable to memory replay attacks on dynamic data and are restricted in their scalability by limited hardware resources (e.g. [Lee et al. 2005] can only support one security domain at a time).

Terra [Garfinkel et al. 2003] was one of the first architectures to suggest leveraging virtualization and TPM-like technologies to bypass the untrusted OS. The goal of Terra was to establish multiple execution environments with different security requirements, in different virtual machines. VMs are isolated from one another by a trusted virtual machine manager, as if they were running on separate hardware platforms. The trusted VMM presents a TPM-like interface for attestation and secure storage to its VMs. The security of these virtual security services needs to be rooted in actual security hardware, possibly a TPM, whose single-user functionality can be virtualized by the trusted VMM. More formal work has also been done to fully virtualize the TPM so that virtual TPM instances can offer the entire set of TPM functionalities to virtual machines [Berger et al. 2006]. An architecture similar to Terra, the Next-Generation Secure Computing Base (NGSCB), was proposed by researchers at the Microsoft corporation [England et al. 2003]. The idea is to have a virtual machine monitor manage two virtual machines, one untrusted machine with a commodity Microsoft operating system, with all its complexity, and another, trusted machine, running security-critical software on a smaller, trusted operating system called a *Nexus*. Rather than a virtual machine per security application, NGSCB envisioned that all security applications (called *agents*) would share the trusted Nexus. As in Terra, a TPM-like chip was to provide security services like secure storage to every agent.

ARM TrustZone [Alves and Felton 2004] is a technology for embedded systems, also aimed at creating a trusted and an untrusted execution environment using virtualization. In this case, however, the microprocessor does not offer generic virtualization support for a large number of virtual machines, e.g., Intel VT technology [Intel 2006]. Rather, the microprocessor hardware is designed to support only two environments. The secure environment is given access to more hardware resources than the untrusted environment, e.g., a secure interrupt controller, an on-chip execution memory, special key registers, a secure UART, a random number generator, etc.

Flicker [McCune et al. 2008] uses Intel's implementation of the dynamic root of trust method to establish, on-demand, a minimal trusted execution environment for a piece of critical application code. Rather than deploy a full-blown trusted virtual machine supported by a trusted virtual machine monitor, Flicker uses the GETSEC[SENTER] instruction to launch a minimal application runtime that is just functional enough to support the piece of critical application code it must run. Virtualization hardware is used to ensure that this runtime and the application code are isolated from the untrusted software stack from where they were invoked. In this case, TPM-based secure storage and attestation services are bound solely to the runtime and the piece of application code. The main drawback of this minimalist approach is that without a secure context manager like a VMM, Flicker must destroy and recreate a secure environment every time a trusted piece of application is invoked. The invocation of GETSEC[SENTER] and TPM operations on every switch to a secure context cause very high overheads in CPU time.

The XOM architecture [Lie et al. 2000] proposes hardware support for the protected execution of trusted software running atop an untrusted commodity operating system. Although XOM requires some modifications to the operating system in order to support the new hardware, it does not trust the OS in establishing the protected application execution environments. The main goal of XOM is to support digital rights management by allowing for the distribution of encrypted program binaries targeted for a particular processor. For example, a software vendor can encrypt its program such that only the XOM processors of paying customers are able to decrypt the software. The program is encrypted using a session key, which the vendor encrypts with the XOM processor's public key before distributing the software. Upon launching a protected application, the processor decrypts the session key and loads it into a hardware table. A new instruction allows entering a XOM compartment, which triggers the decryption of the instruction cache lines using the session key. New instructions are provided to encrypt and integrity check data load and store operations. Upon a context switch, some protected application state is saved in on-chip data arrays while application register state can be encrypted and written off-chip.

The SP architecture [Lee et al. 2005] focuses on protecting user secrets rather than the software vendor's digital rights as in XOM [Lie et al. 2000]. The secrets belonging to each user of an SP device are protected by a set of keys, organized in a hierarchical key chain, rooted by a User Master Key (UMK). Users input their UMK in an SP device using a secure I/O mechanism, assumed to be available and trustworthy. SP devices protect the UMK—and by extension the user's secrets—by enforcing that a Trusted Software Module (TSM) be the only software entity allowed to access the dedicated UMK register. Each SP processor is bound to a specific TSM, whose integrity is protected by a Device Master Key (DMK) rooted in a non-volatile hardware register. Each cache line of the TSM is signed by a DMK-based Message Authentication Code (MAC) so that TSM code can be tamper-evident and distinguishable from untrusted code. The TSM is assumed to be installed during a trustworthy installation procedure. During runtime, untrusted software can request operations on the user's keychain by invoking the TSM via a special instruction switching the processor into Concealed Execution Mode (CEM). The UMK register is only accessible in CEM, where the processor only executes TSM code, whose code cache lines are integrity checked as they are brought on-chip. Secret information manipulated by the TSM is sent in encrypted form to memory using a new `secure_store` instruction, and fetched back in decrypted form using a `secure_load` instruction. Whenever a TSM running in CEM gets interrupted, the processor encrypts and hashes its register state so the intervening operating system handler is unable to snoop on or corrupt intermediate TSM results.

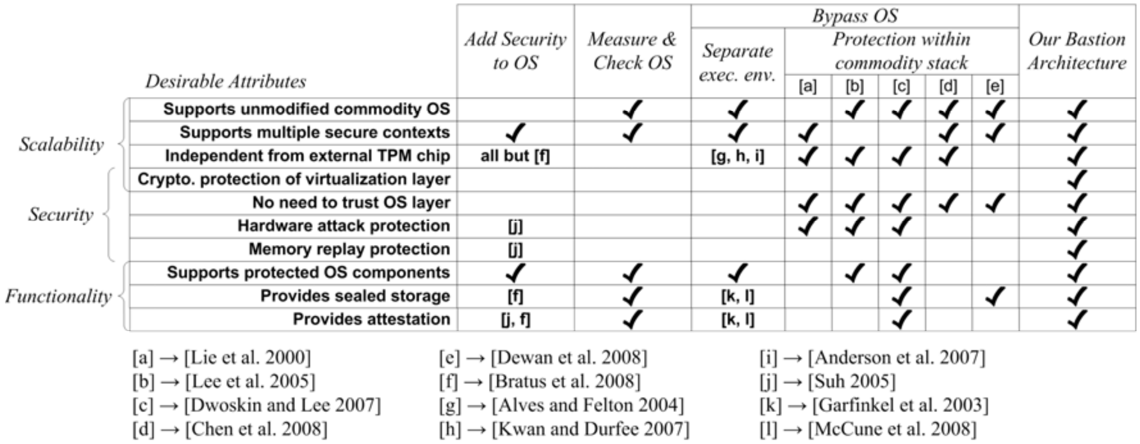
SP authority mode [Dwoskin and Lee 2007] shifts the original SP focus from user secret protection to protection of secrets belonging to a remote authority. In this case, the authority is responsible for initializing the SP device with its own TSM, trusted to protect authority or third-party secrets. Authority secrets are either in local or remote storage on the SP device, and third-party secrets are distributed across a set of remote parties that trust the authority. Secrets under the control of the local platform are protected by a secure storage tree rooted in the processor hardware. This is done by keeping in non-volatile processor registers a key and a hash that respectively encrypt and fingerprint the local storage. The SP authority mode device restricts access to these registers to the signed TSM, which can then decrypt and verify the local authority secrets. To access remote secrets, the SP device must attest to the identity of its TSM to trusted remote parties, using an attestation key. This key is stored in a non-volatile processor register by the authority, during initialization of the device. As in the original SP, the untrusted software stack can gain indirect access to authority secrets by invoking, using dedicated instructions, TSM code running in Concealed Execution Mode. The main limitation of both versions of the SP architecture is that only TSMs from the same trust domain (i.e., they all have access to the protected secrets) can be installed on the device at any one time.

Overshadow [Chen et al. 2008] is a virtualization-based, software-only approach to bypassing an untrusted operating system in providing security to application software. Its main goal is to ensure the confidentiality and integrity of application execution by encrypting and hash verifying the pages composing its virtual address space. This is done by a trusted Virtual Machine Monitor (VMM), upon invocation of a new hypercall by the application. Whenever the application executes, the VMM decrypts and integrity verifies the pages accessed by the application. When the OS preempts the application and takes control of the CPU, the VMM re-encrypts all decrypted pages and fingerprints them with a cryptographic hash function. Communication between the protected applications and the operating system is made possible by the VMM's marshalling of system call parameters. This approach consists in having the VMM intervene upon a system call to copy invocation parameters—e.g., system call identifier, data buffers, etc—from a protected application page to an unprotected virtual page.

An architecture based on Intel VT-x architecture was also proposed to provide isolated application execution environments despite possible untrusted OS code [Dewan et al. 2008]. The main difference with Overshadow is that it creates separate shadow page tables for protected application code, so that this code executes out of a machine memory region inaccessible to untrusted code. Protected programs can also request encryption and decryption of a secure storage area to the hypervisor, which allows them to protect certain persistent storage files against compromised OS or application code.

The Bastion architecture presented in this thesis adopts the strategy of bypassing the commodity operating system to provide to security-critical tasks isolated execution compartments with attestation and 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 many classes of hardware attacks. As shown in Table 2.1, it is also the first security solution to provide strong protection of the virtualization layer against hardware adversaries.

Table 2.1. Comparison of past approaches with our Bastion architecture



	Desirable Attributes	Add Security to OS	Measure & Check OS	Bypass OS							Our Bastion Architecture
				Separate exec. env.	Protection within commodity stack						
					[a]	[b]	[c]	[d]	[e]		
Scalability	Supports unmodified commodity OS		✓	✓		✓	✓	✓	✓		✓
	Supports multiple secure contexts	✓	✓	✓	✓			✓	✓		✓
	Independent from external TPM chip	all but [f]		[g, h, i]	✓	✓	✓	✓			✓
Security	Crypto. protection of virtualization layer										✓
	No need to trust OS layer				✓	✓	✓	✓	✓		✓
	Hardware attack protection	[j]			✓	✓	✓				✓
	Memory replay protection	[j]									✓
Functionality	Supports protected OS components	✓	✓	✓		✓	✓				✓
	Provides sealed storage	[f]	✓	[k, l]			✓		✓		✓
	Provides attestation	[j, f]	✓	[k, l]			✓				✓

[a] → [Lie et al. 2000]
[b] → [Lee et al. 2005]
[c] → [Dwoskin and Lee 2007]
[d] → [Chen et al. 2008]
[e] → [Dewan et al. 2008]
[f] → [Bratus et al. 2008]
[g] → [Alves and Felton 2004]
[h] → [Kwan and Durfee 2007]
[i] → [Anderson et al. 2007]
[j] → [Suh 2005]
[k] → [Garfinkel et al. 2003]
[l] → [McCune et al. 2008]

2.3 Concrete Attacks

2.3.1 Cold Boot Attack

TPM-based systems have been proven vulnerable to several types of attacks. These attacks do not reveal any flaw in the TPM's design, but rather the weakness of the TPM threat model, which assumes that physical attacks are not possible. The Cold Boot attack [Halderman et al. 2008] shows how a TPM-sealed symmetric key can easily be extracted at runtime, from the memory state of the software stack to which the key was bound. As the good software stack carries out decryptions, it loads the plaintext version of the key—which it obtained from the TPM—in main memory. The Cold Boot attack consists in exploiting the remanence of RAM chip to extract this key from main memory by power cycling the platform so it is rebooted with a malicious software stack. At this point, the TPM would refuse to decrypt the sealed symmetric key blob since the identity of this malicious stack differs from the identity of the stack that sealed the key. However, the malicious stack is able to read the plaintext key bits that still persist in the RAM chip. The entire key can be recovered this way, allowing the attacker to decrypt any data that was protected with this encryption key. Another version of the attack uses a cooling spray to enhance the RAM chip's remanence, giving the attacker the time to transfer the chip to another device where the key is read out. The Cold Boot attack was used to defeat commercial disk encryption systems, including Microsoft Vista's BitLocker [Halderman et al. 2008].

2.3.2 TPM Reset Attack

One assumption underlying the design of the TPM chip is that the measurements it stores reflect the identity of the software stack running on the processor the TPM is attached to. The assumption requires that the TPM only gets reset when the processor and memory are reset. Measurements in the TPM are then only wiped out when the software stack running on the processor is itself destroyed by a processor and memory reset. If a malicious software stack is loaded on the platform, its identity will remain in the TPM measurement registers until the stack is destroyed by a processor reset. Thus when a remote party requests an attestation quote from the TPM, the TPM automatically reports the presence of the malicious software.

In practice, however, TPM chips are connected to a slow speed bus, the Low-Pin Count (LPC) bus, whose signals a physical attacker can easily manipulate. In particular, the bus reset signal, which causes all devices connected to the LPC bus to initiate their reset sequence, can easily be asserted by driving the corresponding electrical line to the ground. When this is done by a physical attacker, the TPM wipes out its measurements as part of the reset sequence. Other devices connected to the LPC bus also reset, e.g., the keyboard and mouse, but the CPU itself is not reset. The implication is that a malicious software stack can remain in control of the CPU while the attached TPM contains no measurements. The malicious stack can then feed the TPM measurements corresponding to a trusted software stack, as if the CPU had been reset and was loading such a stack. Subsequent attestation quotes by the TPM report a good software stack even though a malicious software stack is in control of the CPU. This TPM Reset Attack has been implemented on a real-world TPM-enhanced platform [Kauer 2007].

2.3.3 Blue Pill Attack

The Blue Pill attack [Rutkowska 2006] consists in forcefully encapsulating an unwitting operating system running on bare hardware into a virtual machine managed by a malicious hypervisor. The assumption is that the platform is running a Windows operating system on a platform with hardware virtualization support. The goal of the attack is to install a piece of malware that cannot be detected by OS-based rootkit detectors. (Rootkits are a class of malware whose purpose is to hide the traces of the installation and execution of another piece of malware). Blue Pill claims that the operating system is unable to detect it is being virtualized, hence OS-based rootkit detectors are unable to detect the hypervisor-based malware. (This claim has been disputed by security researchers [Garfinkel et al. 2007], who argue that virtualization can always be discovered by a careful analysis of timing or resource usage.) To install the malicious hypervisor, an application-level component of Blue Pill must corrupt OS kernel code. On platforms with hardware support for virtualization, this privileged code can then execute a processor instruction for hypervisor launch, with a reference to the malicious hypervisor as an operand. To corrupt kernel code, applications use an operating system Application Programmer Interface (API) call that gives them raw access to the hard drive. This API is used to overwrite paged out kernel code with malicious code. When the corrupted code is paged back in and executed, it invokes its malware hypervisor installation routine.

2.3.4 SMM Attack

Intel processors have a special System Management Mode (SMM), which is more privileged than all other execution modes, including hypervisor mode. This means that software running in SMM can arbitrarily corrupt virtual machine monitors, even if they were launched using TXT and the Dynamic Root of Trust. Software running in SMM is typically a set of handlers responding to events like CPU overheating or major hardware faults [Intel 2009]. It is loaded upon launch by the BIOS and does not execute unless an exceptional event triggers a Secure Management Interrupt (SMI), which the hardware routes to SMM software.

Intel security architects had envisioned that SMM software, which varies from one motherboard to the next, would be monitored by a special piece of trusted software, to prevent it from corrupting or snooping on VMM software. Effectively, this piece of software was to virtualize the system management mode to allow for monitoring of SMM software operations. This SMM monitor was thus to be measured as part of the trusted computing base during the establishment of the dynamic root of trust. However, no SMM monitor is measured or deployed by Intel TXT technology because such a monitor has not yet been implemented [Wojtczuk and Rutkowska 2009]. As a result, SMM code can be compromised to install malware that is able to corrupt virtual machine monitors launched via the GETSEC[SENTER] dynamic root of trust instruction [Wojtczuk and Rutkowska 2009].

2.3.5 XBOX Attack

The reverse engineering of the Microsoft Xbox security system is a telling example of how physical attackers are a real threat to security-critical hardware [Huang 2003]. In this attack, a custom-built device is attached to a motherboard bus between the memory controller chip and the I/O controller chip. This attacking device is a data logger used to extract a secret boot sector used by the Xbox to initiate its secure bootstrap sequence. As opposed to a regular computer boot sector stored in the first section of the BIOS Flash ROM, this secret sector is stored inside the I/O controller's circuit. The Flash ROM actually contains a decoy boot sector. The data logger was used to extract the encrypted version of the boot sector as it transited on the bus during a boot sequence. It was then analyzed to find a plaintext encryption key that could be used to decrypt the boot sector. The decrypted boot sector in turn provided enough information for hackers to create their own boot sectors that would be able to execute on the Xbox and, for example, bypass Digital Rights Management checks on loaded software.

2.3.6 Cipher Instruction Search Attack

The cipher instruction search attack [Kuhn 1998] demonstrates the vulnerability of bus encryption schemes to hardware attackers. The attack targets a security processor whose threat model, like the one in this thesis, assumes external buses are subject to physical attackers. To counter this threat, the processor encrypts all bus traffic to and from the external memory chip. To attack the encryption scheme and reveal the mapping between processor instructions and their ciphertext, the attacker constructs a hardware attacking device interposing itself between the CPU and RAM chips. The attacking device logs traces of memory traffic and can correlate them with interactions the executing software is having with the outside world, e.g., byte output on the parallel port. From this information, the attacker guesses possible ciphertexts for specific instructions and feeds them to the processor. The attacker then observes the effect of the guessed ciphertext on the outside world, e.g., is a byte with value 0x12 output on the parallel port? Using this approach, the cipher instruction search attack is able to construct a table mapping processor opcodes and operands to ciphertext instructions. This allows the attacker to analyze executing code in plaintext by snooping encrypted bus traffic, or insert arbitrary malicious code into the encrypted instruction stream. This codebook attack shows that without freshness tokens in the encryption primitive, attackers may be able to break cipher used on a narrow set of plaintext payloads. It also demonstrates the need for integrity-checking mechanisms to ensure only genuine code executes on a security processor.

2.4 Our Baseline Platform

We assume that our baseline CPU has virtualization support such as Intel VT-x technology [Intel 2006], AMD-V [AMD 2008a] or UltraSPARC Hyperprivileged edition [Laudon 2006]. 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 also 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. In this thesis, we use the terms interrupt and trap interchangeably.

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* [Adams and Agesen 2006] or *nested paging* [AMD 2008b], the two main techniques for memory virtualization. Mainstream virtualization-enabled microprocessors support both techniques [Intel 2006, AMD 2008a]. 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) miss events, 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 that get 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. We assume a virtually tagged L1 cache and a physically tagged L2. This thesis only considers the case of a uniprocessor. Applying Bastion to multi-threaded, multi-core and multi-chip processor systems is future work.

Chapter 3

Architecture Overview[7]

The Bastion Trusted Computing Base (TCB) is formed of enhanced microprocessor hardware and hypervisor software. The goal of the TCB is to protect the launch, execution and Input/Output (I/O) of security-critical trusted software modules within untrusted commodity software stacks. Secure I/O is achieved with two mechanisms: secure storage and attestation. The Bastion microprocessor protects the launch, the runtime memory state and the secure storage area of our security-enhanced hypervisor against both software and hardware attacks. The hypervisor uses this hardware-protected environment to scale protection up to an arbitrary number of trusted software modules. Each of these modules is provided with controlled launch, an isolated execution environment and its own secure storage area, all protected against compromised operating system code and hardware attackers. The hypervisor also protects the preemption of modules on interrupts to ensure all control flow in and out of modules is secure. The hypervisor leverages a hardware-based attestation mechanism to attest to the identity of trusted software modules in a trust domain, to a local or remote entity. Figure 3.1 depicts an example application of Bastion, where three application or OS modules (A, B, C) run on top or within commodity operating systems. Each module is associated with its own secure storage area on the untrusted disk.

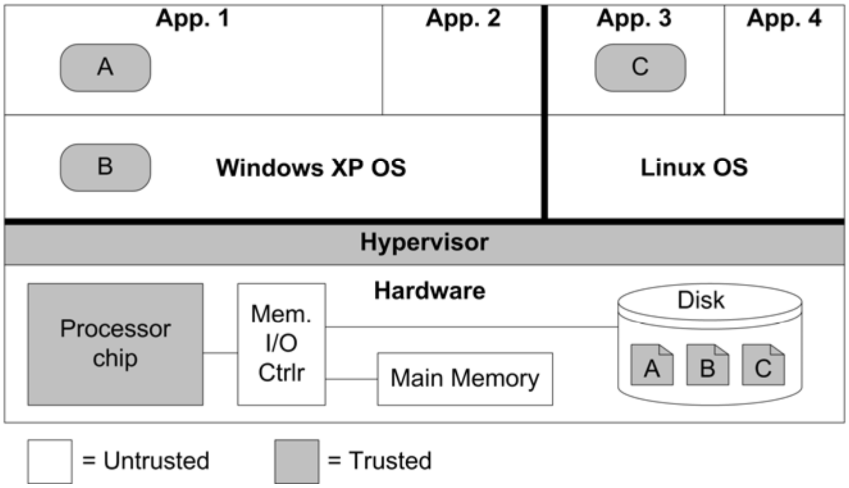


Figure 3.1. Application of Bastion for three software modules A, B and C

[7] The contents of this chapter are based on a paper currently under submission [Champagne and Lee 2010b]

In this chapter, we present an overview of the Bastion architecture described in this thesis. We first describe how we create and maintain secure execution compartments for security-critical software in the OS and application layers. Then we give an overview of the processor mechanisms required to protect our Bastion hypervisor. Then we describe the hypervisor software and processor hardware enhancements required to implement Bastion. Finally, we present three novel classes of security operations enabled by the Bastion architecture.

3.1. Secure Execution Compartments

In this section, we define our key architectural concepts from a software perspective. These are the abstractions and interfaces that programmers of trusted software must work with to obtain TCB protection and gain access to trusted computing services. These abstractions and interfaces form the *Trusted Programming Interface (TPI)*.

At the core of the Bastion architecture is the concept of the software module. All software running on the CPU is part of a module: security-critical software components are encapsulated into *trusted software modules* while the remaining, untrusted parts of the software stack are in a generic untrusted module called *module zero*. Each trusted software module is accompanied by its own *security segment*, a data structure defining the module's code and data space, as well as the protection it must be provided with at runtime. Trusted software modules can invoke security services such as secure storage and attestation via the *Trusted Program Interface (TPI)*, which are calls to the hypervisor. Module zero is not protected by the TCB and its code and data space can expand arbitrarily at runtime; therefore, it does not have a security segment and it does not have access to secure storage or attestation.

Ideally, each security-critical task—e.g., protection of the user's e-banking data and transactions—should be mapped to a single module, using the guidelines provided in Section 3.1.1. All the persistent data linked to the task is thus sealed to this one module and a remote party wishing to inquire on the status of the task only has to ask this module for an attestation report. In modern commodity software systems, however, the code routines and data items involved in a security-critical task may be scattered across different threads, address spaces, privilege levels or virtual machines (VMs). Allowing a trusted software module to span several of these conventional *software entities* would require defining Bastion versions of existing inter-entity transition and communication mechanisms such as Inter-Process Communication (IPC), system calls and inter-VM message passing. In addition to increasing TCB complexity, this would make our architecture dependent on the interfaces and mechanisms used by the operating system, the thread libraries and other implementation-specific features.

To avoid such dependencies, we define a trusted software module as having its code base fully contained in one thread, in one address space, in one privilege level and in one VM. Security-critical tasks that have code spanning multiple software entities must thus be split into several trusted software modules. We define a *trust domain* as the set of trusted software modules collaborating towards the completion of a given security-critical task. Our TCB offers primitives for *secure inter-module collaboration*, i.e., the protected exchange of commands and data between modules in a trust domain. This includes shared memory interfaces protected by the TCB. However, the actual crossing of boundaries between software entities is left to existing software mechanisms, located within untrusted module zero.

The concept of the trust domain is especially useful in scenarios where different software vendors are in charge of writing the modules collaborating towards a given security-critical task, e.g. a trusted application-level library module written by vendor X, requiring the collaboration of a trusted OS device driver module written by vendor Y. In this case, each vendor writes its own module and defines its own security segment. The owner of the security-critical task—either one of the vendors, a user or a third party—then gathers the modules and defines the trust domain. Chapter 8 gives a detailed example of how trust domains can be defined, given specific security policies. The next section gives more details on how to define each module within a trust domain.

3.1.1. Trusted Software Modules

A trusted software module is defined by its *security segment*, a data structure that is part of the data space of the application or OS containing the module. Depicted in Figure 3.2., the security segment has different sections:

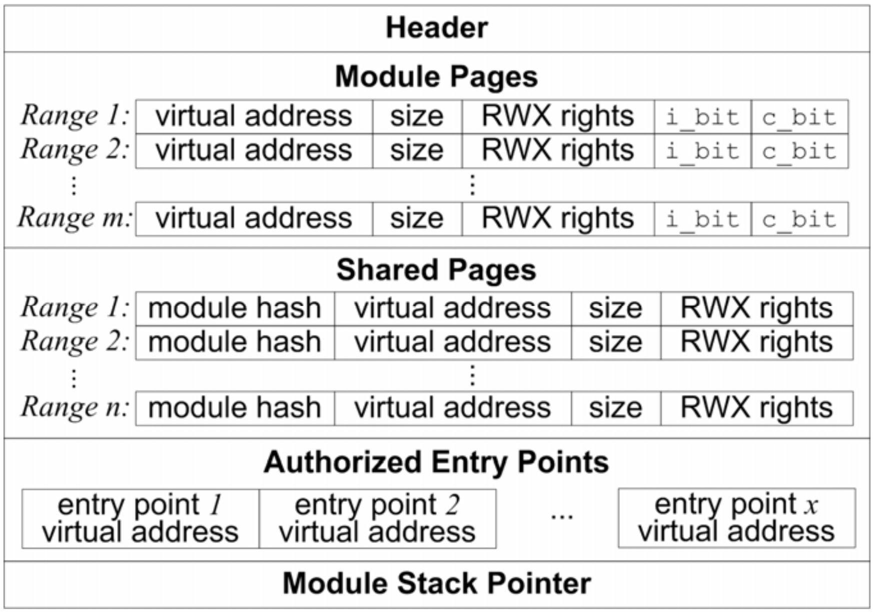

Header
Module Pages
Range 1: virtual address | size | RWX rights | i_bit | c_bit
Range 2: virtual address | size | RWX rights | i_bit | c_bit
⋮
Range m: virtual address | size | RWX rights | i_bit | c_bit
Shared Pages
Range 1: module hash | virtual address | size | RWX rights
Range 2: module hash | virtual address | size | RWX rights
⋮
Range n: module hash | virtual address | size | RWX rights
Authorized Entry Points
entry point 1 virtual address
entry point 2 virtual address
...
entry point x virtual address
Module Stack Pointer

Figure 3.2. The Security Segment

The *Module Pages* section defines the set of virtual pages that are to be accessible to the module. Each entry in this section is a range of virtual addresses (specified as a base virtual address and a size), associated with a set of Read/Write/eXecute (RWX) access rights the module has to these pages, and the `i_bit` and `c_bit` specifying whether pages in that range are to be integrity protected and encrypted by crypto engines in the Bastion hardware. The *Shared Pages* section defines shared memory interfaces between the module and either other modules within its trust domain or module zero. Each entry in this section contains a triple: [module hash, range of pages (base virtual address, size), RWX rights]. The *module hash* (defined below) identifies the module which is allowed to access the shared page, the range specifies the virtual pages to be shared and the RWX rights specify the Read, Write and eXecute rights for this module to these shared pages. Any page in Module Pages but not in Shared Pages is a *private module page*; it is accessible exclusively by the module being defined. The *Authorized Entry Points* section lists the entry points into module code that the TCB must check at runtime. The *Module Stack Pointer* section identifies the top of the module's private stack. Private module pages must be reserved for the module's private heap and stack. Modules requiring a heap must have their own copy of heap allocation functions[8], to allocate memory from the private heap pages.

Given a security-critical task, one must first define a security policy specifying the data items to be protected (and how, with the `i_bit` and `c_bit`), the operations allowed over these data and the authorized interactions with the outside world. Only then can the code and data used in completing the task be identified and partitioned into modules. Once modules are formed, they must be inspected for correctness and the absence of software vulnerabilities. Note that memory pages shared between a trusted software module and the untrusted module zero must be left unprotected (`i_bit` and `c_bit` set to zero) since memory protection mechanisms do not apply to module zero: by definition, these module zero data or code pages can be changed arbitrarily. In implementations of Bastion where encryption adds a significant overhead in memory latencies or power consumption, pages containing data that is not confidential can be left unencrypted (`c_bit` = 0), although integrity verification should be applied to all private pages.

Trusted modules should be made orders of magnitude smaller than the commodity operating systems and applications, making it feasible to obtain formal security proofs for their code base. However, we do support large modules such as a whole application or OS, to provide TCB protection to legacy software that cannot be easily partitioned, e.g., when the source code is unavailable. The construction of security policies and the methodologies for module partitioning and inspection are outside the scope of this thesis. We refer the reader to the literature on security policies (e.g., [Goguen and Meseguer 1982]), partitioning of software into critical and non-critical parts (e.g., [Chen and Lee 2009]) and formal code inspection (e.g., [Kammller 2008]) for a study of these important, but orthogonal research problems.

Precise identification of modules is essential for the programmer wishing to establish a secure shared memory interface with another module. It is also central to the security of the secure storage and attestation services offered by the hypervisor, where a piece of persistent storage data or an attestation report is bound to a module identity. Our TCB's definition of module identity includes three components answering the following questions:

[8] This amounts to only 746 lines of code in our implementation, described in Chapter 7.

1) What are you made out of?

2) How are you protected?

3) Who do you trust?

The first component is the *module hash*, a cryptographic hash taken over the initial state of the module's code and data space. The second component is the *security segment hash*, taken over the security segment, to reflect the protection requested by the module from the TCB. Finally, the *trust domain hash* is a fingerprint over the Trust Domain Descriptor, described next, which identifies the modules collaborating with the module of interest.

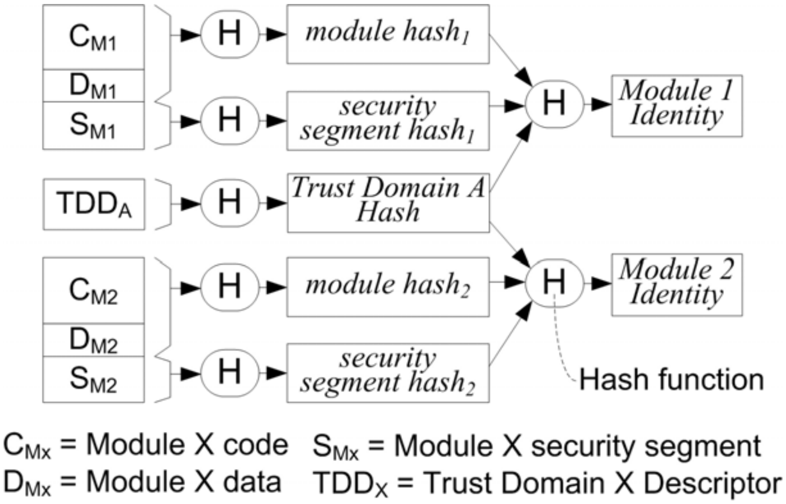


Figure 3.3. Module identity for two modules forming trust domain A

As depicted in Figure 3.3, the identity of a module is a cryptographic hash taken over the concatenation of these three hash values. This means that two copies of a given code and data component are considered to be different modules if they have different shared memory interfaces, authorized entry points or trust domains. A special module hash value (a hash taken over the value zero) is reserved to identify module zero in calls to untrusted code from a trusted software module, and in establishing memory interfaces with untrusted code.

3.1.2. Trust Domains

Every trusted software module is part of a trust domain, which consists of one or more software modules trusted to carry out a given security-critical task. In addition to its security segment, each trusted software module comes with its *trust domain descriptor* which is a list of (module hash, security segment hash) pairs (see Figure 3.4). For a given trust domain, the descriptor contains one such pair for every module in its domain. The trust domain descriptor can thus be used to recover the module identity of every module in the trust domain. Module hashes used to identify sharing modules in the security segment are thus sufficient to precisely identify the sharing module. By specifying the bit identity of modules, our trust domain descriptor makes trust domains into rather static groups of modules. This definition is used to simplify the presentation, but could be enhanced to allow for more flexibility. For example, trust domain descriptors could name modules using a naming scheme where each module name translates into a set of acceptable module hashes. Alternatively, a module could be represented in the trust domain descriptor as the public key of an entity responsible for specifying the acceptable module hashes and security segment hashes for that module. Defining the public key or name server infrastructure required to support these schemes is outside the scope of this thesis.

module hash$_1$	*security segment hash*$_1$
module hash$_2$	*security segment hash*$_2$
⋮	
module hash$_n$	*security segment hash*$_n$

Figure 3.4. A trust domain descriptor for a trust domain composed of n modules

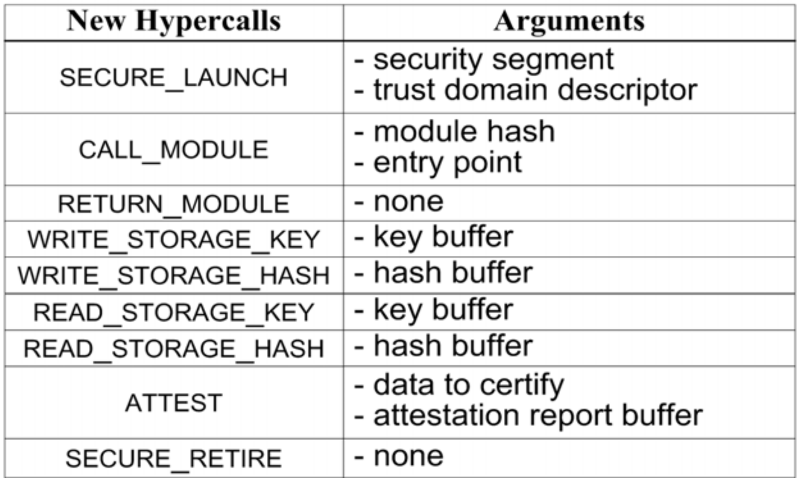
Trusted modules must validate any data they receive from outside their trust domain, i.e., from module zero or from a module in another trust domain. This can be a formal cryptographic validation using keys from the module’s secure storage area, or simply an application-specific sanity check. For example, data from a remote host may transit via an untrusted network, and be handled by an untrusted network stack before reaching a trusted application module. The module and the remote host can then agree on a Message Authentication Code (MAC) scheme, using pre-established keys shared with the remote host and stored in the module’s secure storage area. This allows the module to cryptographically validate the integrity of messages it receives from the untrusted network stack, to detect whether they have been tampered with on the way from the remote host. As another example, a trusted database manager module may receive database queries from untrusted software. In this case, the module should perform sanity checks on the queries to ensure they do not try to access data in a way that violates the module’s security policy.

3.1.3. Trusted Programming Interface (TPI)

The Trusted Programming Interface is the hypervisor interface that must be used by trusted software modules in order to obtain TCB protection and security services. We also describe module transition hypercalls, which allow calls and returns to and from trusted software module code. A list of TPI hypercalls is given in Table 3.1.

Module Launch: A trusted software module uses a new SECURE_LAUNCH hypercall to request that the hypervisor create a TCB-protected execution environment for it when it is first launched. The two arguments to the hypercall are pointers to the security segment and the trust domain descriptor. The hypervisor parses these data structures to determine the set of private and shared module pages. As it does so, the hypervisor computes the three components of module identity and activates hardware-based runtime memory protection mechanisms for the module’s pages (described in Chapter 5). The hypervisor then establishes shared memory interfaces requested in the module's security segment. Finally, the hypervisor checks whether the identity of the module corresponds to a module identity associated with a piece of sealed data. If so, it unlocks the sealed storage, a procedure described in Chapter 6.

Table 3.1. Trusted Programming Interface: New Bastion hypercalls



New Hypercalls	Arguments
SECURE_LAUNCH	- security segment - trust domain descriptor
CALL_MODULE	- module hash - entry point
RETURN_MODULE	- none
WRITE_STORAGE_KEY	- key buffer
WRITE_STORAGE_HASH	- hash buffer
READ_STORAGE_KEY	- key buffer
READ_STORAGE_HASH	- hash buffer
ATTEST	- data to certify - attestation report buffer
SECURE_RETIRE	- none

Following the completion of SECURE_LAUNCH, the TCB enforces the following security properties:

- Module code can only be invoked via the new CALL_MODULE hypercall (described below), through an entry point authorized in the security segment, and with its initial stack pointer set to the value defined in the security segment.

- Module code can only be exited via the new RETURN_MODULE hypercall (described below) or on an interrupt, via hypervisor-mediated preemption.

- Module data can only be accessed by module code, except for data in shared memory pages,

- Data in shared memory pages can only be accessed by the module itself and by the sharing modules identified in the security segment.

- Module state in machine memory (hardware RAM chips) and on external memory buses is automatically encrypted and hashed by the hardware on exiting the microprocessor chip, based on the requested `i_bit` and `c_bit`.

Module Transitions: To invoke a trusted software module's functionality, caller code—either in module zero or in a trusted software module—must use the new CALL_MODULE hypercall. The arguments to this hypercall are a module hash, identifying the callee module, and a virtual address identifying an entry point into callee code. The caller and callee modules must be part of the same virtual address space, at the same privilege level. The hypervisor intervenes to check the legitimacy of the transition, record the calling site and carry out the jump to callee code. When the callee module is done with its computation, it invokes the new RETURN_MODULE hypercall, which does not require any arguments. The hypervisor services this hypercall by checking that it matches a previously validated CALL_MODULE, and returns to caller code, at the instruction following the registered call site.

Secure Storage: A module can seal persistent security-critical data to its identity as follows. First, it must generate a symmetric encryption key K and use it to encrypt the data. Second, it must compute a cryptographic hash H over the resulting ciphertext to fingerprint the data. Then, the module must invoke the WRITE_STORAGE_KEY and WRITE_STORAGE_HASH hypercalls, with the key or hash as the argument, respectively. This seals the data, via the (key, hash) pair, to the current module's identity. The data thus protected can then be written to the untrusted disk using an untrusted file system manager in module zero. Following a reboot of the platform, a securely launched trusted software module can recover this (key, hash) pair by invoking the READ_STORAGE_KEY and READ_STORAGE_HASH hypercalls. The hypervisor will only return a (key, hash) pair to the module if the pair is sealed to a module identity matching the one computed during the requesting module's SECURE_LAUNCH invocation. The key and hash can then be used to decrypt the contents of the module's secure storage area and verify its integrity. As detailed in Chapter 6, the hypervisor uses a new microprocessor register to protect this storage against replay attacks.

Attestation: Trusted software modules can attest to their identity or certify a computation result for a remote party, by invoking the hypervisor's ATTEST hypercall. This takes two pointers as arguments, one pointing to the data to certify (if any), the other pointing to a module memory region where the attestation report should be returned. The hypervisor services this hypercall by creating an attestation report, signed by the microprocessor, which includes the module data to be certified, the identities of all modules within the requesting module's trust domain and the identity of the hypervisor.

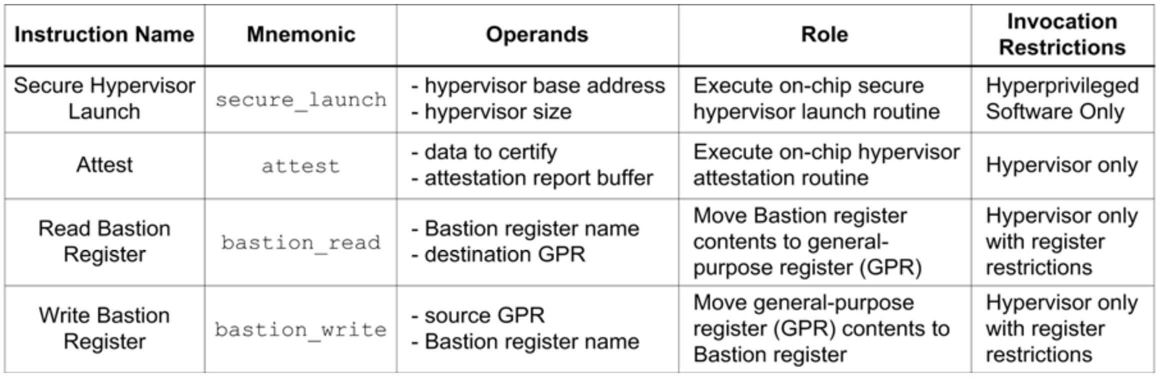
Retirement: A new SECURE_RETIRE hypercall can be invoked by modules wishing to tear down their secure execution compartment and make inaccessible their code and all their protected data resources, in registers, VM memory and hypervisor data structures.

3.2 Hardened Virtualization Layer

The Trusted Programming Interface described above relies on our security-enhanced hypervisor which is directly protected by new Bastion hardware features in the processor. The new Bastion instructions are shown in Table 3.2, and the new Bastion registers in Table 3.3.

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's responsibility is to fetch the hypervisor binary image from persistent storage (e.g., Flash memory or disk), load it into memory and jump to the hypervisor's initialization routines. The Bastion-modified 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 CPU 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 the new Hypervisor Identity register (*hv_identity*). This value is the identity of the loaded hypervisor, used to identify the hypervisor during attestation and to bind the hypervisor to its secure storage area. The `secure_launch` instruction can only be invoked once per power cycle. This ensures that a securely launched hypervisor remains in control of the CPU until the next platform reset.

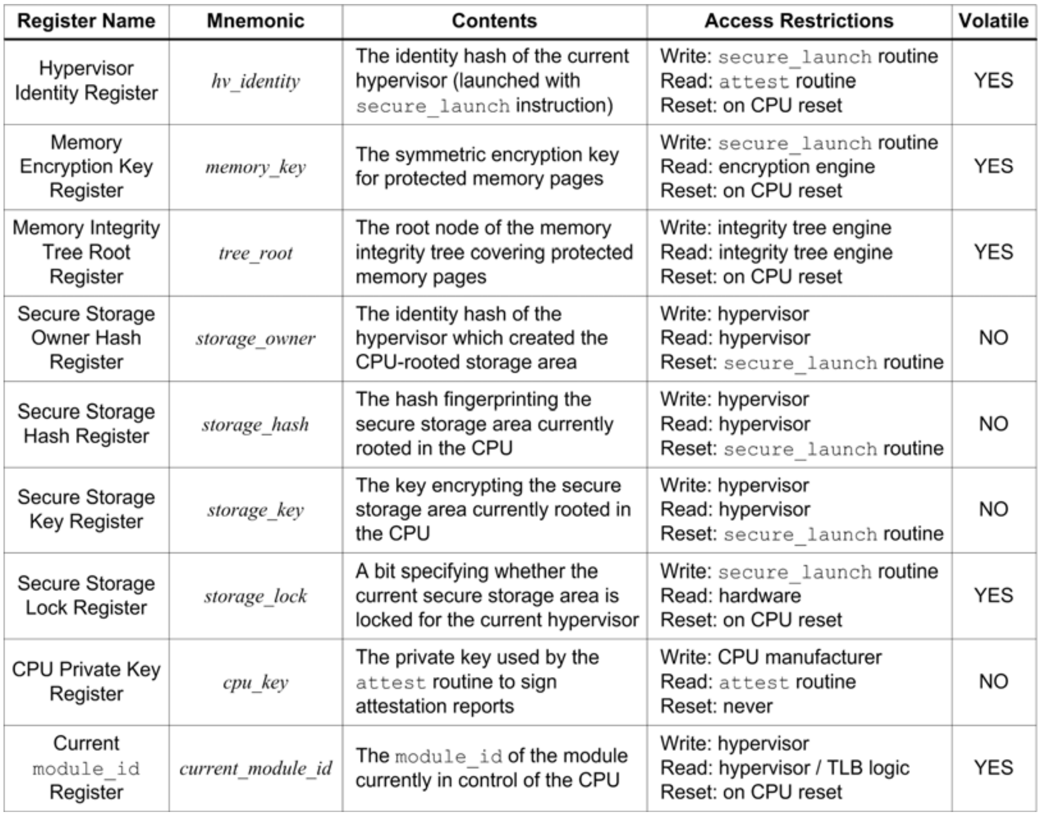
Table 3.2. New Bastion instructions



Instruction Name	Mnemonic	Operands	Role	Invocation Restrictions
Secure Hypervisor Launch	`secure_launch`	- hypervisor base address - hypervisor size	Execute on-chip secure hypervisor launch routine	Hyperprivileged Software Only
Attest	`attest`	- data to certify - attestation report buffer	Execute on-chip hypervisor attestation routine	Hypervisor only
Read Bastion Register	`bastion_read`	- Bastion register name - destination GPR	Move Bastion register contents to general-purpose register (GPR)	Hypervisor only with register restrictions
Write Bastion Register	`bastion_write`	- source GPR - Bastion register name	Move general-purpose register (GPR) contents to Bastion register	Hypervisor only with register restrictions

Runtime hypervisor memory protection is provided by two new cryptographic engines on the microprocessor chip, located between the last level of on-chip cache (denoted L2 here) and the interface to external memory. One of the engines provides memory confidentiality by encrypting all hypervisor cache lines. The other provides memory authentication by maintaining a memory integrity tree over all of the hypervisor’s memory space. Both schemes are initialized by the on-chip hypervisor launch routine invoked by the `secure_launch` instruction. The key used for encryption and the root of the integrity tree—the trusted reference used for integrity verifications—are both stored in new dedicated processor registers (the Memory Encryption Key register, *memory_key* and Memory Integrity Tree Root register*, tree_root*) located within the crypto engines. During runtime, every hypervisor cache line fetched from external memory is decrypted and integrity-checked by the engines before being fed to the processor pipeline. Upon eviction of a dirty hypervisor line from the L2 cache, the line is encrypted before being written back to main memory, and the integrity is updated to reflect the new state of the line. As detailed in Chapter 5, the hypervisor leverages these same memory protection mechanisms to provide encrypted and integrity-checked pages to trusted software modules.

Table 3.3. New Bastion registers



Register Name	Mnemonic	Contents	Access Restrictions	Volatile
Hypervisor Identity Register	*hv_identity*	The identity hash of the current hypervisor (launched with `secure_launch` instruction)	Write: `secure_launch` routine Read: `attest` routine Reset: on CPU reset	YES
Memory Encryption Key Register	*memory_key*	The symmetric encryption key for protected memory pages	Write: `secure_launch` routine Read: encryption engine Reset: on CPU reset	YES
Memory Integrity Tree Root Register	*tree_root*	The root node of the memory integrity tree covering protected memory pages	Write: integrity tree engine Read: integrity tree engine Reset: on CPU reset	YES
Secure Storage Owner Hash Register	*storage_owner*	The identity hash of the hypervisor which created the CPU-rooted storage area	Write: hypervisor Read: hypervisor Reset: `secure_launch` routine	NO
Secure Storage Hash Register	*storage_hash*	The hash fingerprinting the secure storage area currently rooted in the CPU	Write: hypervisor Read: hypervisor Reset: `secure_launch` routine	NO
Secure Storage Key Register	*storage_key*	The key encrypting the secure storage area currently rooted in the CPU	Write: hypervisor Read: hypervisor Reset: `secure_launch` routine	NO
Secure Storage Lock Register	*storage_lock*	A bit specifying whether the current secure storage area is locked for the current hypervisor	Write: `secure_launch` routine Read: hardware Reset: on CPU reset	YES
CPU Private Key Register	*cpu_key*	The private key used by the `attest` routine to sign attestation reports	Write: CPU manufacturer Read: `attest` routine Reset: never	NO
Current `module_id` Register	*current_module_id*	The `module_id` of the module currently in control of the CPU	Write: hypervisor Read: hypervisor / TLB logic Reset: on CPU reset	YES

3.3 Hardware and Hypervisor Extensions

As depicted in Figure 3.5, implementation of the Bastion architecture requires changes in both the processor hardware and in the hypervisor. The processor hardware must be modified to add hardware tags in the L2 cache and Translation Lookaside Buffers (TLBs), mapping modules to their secure execution compartments. A Bastion register file must be created to store the new volatile and non-volatile Bastion registers. This includes the Hypervisor Identity register (*hv_identity*), the registers protecting the hypervisor's secure storage area (the Secure Storage Owner Hash register, *storage_owner*, the Secure Storage Hash register, *storage_hash*, the Secure Storage Key register, *storage_key* and the Secure Storage Lock register, *storage_lock*), the register containing the processor's private key used for attestation (the CPU Private Key register, *cpu_key*) and the register identifying the module currently executing (the Current `module_id` register, *current_module_id*[9]).

Two new instructions must be provided to access these new registers, the Read Bastion register instruction (`bastion_read`) and the Write Bastion register instruction (`bastion_write`). Opcodes must be assigned to implement the new `secure_launch`,

[9] For the sake of clarity, this register is depicted as part of the Bastion register file. In practice, it should be located within the TLB itself, where it is accessed by our extended TLB lookup logic.

`attest`, `bastion_read` and `bastion_write` instructions. On-chip memory must also be dedicated to storage and execution of the secure routines invoked by these two instructions. Two new hardware crypto engines are added between the L2 cache and the memory interface. They host two registers, the *memory_key* and *tree_root* registers, containing respectively the key encrypting Bastion-protected memory and the root of the integrity tree covering Bastion-protected memory.

The hypervisor must be extended with handlers to service our new hypercalls. Hypervisor memory must be allocated to store the new Module State Table and new data structures we call Bastion Maps, described in Chapter 5. The main changes to existing hypervisor logic are in the TLB miss handlers and page table logic, which must be extended to enforce module compartmentalization.

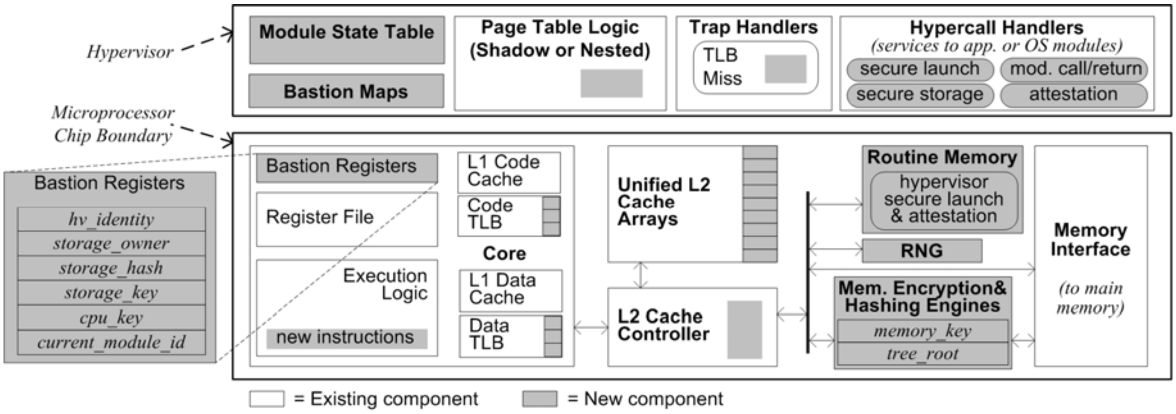


Figure 3.5. New hypervisor and processor components in Bastion

3.4 New Applications

The improved security, scalability and performance of our security primitives, and Bastion's fine-grained protection of trusted software modules, enable new classes of applications. We briefly describe three of these application classes that take advantage of fine-grained trusted computing primitives in an untrusted software stack. Chapter 8 evaluates new Bastion functionality in greater depth by presenting an extensive usage scenario.

3.4.1 Policy-Protected Objects

The TCB's protection of low-overhead software modules that encapsulate arbitrarily small amounts of data and code opens the door to having thousands or millions of protected compartments. Each compartment could contain as little as one data structure and the code routines trusted with reading and writing that data structure. With the hardware enforcing that a data structure cannot be accessed except through its associated code routines, which can enforce arbitrary security polices for this data, we can implement policy-protected objects. This mimics the object-oriented programming paradigm of a *class*, which groups access routines and operators for a given set of data.

The class construct provides a useful abstraction for the programmer, with clean interfaces, thus reducing the chances for programming mistakes that could corrupt the object. However, the class cannot provide runtime protection against intentionally malicious code in the rest of the application or in the OS, and it does not protect against physical attacks. With our TCB, each data item, encapsulated in a module with its methods, is provided with runtime protection against these threats. Moreover, these modules can contain methods that flush dynamically modified state, encrypted and hashed, to a protected persistent storage area sealed to the module's identity. This protects against offline attacks on the module's persistent state, which could be stored locally on the platform's disk, or remotely on network or "cloud" storage systems. Finally, fine-grained provision of attestation services allows the object itself to attest to its identity and the current state of its data items to remote parties interested in a particular aspect of the system (e.g., "What is the state of firewall rules file currently in use?").

3.4.2 Protected Monitor

Dynamic binary translators, application-level sandboxes (e.g., the Java Virtual Machine) and OS-based rootkit detectors all perform some form of monitoring, which is often critical to security[10]. While they may try to detect or prevent security violations, these monitors are themselves exposed to attacks from a compromised OS or physical attacks. Our architecture can protect a monitor *within* a potentially compromised software stack, and allow the monitor to securely report on its status using attestation services. Sealed storage services also allow the monitor to maintain an audit trail that only it can modify, even if the OS has been hijacked or the physical disk has been stolen.

For example, by defining a monitor as a Bastion-protected module storing an audit trail in its secure storage area, an adversary stealing a hard drive containing the monitor's secure storage area will be unable to observe the audit trail or modify it without being detected when it is used. An attacker trying to write to the logs via the monitor will be denied access since a trustworthy monitor should not generate audit trails on request. The attacker is also unable to force the monitor to modify the logs by corrupting the monitor, since its code base is protected by Bastion's memory integrity tree mechanism. An adversary trying to read or write the raw bytes of the secure storage area—either indirectly, through software other than the monitor, or directly, with hardware probes—will only be able to read ciphertext bytes and remain unable to update the secure storage hash of the monitor—i.e., any modification will be detected upon usage of the logs by the monitor. Finally, since a trustworthy monitor is expected to implement proper user authentication, an unauthorized adversary trying to read logs via the monitor software will not be granted access.

Ensuring that an application monitor is not bypassed is a harder problem. We do not try to address it in the general case in this thesis, but we note that a trusted software module within the OS may be able to ensure the monitor gains CPU control upon events

[10] For more information about the general concept of a reference monitor, see [Anderson 1972].

that require monitoring. In any case, the protected monitor must be written so that it can report that it was prevented from completing its task when an adversary targets it in a denial of service attack.

3.4.3 Hardened Extensions

Modern application or operating system software is often dynamically extensible. These extensions may contain security-critical code or manipulate sensitive data—e.g., an e-banking browser plug-in, a DRM media player plug-in or a device driver for secure display hardware (e.g., HDCP [Lyle 2002]). Unfortunately, their security is usually fully dependent on the correctness of all the code located within the address space where they are "plugged in". With its fine-granularity protection, our TCB allows for trusted application plug-ins or security-critical device drivers to be protected from software vulnerabilities in the surrounding code base and from physical attacks (e.g., a user trying to violate DRM restrictions by using a hardware "mod chip" to bypass the platform's secure boot sector).

3.5 Chapter Summary

In this chapter, we presented an overview of the Bastion architecture. We described the software model we are considering and introduced the *Trusted Programming Interface*, a set of basic abstractions programmers of security-critical tasks must use to obtain Bastion protection. We also summarized the Bastion instructions, hypercalls, registers and other new features in our Trusted Computing Base, formed of the microprocessor hardware and hypervisor software. In the next Chapter, we detail the microprocessor mechanisms that enable protection of the software part of the TCB, the Bastion hypervisor.

Chapter 4

Hardened Virtualization Layer[11]

The hypervisor—i.e. the virtualization layer—is an integral part of our Trusted Computing Base. Protection of the hypervisor is thus essential to achieving the security objectives of the Bastion architecture. As a software component, the hypervisor is vulnerable to attacks from other software layers and from hardware adversaries. These attacks can occur *online*, while the hypervisor is running, or *offline*, while the platform is powered off or when the platform runs a software stack without a hypervisor.

Existing microprocessor support for virtualization can protect a virtualization layer against online software attacks, and in some cases enable detection of offline attacks. However, existing approaches do not defend against online hardware attacks. Addressing this problem, the mechanisms presented in this chapter enable a *hardened hypervisor*, protected against all online and offline attacks defined in our threat model. Although the presentation targets the Bastion architecture, the ideas introduced here can provide strong protection to the virtualization layer in any application where this layer is critical to security, reliability and functionality. In the rest of this chapter, we first provide some background on virtualization. We then present our approach to protecting hypervisor launch and runtime state, defending against offline and online attacks.

4.1 Background on Virtualization

A major trend in industry is to use computing platform virtualization for sharing of hardware resources, software compatibility and isolation of workloads. In a virtualized software stack, a hypervisor (a.k.a. Virtual Machine Monitor, VMM) creates and manages concurrently running Virtual Machines (VMs) that each contain a *guest Operating System* (OS) hosting user applications. To share a platform's hardware resources (e.g. CPU, memory, network, storage, display, keyboard) between the VMs, the VMM allocates a subset of the resources to each VM. The VMM *virtualizes* that share of

[11] This chapter is an expansion of prior publications [Champagne and Lee 2010], [Champagne and Lee 2010b], [Lee and Champagne 2010]

resources such that the VM’s guest OS has the illusion of running directly on bare hardware, with full control over all platform resources.

4.1.1 Motivating Virtualization

As with many technological advances, virtualization was developed in large part as a cost reduction technology. Many organizations have vast amounts of computing and storage resources that remain untapped as a result of dedicating one computer to each workload [Carr 2005]. Virtualization enables sharing of a platform’s hardware resources amongst workloads—each potentially requiring a different software stack—and thus allows organizations to exploit their physical computing resources more efficiently. The strict isolation of resources provided by the hypervisor can ensure workloads do not interfere with one another, which can be required to conform to specific business or legal requirements (e.g. separate workloads belonging to two clients who are competitors). Virtualization also reduces costs by extending the compatibility of a given hardware platform. The virtualization layer can run different operating systems and emulate an Instruction Set Architecture (ISA) other than that of the underlying microprocessor. This avoids having to maintain a large farm of heterogeneous computer systems simply to support the software and hardware interface requirements of legacy software.

Virtualization can also improve the reliability of computing systems in the face of hardware failures. Hypervisors can present a constant hardware interface (e.g., fixed memory, disk and I/O resources, constant ISA) across different physical computers [Madnick and Donovan 1973]. This allows for easy migration of a VM and its workload from a failing computer to a functioning computer [Nelson et al. 2005]. Such quick migration of Virtual Machines can also be exploited when there is no failure, as part of a load balancing scheme between computers [Clark et al. 2005]. These applications of virtualization have spawned new business models for low-cost virtual managed servers and on-demand computing, e.g. [VI 2009].

4.1.2 Forms of Virtualization

This thesis is concerned with *platform virtualization*, where an entire software stack, including an operating system runs in a Virtual Machine, itself running on a hypervisor. This differs from *application virtualization*, where applications are executed within a Virtual Machine sandbox running on a Virtual Machine Monitor, which itself runs as an application on a regular operating system. Platform virtualization is concerned with isolation and resource sharing between full software stacks, while application virtualization is mostly concerned with portability and restricting application code behavior, e.g. the Java Virtual Machine [Venners 1999]. We do not discuss application virtualization solutions further since they rely on the operating system (considered untrusted in our threat model) to protect the Virtual Machine Monitor.

We distinguish between two main techniques for platform virtualization: *para-virtualization* and *full virtualization*[12]. In para-virtualization [Armstrong et al. 2005, Saulsbury 2008], guest operating systems must be modified to allow for virtualization to take place while in full virtualization [Intel 2006, AMD 2008a], virtual machines can host unmodified guest operating systems. In para-virtualization, a guest OS is partially rewritten so privileged instructions modifying OS-specific hardware state are re-routed to the hypervisor. Without this re-routing, the hypervisor cannot retain ultimate control over how hardware resources are used, hence it cannot share them amongst the different Virtual Machines. The handling of re-routed privileged instruction can be done either directly in the hypervisor or offloaded to a special VM dedicated to managing platform resources. As an example of para-virtualization, the Xen hypervisor hosts a VM called Domain0 responsible for managing the resources of other domains, including their access to virtual network devices and virtual block devices [Barham et al. 2003].

Full virtualization techniques support concurrently-running unmodified operating systems either with dynamic binary translation or dedicated microprocessor hardware. In the former, a host OS runs a Virtual Machine Monitor application performing dynamic binary translation on its Virtual Machines, e.g. VirtualBox [Watson 2008]. The goals of the translation are 1) to emulate privileged instructions emitted by the VM (which actually executes in unprivileged mode) and 2) to account for discrepancies between the hardware expected by the VM and the actual hardware the VMM is running on. With hardware support for virtualization, the microprocessor itself can re-route VM privileged instruction invocations to the hypervisor. A hardware-supported hypervisor can thus retain ultimate control over hardware resource usage without having to rewrite the operating system and without having to rely on time-consuming emulation of privileged instructions by a dynamic binary translator.

This thesis assumes hardware support for full platform virtualization, which is now available from most mainstream microprocessor vendors, e.g. [Intel 2006, AMD 2008a]. This allows supporting legacy and future operating systems without requiring modification in their code base. Our baseline platform also avoids the overheads of dynamic privileged instruction emulation and does not require trust in the operating system code, e.g. for a management VM or as a host for a dynamic binary translator application. This thesis does not consider recursive virtualization, where virtual machines can host hypervisors hosting virtual machines hosting hypervisors, etc. However, we note that to enable scalability, the current Bastion architecture is already designed as a set of “recursive” security services (the hardware protects the hypervisor, which in turn protects any number of modules). At first glance, this single level recursion appears to be extensible to an arbitrary number of levels. Recursive hypervisors could be treated as simple software modules by our TCB’s hypervisor. They would then leverage this TCB protection to provide security services to the next level of recursive virtualization. A thorough treatment of security in recursive virtualization environments is left to future work.

[12] For the sake of clarity, we limit ourselves to two broad categories. A more detailed taxonomy of techniques for virtualization can be found in the literature [Smith and Nair 2005].

4.1.3 Methods for Memory Virtualization

Unmodified guest operating systems expect to have full control over the physical address space where they execute. This means they must be able to access every physical address from zero to the highest address accessible in that space. However, on a given hardware platform, there is only one zero-based physical address space (corresponding to the actual memory available in the RAM chips), regardless of how many guests are running. One of the core tasks of any hypervisor is to thus virtualize platform memory by providing the guest operating system in each Virtual Machine with the illusion it has access to its own zero-based physical address space. 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. We call these page tables the *guest page tables*. To map guest virtual memory to machine memory, hypervisors use either *shadow page tables* [Adams and Agesen 2006] or *nested paging* [AMD 2008b], the two main techniques for memory virtualization. Mainstream virtualization-enabled microprocessors support both techniques [Intel 2006, AMD 2008a][13]. Nested paging requires dedicated hardware support while shadow page tables can be implemented without modifications to the hardware, using existing hardware Memory Management Unit (MMU) functionality. Nested paging was introduced to overcome the large overheads that are typically introduced by the management of shadow page tables [AMD 2008b].

Shadow Page Tables: With shadow page tables, the hypervisor maintains for each application a shadow page table translating virtual addresses to machine addresses. To enable shadow page table *maintenance*, the hypervisor must intercept all guest OS writes to its guest page tables. This allows the hypervisor to update the appropriate shadow page table upon every modification of a guest page table by an OS. To intercept guest page table writes, hypervisors can mark as read-only the machine pages containing the tables. This way, write accesses to guest page tables trigger a protection fault in the microprocessor, which traps to a hypervisor handler. The hypervisor handler can then determine the fault was caused by a page table write, carry out the write to the targeted guest page table entry and reflect this write in the appropriate shadow page table. These maintenance operations are a major source of overhead in virtualized systems using shadow page tables [AMD 2008b]. Nested paging, described next, was introduced to eliminate this performance overhead and significantly reduce the memory overhead due to storage of one shadow page table per application.

To translate a virtual address used by an application, the microprocessor uses the shadow page table entry translating that virtual address directly to a machine address. The most frequently used shadow page table entries are cached for quick access in the hardware Translation Lookaside Buffers (TLBs), just like page table entries are cached in TLBs on non-virtualized systems. On a TLB miss, the shadow page table for the

[13] The Intel nested paging mechanism is called Extended Page Tables (EPT).

executing program is walked either by a hypervisor handler (on platforms with software-handled TLB misses) or by a hardware walker (on platforms with hardware-handled TLB misses). The goal of the walk is to find the entry that translates the virtual address which caused the TLB miss, and insert this entry in the TLB.

Nested Paging: In nested paging, the hypervisor only keeps track of guest-physical-to-machine memory translations, in data structures called *nested page tables*. This means it maintains a single nested page table per guest rather than one shadow page table per application. Virtual-to-machine address translations are provided to the processor on Translation Lookaside Buffer (TLB) miss events, either by a hardware page table walker or by the hypervisor itself. A nested page table (guest-physical-to-machine translations) is looked up on every guest page table access (virtual-to-guest-physical translations) to construct the missing translation on-the-fly. The extra level of translation increases the TLB miss latency, but it can be reduced significantly by choosing very large page sizes for the nested page tables [AMD 2008b]. Translations in nested page tables typically do not change after being written upon initialization of a guest. During runtime, the hypervisor does not need to intervene on guest page table writes, since nested page tables do not need to keep track of virtual-to-guest-physical mappings performed by the guest OS. To support nested paging, a hardware page table walker must be modified to support the extra level of translation.

4.1.4 Methods for I/O Virtualization

In addition to virtualizing memory, hypervisors must also virtualize Input and Output (I/O) devices so they can be shared amongst the platform’s multiple guest operating systems. On platforms with hardware support for virtualization, the hypervisor is the only software allowed to change the device interrupt vectors, hence it retains ultimate control on how software interacts with hardware devices. To further control guest communication with I/O devices, it can intercept any I/O instruction emitted by a guest and it can restrict guest accesses to memory-mapped I/O by changing access rights to machine memory space (where I/O device control and data registers are mapped by the hardware memory and I/O controller circuits).

A simple way to “share” a device is for the hypervisor to give a guest full control over the device and hide the device from other guests. In this case, the designated guest configures the device directly and the hypervisor routes all interrupts originating from that device to the guest. To share a given device amongst multiple guests, the hypervisor typically hosts its own device driver which time-multiplexes accesses to the I/O device and ensures device-specific sessions or transactions are held consistent when switching contexts between two guests. This device driver is at the receiving end of all interrupts from the device. It routes these interrupts to the interrupt handler of the guest that carried out the I/O transaction causing the interrupt.

4.1.5 Virtualization Layer Security

The main mechanism for protection of the virtualization layer is the hyper-privileged processor mode (sometimes called privilege ring -1) where the hypervisor can execute outside the reach of guest operating systems. Hypervisor software running in this mode can set up memory virtualization mechanisms such that guest operating systems are unable to reference hypervisor memory. Instructions that could allow OS or application software to violate its virtual machine abstraction are reserved to hyper-privileged software. Invoking these instructions in user mode or privileged mode causes a hyper-privileged instruction exception which immediately traps to the hypervisor. Past work has shown that the hypervisor layer itself can be structured to improve its trustworthiness and achieve high levels of assurance during security verification [Karger et al. 1991]. In this chapter, we concentrate on protection the layer itself, regardless of the size and structure of its code base.

To prevent malicious or buggy software from accessing hypervisor memory state through Direct Memory Access (DMA) mechanisms, certain virtualization-enabled hardware platforms also include DMA exclusion vectors [Strongin 2005, Intel 2007]. These vectors, specified in the hardware I/O controller, limit a given guest to a range of machine pages and can only be configured by hyper-privileged software. However, these mechanisms cannot protect from online hardware adversaries (hardware attackers at runtime), who can corrupt hypervisor state or violate VM isolation by gaining direct access to memory chips and to the buses linking these chips to the memory controller.

Recent innovations allow for protection of the virtualization layer during hypervisor launch, to prevent offline attacks on hypervisor state [Strongin 2005, Intel 2007]. The main concern in these architectures is to verify that the loaded hypervisor is a good hypervisor. This is achieved by connecting a Trusted Platform Module (TPM) chip to the processor, where the cryptographic fingerprint of the loaded hypervisor computed during launch can be securely stored, in a tamper-resistant register. The TPM chip can later attest to the identity of the loaded hypervisor to remote parties, who can then determine whether or not the platform runs a good hypervisor. This allows for detection of attacks on the offline state of the hypervisor, i.e. corruption of its binary image in PROM, Flash or on disk. In Intel Trusted eXecution Technology (TXT) [Intel 2007], the fingerprint of the hypervisor is computed on the CPU by a trusted software routine (loaded from RAM and authenticated using public-key cryptography) executing solely on-chip, where it is pinned within the L2 cache array.

Although they can detect hypervisor corruption at launch time, these security mechanisms are unable to prevent malicious modification or observation of the hypervisor's memory state at runtime. They are also vulnerable to attacks on the public key used to verify the trusted software routine. This key sits outside the processor (within the chipset) and must be brought into the processor for verification to take place. A sophisticated hardware adversary could thus generate his own key pair, sign malicious versions of the "trusted" software routine with his own private key and overwrite the good public key (in storage or in transit on buses) with his own public key. As a result, the processor accepts the malicious software routine as genuine and executes it. The malicious routine can then store fake hypervisor fingerprints in the TPM chip to defeat the attestation procedure. While the public key could be stored directly in the processor to avoid this class of attacks, the fingerprinting and attestation performed by Intel TXT-based mechanisms remain vulnerable to attacks on the TPM chip itself [Kauer 2007, Halderman et al. 2008].

The Bastion architecture presented in this thesis addresses the two main shortcomings of existing virtualization layer security solutions:

- the lack of a secure hypervisor launch procedure independent from the TPM chip,
- protection of hypervisor memory state beyond the hyper-privileged processor mode, to enable confidentiality and tamper-evidence in the face of hardware adversaries.

4.2 Secure Hypervisor Launch

New Bastion hardware features in the processor directly protect the launch of our security-enhanced hypervisor without a TPM chip. A new `secure_launch` instruction executes a trusted on-chip routine using the *measured boot* approach to fingerprint the loaded hypervisor. It then sets up its runtime memory protections. Our approach is similar to the late launch mechanism of Intel TXT [Intel 2007], where the measured boot procedure is triggered by an instruction executed at an arbitrary point in time, rather than by the processor reset itself. Prior to describing our secure launch procedure in detail, we compare the two available approaches for protected software launch—measured boot and *secure boot*—and present an argument supporting our architectural choices.

4.2.1 Measured Boot versus Secure Boot

A protected software launch consists in identifying the software being launched and providing it with specific privileges based on its identity. The vast majority of secure computing platforms identify a piece of software by fingerprinting its initial memory space using a cryptographic hash function (e.g. SHA-1 [NIST 1995]). Once the identity is computed, a platform takes one of two approaches in completing the protected software launch. In the secure boot approach (e.g. AEGIS [Arbaugh et al. 1997]), the computed identity (i.e. the hash value) is immediately compared against a list of known good software identities called a *whitelist*. When the software being launched is not on the whitelist, the launch procedure fails, and the software is prevented from executing. Whitelisted software is allowed to execute, with all the privileges associated with its execution mode (hypervisor, privileged or user mode).

In the measured boot approach, the identity of software being launched is not checked against a whitelist, but simply stored in a secure area (e.g. a TPM chip) for future reference. In these systems, the fingerprinting process is often called *measuring* while the resulting identity is a *measurement*. Any piece of measured software can execute, but it can only get access to certain privileged resources if these resources are explicitly bound to its launch-time measurement. This binding is enforced either by a trusted hardware component (e.g. a TPM chip) or a trusted remote party. In BitLocker [Microsoft 2009] for example, the TPM is used to bind a disk encryption key to a known good version of the Windows kernel. The TPM chip only releases the key if the measurement of the kernel launched on a platform reset is identical to the measurement bound to the key. A Linux kernel, or a corrupted or unknown version of the Windows kernel would be allowed to execute on the platform, but it is not allowed access to the encryption key. The assumption underlying such disk encryption systems is that the good version of the OS kernel will properly authenticate users before they are given access to disk data.

In this thesis, we use the measured boot approach in launching our hypervisor and trusted software modules. The main reason for our design choice is to allow the different parties interacting with the platform to use different definitions of trust. Using secure boot would require hardwiring a definition of trust into the TCB—either as an actual whitelist or as the public key of an entity trusted to define a whitelist—and thus prevent parties with conflicting definitions of trust from interacting with the same platform in certain cases. For example, a user's bank could require the use of only the latest, fully-patched version of a web browser in order to access bank accounts via a web interface. This definition of trust—i.e. only browsers with all the latest patches are good—may conflict with that of the user's employer for intranet access. The employer might disallow the use of some of the latest browser patches, e.g. if they add a feature that has not undergone thorough testing yet. In this example, the definitions of trust for the user's bank and employer conflict and prevent the creation, on the user's platform, of a secure boot whitelist satisfying both parties. With a measured boot approach on the other hand, it is up to the remote parties to check the measurement of software running on the user's platform before granting access to privileged resources (accounts and intranet). In the example, the user could then keep two versions of the browser at different patch levels in order to satisfy both parties.

4.2.2 Hypervisor Identification

The measured boot of our hypervisor is carried out by a new on-chip Secure Launch routine, which can only be invoked by our `secure_launch` instruction. The Secure Launch routine is mapped to an architecturally defined region of machine address space, reserved for its code and data space. It is stored and it executes within the boundary of the microprocessor chip. The processor core executes the `secure_launch` instruction as a jump to the base address of the Secure Launch routine. The routine is responsible for identifying the hypervisor being loaded and for storing the identity it computes in a dedicated on-chip register, where it is protected from adversaries. As a result, our measured boot takes place entirely within the CPU and thus proceeds without the need for an external TPM chip, as opposed to Intel and AMD approaches centering their secure launch around the TPM. The `secure_launch` instruction invoking the Secure Launch procedure is executed during the Bastion platform's boot sequence.

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 boot firmware, like for a traditional processor—e.g. the Basic Input/Output System (BIOS) on an IBM PC Compatible or the OpenBoot boot monitor on a SPARC, both stored in a Programmable Read-Only Memory (PROM) chip separate from the processor. This boot firmware code sets up a basic execution environment and then relinquishes CPU control to an untrusted hypervisor loader. The loader's responsibility is to fetch the hypervisor binary image from persistent storage (e.g., disk, PROM or flash storage devices), load it into memory and jump to the hypervisor's initialization routines. The Bastion-modified secure hypervisor must then invoke a new on-chip `secure_launch` instruction for the Secure Launch routine to start executing. The instruction takes two operands: a register containing *hv_pointer* and a register containing *hv_size*, specifying respectively the base address and size in bytes of the hypervisor to be launched. These are used by the Secure Launch routine to determine the location and size of the hypervisor. A more complex hypervisor descriptor might be needed for hypervisors that span multiple non-contiguous regions of memory. For the thin Bastion hypervisor, however, we assume the hypervisor code and data spaces are part of a single contiguous memory region, as is the case for the Sun hypervisor [Saulsbury 2008].

From processor reset up to the point in time where `secure_launch` is invoked, the processor runs without memory translation, i.e. all instructions and data are referenced using machine addresses. This continues throughout the execution of the Secure Launch routine, which is mapped to a reserved segment of the machine address space. The *hv_pointer* operand to the `secure_launch` instruction is thus a machine address pointer. Hence as the Secure Launch routine executes, it can dereference this pointer to load into a processor register the first bytes of the hypervisor's initial memory state. Similarly, it fetches the rest of the hypervisor state by forming pointers to memory up to machine address *hv_pointer+hv_size*. The Secure Launch routine reads all hypervisor state on-chip to compute the hypervisor identity, a cryptographic hash over the memory region defined by *hv_pointer* and *hv_size*. For this identity to truly represent the initial state of the hypervisor, the hypervisor instructions leading up to `secure_launch`, if any, must not modify any of the hypervisor's state.

The cryptographic hash function used by the Secure Launch routine is implemented in software, within the code space of Secure Launch. Although software implementations of such functions can be orders of magnitude slower than equivalent hardware implementations, this design choice does not affect performance in a significant way since Secure Launch is only executed once every platform reset. Many of the operations carried out during platform boot require tens of thousands of cycles, e.g. disk transactions, and thus mask the latency of a slower Secure Launch with respect to user-perceptible delays. Where minimizing boot latency is a crucial goal, the Secure Launch routine should be modified to invoke a hardware hashing engine rather than its own software implementation. To enable this, extra command and data interfaces could be added to the hardware hashing engine Bastion already uses to provide memory integrity. Via these interfaces, a faster Secure Launch routine could request from the hardware engine a hash value computed over the hypervisor's initial memory state.

Once the hypervisor identity is computed, the Secure Launch routine writes it to our new Hypervisor Identity register (*hv_identity*, introduced in Chapter 3). This register can only be written by the Secure Launch routine itself, once every platform reset. To enforce this condition, the register file containing *hv_identity* must be provided with a write enable signal activated only when Secure Launch executes. One way to generate this signal is to use the output of a circuit comparing the current Program Counter (PC) to the range of machine addresses reserved for Secure Launch. Alternatively, it could be sourced from a new status register keeping track of invocations of the `secure_launch` instruction. In all cases, as soon as Secure Launch writes to *hv_identity*, this write enable signal must be deactivated until the next platform reset. The identity written in this register is to be used to identify the hypervisor to remote parties and to bind the hypervisor to its own secure persistent storage area. These topics are covered in Chapter 6, entitled *Secure Input and Output Support*.

4.2.3 Runtime Hypervisor Protection Setup

In addition to identifying the hypervisor, the Secure Launch routine is responsible for initializing the mechanisms providing runtime memory protection to the hypervisor. These will ensure the integrity and confidentiality of the hypervisor memory space, i.e. external entities such as hardware attackers cannot snoop on or corrupt hypervisor memory state on buses and in memory chips during runtime. Our integrity mechanism makes memory state tamper-evident, not tamper-resistant. This means we can detect corruption of protected memory state, but we cannot prevent it and we cannot recover from it. When corruption is detected, the software (the hypervisor or a protected software module) is simply halted; if the halted software is the hypervisor, the platform effectively stops functioning and must be rebooted. This is sufficient to prevent security-critical software from executing corrupted code or from operating on corrupted data, but it cannot ensure the availability of security-critical software under attack. As stated in our threat model, availability is a security function which we do not consider in this thesis.

Described in more detail below, our integrity mechanism detects memory corruption by comparing hypervisor state fetched from external memory against a trusted reference representing the current good version of the hypervisor memory space. The initial value of this trusted reference must be computed over the exact same hypervisor state that was used to compute the hypervisor identity. To guarantee that the initial memory integrity reference corresponds to the measured hypervisor identity, we compute the reference during the Secure Launch procedure. Every cache line of hypervisor state brought on-chip by the Secure Launch routine to compute the hypervisor identity is also hashed by the hardware memory integrity engine to compute the initial memory integrity reference. Execution of the Secure Launch routine does not interfere with reference computation since Secure Launch private data and instructions are fetched from on-chip memory rather than off-chip memory, and therefore they do not go through the memory integrity engine, which only deals with external memory contents.

Our memory confidentiality mechanism, described in more detail below, is also initialized during Secure Launch, as the hypervisor identity and memory integrity reference are computed. In combining memory encryption and hashing (for integrity), we adopt the encrypt-then-hash approach rather than the hash-then-encrypt approach. Not only does theoretical cryptography provide stronger proofs of security with encrypt-then-hash [Bellare and Namprempre 2008], this approach also allows us to verify the integrity of a line fetched from memory without having to decrypt it. We thus encrypt each hypervisor cache line for confidentiality and then hash the encrypted cache line to compute the integrity reference. To do so, the Secure Launch routine activates cache line encryption using an interface to the hardware crypto engines. This interface is simply a set of memory-mapped command and data registers allowing software running on the CPU core to interact with the on-chip crypto engines, as if they were memory-mapped I/O devices. This approach is similar to that used in UltraSPARC for communication between the CPU core and other on-chip units [Laudon 2006].

Cache lines of hypervisor state fetched from external memory go through several steps during Secure Launch:

1) Secure Launch fetches a line from external memory,

2) plaintext line goes to L2 cache (and eventually to L1 and a general-purpose register, where it is manipulated by Secure Launch),

3) a copy of the plaintext line goes through the hardware encryption engine to be encrypted,

4) the encrypted line goes through the hardware integrity engine, where the memory integrity reference is computed,

5) the encrypted line, now fingerprinted by our memory integrity scheme, is written off-chip, where it replaces the plaintext version of the line.

During runtime, hypervisor cache lines read from external memory can thus be decrypted and integrity-verified using the memory protection mechanisms. We note that encryption of the initial hypervisor state is not necessary: this state sits in plaintext in the hypervisor binary on the disk or in PROM so encrypting it does not provide confidentiality. This approach was taken to simplify confidentiality protection for dynamically-generated hypervisor secrets: during runtime, the on-chip encryption engine processes all hypervisor state as it transits between the CPU and main memory. Our approach removes the need for explicitly specifying which hypervisor pages contain secrets, and which ones do not. We describe our memory protection mechanisms in more detail in the next section.

In addition to measuring the hypervisor and setting up its memory protection, the Secure Launch routine must check certain processor-specific settings to ensure the hypervisor will truly remain in full control of the software stack once launched. This includes disabling certain debugging features that could interfere with hypervisor execution, or special processor modes of operation that could circumvent hypervisor authority. For example, Intel's System Management Interface (SMI) [Intel 2009] on x86 processors allows for a fully-privileged system management software stack to execute orthogonally to the software stack controlled by the hypervisor. The SMI software stack can thus spy on hypervisor registers or change critical processor settings. Secure Launch must also ensure that all trap and interrupt vectors initially point to code within the loaded hypervisor, so that the hypervisor does not lose CPU control upon a trap or interrupt. As the hypervisor executes, it can decide to delegate some of the traps and interrupts to guests by changing the related vectors. Finally, the Secure Launch routine must insert entries in the TLBs that will allow hypervisor code and data to be fetched from memory when it starts executing. Any TLB entry defined by code that executed prior to Secure Launch must be flushed. Once all the secure initialization tasks described above are complete, the Secure Launch routine jumps to the hypervisor's architecturally defined entry point, typically the first interrupt vector, pointing to the hypervisor's handler for processor reset events.

4.3 Runtime Hypervisor Memory Protection

Bastion protects the integrity and confidentiality of the memory state of securely launched hypervisors. Every cache line of hypervisor state fetched from untrusted external memory is verified against a trusted reference using the hardware memory integrity engine. This *memory authentication* ensures corrupted hypervisor instructions or data are detected before they are used by the processor core. Hypervisor cache lines are also encrypted when written off-chip and decrypted when brought back on-chip, to ensure secrets handled by the hypervisor during runtime are inaccessible to adversaries with direct access to memory buses and chips. We first provide some background on cryptographic memory protection and then we present our memory integrity and confidentiality mechanisms.

4.3.1 Background on Cryptographic Memory Protection

Memory Authentication: Memory authentication consists in verifying that the data read from memory by the processor at a given address is the data it last wrote at this address. In past work on memory authentication, the main assumption of the trust model is that the processor chip itself is trusted, as is the case in this thesis. A naïve approach to memory authentication would be to compute a cryptographic digest of contents of the entire external memory, store it on the trusted area (i.e., the processor chip) and use that digest to check the integrity of every block read from external memory. Since the digest is inaccessible to the adversary, any malicious modification of the memory can be easily detected. However, this solution requires fetching on-chip all external memory contents on every read operation to check the digest, and on every write operation, to update it; clearly, this generates an unacceptable overhead in memory bandwidth. Another simple solution would be to keep on-chip a cryptographic digest of every memory block (e.g., cache line) written to external memory. Although this strategy greatly reduces memory bandwidth overhead compared to the previous solution, it imposes an unacceptable cost in terms of the amount of on-chip memory required for storing all the digests.

To improve on these two extreme solutions (i.e., to reduce memory bandwidth overhead to a small number of metadata blocks and the on-chip memory cost to a few bytes), researchers [Blum et al. 1991, Gassend et al. 2003, Hall and Jutla 2005, Elbaz et al. 2007] have proposed tree-based structures, based on the Merkle Tree [Merkle 1980], whose leaves are the memory blocks to be protected, and the value of the root is computed by recursively applying an authentication primitive to tree nodes starting from the leaves. Thus, the root node captures the current state of the memory blocks, and it is made tamper-resistant by being stored on-chip. When the processor reads a datum from external memory, a dedicated on-chip memory authentication engine computes the root node value by using values stored in internal nodes lying on the path from the leaf (corresponding to the memory block read) to the root. If the memory block was not tampered, then the computed root value matches the one stored on-chip, else they would differ. On a write, the authentication engine updates the root value to reflect the change in state of memory due to the newly written value.

Memory Confidentiality: Encryption of memory contents has been used in past secure computing solutions to provide memory confidentiality in the face of hardware adversaries [Lie et al. 2000, Suh et al. 2003, Lee et al. 2005, Yan et al. 2006]. In most cases, a symmetric key cipher implemented in a dedicated hardware engine encrypts cache lines written back to external memory and decrypts cache lines fetched on-chip from external memory. The symmetric key used by the engine is generated and stored on the processor chip, and is accessible only to the hardware encryption engine, outside the reach of software. Hardware adversaries probing on external buses and memory chips only have access to the ciphertext of protected software memory so they are unable to infer the plaintext value of memory data from their observations. This provides memory space confidentiality to all software executing on the processor. In certain architectures [Lie et al. 2000, Suh et al. 2003, Lee et al. 2005], only software executing in a special processor mode (or within a specific region of memory) have their memory space encrypted. This way, the contents of their memory space is not only obfuscated from hardware adversaries, but also from software executing outside this special mode (or without access rights for the encrypted memory region). To improve performance or reduce complexity, some secure computing solutions do not employ a symmetric key cipher as a memory encryption primitive. Rather, they use One-Time Pad [Suh et al. 2003b] or authenticated encryption schemes combining memory integrity and confidentiality [Yan et al. 2006].

4.3.2 Bastion Hypervisor Memory Protection

Bastion Integrity Tree: To protect the integrity of external memory against physical attackers, Bastion enhances the Merkle tree technique [Merkle 1980]. This technique was initially introduced to protect the integrity of large public revocation lists [Merkle 1980], then suggested as a method for protecting the integrity of computer memories [Blum et al. 1991] and, recently, was adapted to cache memory systems to provide external memory authentication in modern microprocessor systems [Gassend et al. 2003]. The idea behind the Merkle tree is to split the memory space to be protected into N equal size blocks which are the leaf nodes of a balanced A-ary integrity tree. The remaining tree levels are created by recursively applying a cryptographic hash function, e.g. SHA-1 [NIST 1995], over A-sized groups of memory blocks, until the procedure yields a single hash called the root of the tree. Making the root tamper-resistant—e.g. by storing it on-chip—ensures tampering with the memory space can be detected. To verify the integrity of a memory block fetched from memory, the path starting from the block and ending at the root is recomputed: if the recomputed root is equal to the tamper-resistant root, the check passes; otherwise the block is considered as corrupted. Every time a memory block is modified, the hash nodes on its path to the root are recomputed and the tamper-resistant root is updated. Figure 4.1 shows a 2-ary (binary) tree protecting a memory of 4 data blocks by storing the root on-chip.

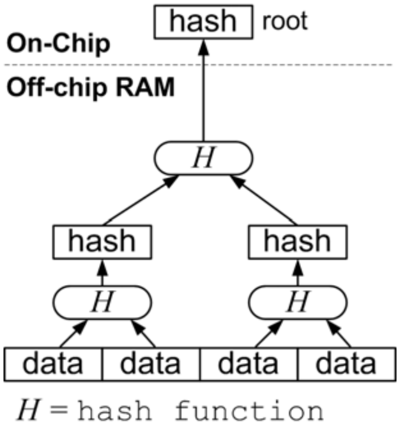


Figure 4.1. A binary Merkle hash tree

Bastion uses a single memory integrity tree to protect all security-critical software, from the hypervisor to the sensitive software modules in the Virtual Machines. We use a single tree to protect the memory of multiple software entities since these entities ultimately share a common machine memory space. Due to virtual memory management done by guest operating systems, some of the protected module pages may be swapped to an on-disk page file during runtime. These pages must also be covered by the integrity tree to ensure trusted module pages are not corrupted while they sit on the disk. Traditional memory integrity trees like the Merkle tree cover an entire physical memory space without exception. They are not compatible with our need for selective coverage of hypervisor and trusted module pages only. They also cannot fingerprint data lying outside the physical memory region they cover, i.e. pages swapped to disk. We thus modify the existing Merkle tree scheme to provide memory integrity in Bastion (see Figure 4.2).

To enable protection at a page granularity, we assign a new `i_bit` to each machine memory page. This bit tells the on-chip integrity checking engine whether a page is to be covered by the integrity tree. When the bit is ‘1’, accesses to the page trigger a typical Merkle Tree hash verification on the data read or written off-chip. When the bit is ‘0’, accesses to the page proceed without any integrity verification. The `i_bit`, as well as the `c_bit` explained below, are made available to the processor hardware via the TLB unit, which is looked up by the processor core upon every memory access. We require, as is the case in the UltraSPARC architecture [Laudon 2006], that even memory accesses that do not require translation (e.g., during hypervisor execution, where machine memory addresses are used directly) need to go through a TLB lookup. The lookup of this "identity" translation provides the processor core with the Read-Write-eXecute access rights for the page and in Bastion, with the `i_bit` and `c_bit`. As depicted in Figure 3.5 in Chapter 3, extensions to L2 cache lines also store a copy of these bits so they are available to the on-chip cryptographic engines upon filling or evicting a line. Upon allocating a new L2 line, these extensions are written with the `i_bit` and `c_bit` of the page containing the new line, copied from the page's TLB entry. Following Secure Launch, the hypervisor is the only entity allowed to insert entries into the TLB. It is thus the hypervisor's responsibility to set the `i_bit` and `c_bit` in the TLB, as an extension of existing TLB entries.

To allow for selective coverage of the machine memory space, our integrity tree, shown in Figure 4.2, 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 memory integrity engine and computed over the page roots, to protect their integrity. Upon platform reset, there are no pages to protect so the engine initializes the top cone on a set of null page roots. When a page is filled with memory contents, its page tree is computed and its page root replaces a null page root. Chapter 5 discusses the Bastion integrity tree in more detail and presents our solution to covering disk pages. For now, we simply point out that pages swapped to disk can be protected by storing their page roots in an integrity-checked hypervisor memory page, as depicted in Figure 4.2, for page roots labeled 5 and 6. 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 can be addressed using techniques such as those introduced in [Champagne et al. 2008], summarized in Appendix A.

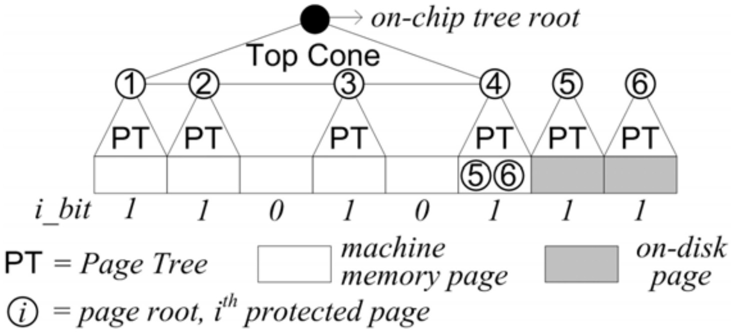


Figure 4.2. The Bastion integrity tree

As hypervisor state is read on-chip by the Secure Launch routine, the memory integrity engine computes the page trees covering the hypervisor's initial memory space. As each page tree is computed, its page root is added to the coverage of the top cone, replacing one of the null page roots. Therefore, the root of the integrity tree obtained at the end of the Secure Launch routine is the initial trusted memory integrity reference.

This tree and its root can thus be used as the hypervisor starts executing, to verify the integrity of each hypervisor cache line fetched on-chip.

Bastion Memory Encryption: Memory confidentiality for hypervisor and protected module state is also provided at a page granularity by a dedicated hardware encryption engine located between the L2 cache and the memory controller. The `c_bit` of each machine memory page is propagated from TLB entries to L2 cache line tags just like for the `i_bit`. The Bastion TCB enforces that all pages with a set `c_bit` also have their `i_bit` set. This prevents attacks on reused memory pages, described in Section 8.1 of Chapter 8. In these attacks, a malicious software module requests Bastion protection and attempts to insert into its memory space an encrypted page from another trusted software module. Requiring integrity verification of encrypted pages ensures that these insertions are detected, and the malicious module cannot get the page decrypted by the Bastion CPU.

Any cache line part of a confidential page is encrypted by the on-chip memory encryption engine when written off-chip and decrypted when fetched from external memory. The encryption engine uses a symmetric key block cipher in Electronic Code Book (ECB) mode on each cache line. These cryptographic operations are performed using the on-chip symmetric key stored in the dedicated Memory Encryption Key register (*memory_key*, introduced in Chapter 3), within the hardware encryption engine. Using the on-chip Random Number Generator (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 hypervisor can thus protect the confidentiality of secrets it generates during runtime.

4.3.3 Limitations

The goal of the Bastion memory protection mechanisms is to show how a single integrity tree and a single memory encryption key can provide memory security to a hypervisor and an arbitrary number of trusted software modules, despite multiple untrusted operating systems. We do not claim the simple mechanisms we present offer better performance characteristics over mechanisms that have been proposed in the literature [Suh et al. 2003, Gassend et al. 2003, Suh 2005, Yan et al. 2006, Rogers et al. 2006, Rogers et al. 2007, Elbaz et al. 2006, Elbaz et al. 2007, Champagne et al. 2008, Rogers et al. 2008], including two advanced mechanisms proposed by the author of this thesis [Elbaz et al. 2007, Champagne et al. 2008]. Rather, we chose the simple Merkle Tree to illustrate clearly how an integrity tree scheme can be adapted to selectively protect, with a single hardware-rooted tree structure, multiple software entities, some with pages swapped to disk. Similarly, we chose a block cipher operating in the simple ECB mode to show that a single encryption could provide memory confidentiality to multiple, sometimes mutually distrustful, software entities. The Bastion architecture was designed to allow for substituting these simple memory protection mechanisms for more sophisticated mechanisms. These more advanced approaches, discussed in detail in Appendix A, can overcome some of the performance and security limitations of our simple approach.

Security Limitations: For each block of plaintext P to encrypt with key K, the Electronic Code Book (ECB) block cipher mode outputs a ciphertext block C = $E_K(P)$, where E is a stateless encryption function. Just like in a paper code book, each ciphertext value corresponds to one and only one plaintext value. As a result, a block of data set to a given plaintext value will always be encrypted to the same ciphertext value, regardless of where it is located, when it was encrypted and what data surrounds it. This makes for a simple implementation, but raises some security issues. First of all, it allows an attacker to determine whether the plaintext data at two different memory locations are identical just by observing whether their ciphertext data are identical. This can provide the attacker some information as to what a given piece of sensitive software is currently doing, and what data values it is manipulating. In addition, two writes of the same ciphertext value to address A at time T_1 and T_2, where $T_1 \neq T_2$ inform the attacker that at time T_2, the plaintext data at address A is the same as it was at time T_1. Adding a nonce (NUmber used ONCE) to the encryption pre-image can solve these problems. However, extending plaintext blocks and storing and retrieving nonce values significantly increase the complexity of a memory encryption scheme. For the sake of clarity, we omit these enhancements from the scheme proposed in this chapter and refer the reader to Appendix A for our advanced memory encryption scheme.

Performance Limitations: Integrity trees can generate a high overhead in terms of execution time due to the $\log_A(M)$ checks required on each load from the external memory, where A is the arity of the tree and M is the number of memory blocks to protect. Gassend et al. [Gassend et al. 2003] show that the performance slowdown can reach a factor of 10 with a Merkle Tree. To decrease this overhead, they propose to cache tree nodes. When a hash is requested in the tree authentication procedure, it is brought on-chip with its siblings in the tree that belong to the same cache block. This way, those siblings usually required for the next check in the authentication procedure are already loaded. However, the main improvement comes from the fact that once checked and stored in the on-chip cache, a tree node is trusted and can be considered as a local tree root. As a result, the tree authentication and update procedures are terminated as soon as a cached hash (or the root) is encountered. With this cached tree solution, Gassend et al. decreased the performance overhead of a Merkle Tree to less than 25%. By changing the hash function—from SHA-1 [NIST 1995] to the GCM [McGrew and Viega 2004]—some researchers even claim to reduce the average performance overhead to 5% [Yan et al. 2006] or even 2% [Rogers et al. 2007]. Although this chapter does not discuss these enhancements to clarify the presentation, they are compatible with the Bastion architecture. Caching of tree nodes can be implemented by having the memory integrity engine request nodes from L2 rather than from external memory. Our simple hashing engine can also be substituted by one with better performance characteristics. The design space for memory integrity tree schemes is further explored in Appendix A.

4.4 Chapter Summary

In this chapter, we introduced the set of Bastion mechanisms providing direct launch and execution protection for a virtualization layer—in this case, the Bastion hypervisor. We first provided background information on virtualization and virtualization security, before describing our approach to secure hypervisor launch. We then described our mechanism for the runtime protection of hypervisor memory, which ensures secure hypervisor execution. With the hypervisor securely launched, and executing in a protected memory space, the Bastion TCB is now fully deployed. In the next chapter, we described how we leverage this TCB to provide security-critical application and OS modules with protected execution compartments.

Chapter 5

Secure Execution Compartments[14]

The launch of an identified hypervisor within a protected memory space completes the establishment of the Bastion Trusted Computing Base (TCB). This hardware-software TCB—formed of hypervisor software and the trusted Bastion microprocessor—can then be used to create secure execution compartments for security-critical OS and application modules running in the virtual machines. In this chapter, we present our architectural mechanisms for creating and maintaining these secure execution compartments. We describe the detailed architecture of hypervisor and CPU structures and procedures for efficient support of these TCB mechanisms.

In Bastion, the software stack in each virtual machine is constructed as part of a regular bootstrap sequence initiated by the hypervisor, whose existing guest management logic creates and provisions VMs. A software module can request a secure execution compartment for itself at any point during a VM's bootstrap sequence by invoking the new SECURE_LAUNCH hypercall. Modules executing within a Bastion compartment are protected from snooping and corruption by surrounding software or hardware adversaries. The Bastion TCB also protects transitions into and out of compartments, when a module is invoked by other software, or when it is preempted by operating system traps and interrupts. Finally, a new SECURE_RETIRE hypercall allows for de-allocation of Bastion resources in a secure module retirement procedure.

Hypervisors typically create, immediately after initialization, their virtual machines based on a configuration file fetched from disk or PROM. For each virtual machine, this file tells the hypervisor how much guest physical memory to allocate and which hardware devices to associate exclusively with a given VM, or virtualize—i.e., share between VMs. With or without a configuration file, the hypervisor ultimately determines which resources are exported to which VM, as well as which resources the hypervisor reserves for itself. In addition to these "boot time" VMs, the hypervisors may allow for dynamic VM creation during runtime, requested through a hypercall interface. These requests may be restricted to a guest OS hosted in a VM with a special "platform management" role, as the Domain0 in Xen [Barham et al. 2003].

[14] This chapter is an expansion of prior publications [Champagne et al. 2009], [Champagne and Lee 2010], [Champagne and Lee 2010b], [Lee and Champagne 2010]

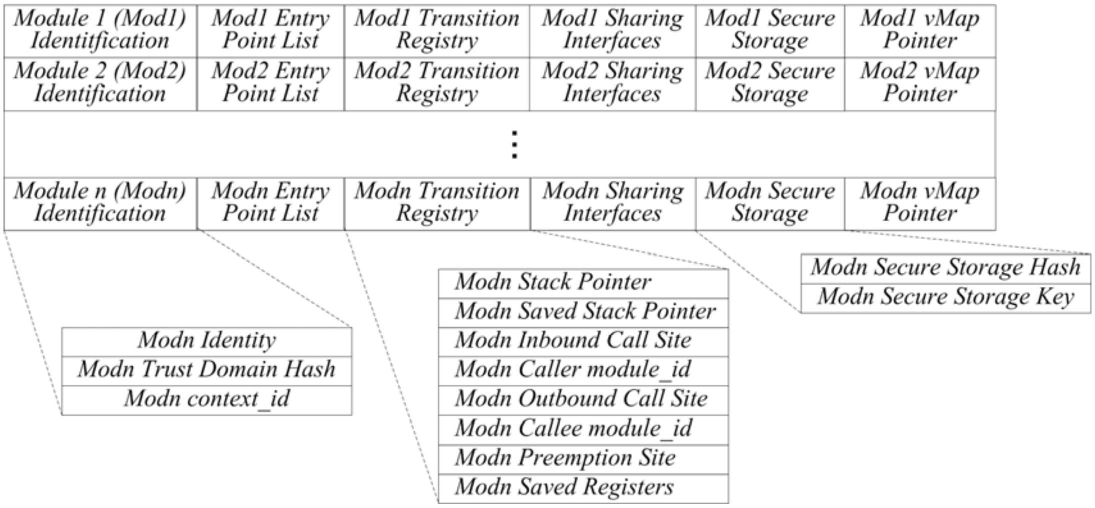
The hypervisor provisions a new VM from available system resources and creates this new VM by defining its configuration and initial state in hypervisor data structures—e.g., shadow or nested page table, device interrupt queue, and virtual network configuration. Once a VM is created, the hypervisor copies or maps boot code (e.g. OpenBoot or BIOS code) into the VM's memory space and then jumps to that boot code. The boot code initializes the virtual machine and eventually loads an operating system—either a minimal runtime environment for application code, a specialized operating system or a full-blown commodity OS—which in turn loads any number of applications. At any point during runtime, a security-critical OS or application module may invoke the new SECURE_LAUNCH hypercall to request a secure execution compartment from the hypervisor.

5.1 Secure Launch

Security-critical modules invoke the SECURE_LAUNCH hypercall to describe their memory space to the hypervisor, request encapsulation in a Bastion compartment and define their trust domains. The memory space of a module and the protections it requests are described using the new security segment associated with each module. As part of SECURE_LAUNCH, the hypervisor computes the module's identity and updates new Bastion data structures to keep track of the module's state. The hypervisor allocates the module its own entry in the new *Module State Table* and maps the module's virtual pages to its machine pages in the new *Module vMap* (forward mapping) and *Machine mMap* (inverse mapping) data structures. A *Module Swap dMap* also keeps track of module pages that have been swapped to disk by guest operating systems. Finally, a *Trust Domain tdMap* keeps track of the different trust domains that have been requested by modules launched with SECURE_LAUNCH since the last platform reset. To associate a module with its resources within these data structures, the hypervisor assigns each module with a shorthand identifier called the `module_id`. We first define the Module State Table and other hypervisor data structures before we describe the servicing of the SECURE_LAUNCH hypercall in more detail.

5.1.1 Hypervisor Data Structures

Module State Table: The Module State Table is a centralized repository for all information defining modules that have been processed by the SECURE_LAUNCH hypercall. Some of its contents are constant from module launch to retirement—e.g., the module identity—while some contents represents volatile module state that changes as the module executes—e.g. the saved state of processor registers for an interrupted module, in the Transition Registry. Depicted in Figure 5.1, the Module State Table contains one entry for each module that has been securely launched since the last platform reset. The first three fields of this table contain information identifying the module of interest: the module's identity and trust domain hash (both computed during SECURE_LAUNCH), and the module's *context_id*, described next.



Module 1 (Mod1) Identitfication	*Mod1 Entry Point List*	*Mod1 Transition Registry*	*Mod1 Sharing Interfaces*	*Mod1 Secure Storage*	*Mod1 vMap Pointer*
Module 2 (Mod2) Identification	*Mod2 Entry Point List*	*Mod2 Transition Registry*	*Mod2 Sharing Interfaces*	*Mod2 Secure Storage*	*Mod2 vMap Pointer*
⋮					
Module n (Modn) Identification	*Modn Entry Point List*	*Modn Transition Registry*	*Modn Sharing Interfaces*	*Modn Secure Storage*	*Modn vMap Pointer*

Figure 5.1. The Module State Table

The *context_id* field of each entry is a platform-specific piece of data identifying the virtual address space where the module executes. For processor architectures where operating systems must explicitly specify an address space identifier in a hardware register—e.g. in the Sun UltraSPARC contextID register [Laudon 2006]—the *context_id* field contains the register value associated with the module's virtual address space. For processor architectures without explicit address space identifiers, the value in the register defining the base address of the guest page table, e.g., the CR3 register in Intel x86 [Intel 2009], can be used to uniquely identify the virtual address space where a module executes. Although we are not aware of processor architectures with hardware virtualization support where neither of these types of address space identifiers is available, Bastion could still be implemented in these cases, with a few additions to the architectural definition. To do so, modules can reserve a page within their virtual address spaces during compilation[15] and specify its address in a new security segment field. During SECURE_LAUNCH, the hypervisor writes a *context_id* to this reserved page and makes the page read-only. During runtime, the hypervisor can identify the current virtual address space by reading the *context_id* from the reserved page[16]. A full-blown definition of this alternative architecture is left for future work.

The Entry Point List contains one or more authorized entry points into the module's code. The hypervisor looks it up on module invocation via the CALL_MODULE hypercall, to determine whether the requested entry point is legitimate.

[15] A specific virtual page can be reserved by defining a dummy page-sized datum at a specific virtual address in a *map file* or *linker script* passed to the program linker during compilation.

[16] Processor architectures such as the Sun UltraSPARC Hyperprivileged Edition [Laudon 2006] allow hypervisors to carry out virtually addressed memory accesses as if they were the currently running operating system or user application. When such memory accesses are not available to the hypervisor, it should either 1) force the virtual-to-machine translation into the TLB, read in the machine address and perform a machine memory access or 2) jump to a dedicated piece of code in application or OS space that reads the *context_id* from the reserved virtual page and provides it to the hypervisor via a hypercall.

The Transition Registry stores constant and volatile state related to transitions into and out of the module. It contains one field for storing the module's private stack pointer (*Stack Pointer*), and one field storing the stack pointer which must be restored when the module is exited (*Saved Stack Pointer*). Upon calling another module, the caller's call site is registered in the *Outbound Call Site* of the caller's Transition Registry, and the callee's `module_id` in the *Callee module_id* field. The hypervisor also saves the caller's call site and `module_id` into the *Inbound Call Site* and *Caller module_id* fields of the callee's Transition Registry. This double registration is necessary to allow for calls to and from module zero, which does not have its own Module State Table entry. When a module is preempted by OS intervention, the hypervisor saves the module's register state in the *Saved Registers* field, and the address of the module's interrupted instruction in the *Preemption Site* field. Once the registers contents are saved in the Module State Table, the hypervisor zeroes the corresponding hardware registers in the processor to avoid preempting software from reading their values.

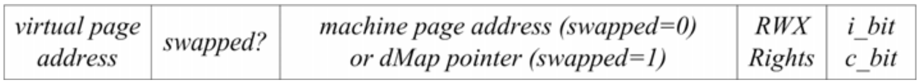
To define authorized memory sharing interfaces, Module State Table entries contain a list of module pages that are accessible to other modules. As in the security segment, each item in this list describes a shared page with a triple formed of a module hash identifying a module to share the page with, the virtual address of the page and the RWX access rights of that module to the page. When SECURE_LAUNCH for module Y determines that Y requests a shared page with module X, it looks up X's list of authorized sharing interfaces in the Module State Table to see if Y's request is compatible with X's authorizations. These sharing interface lists are only looked up during module launch, and they usually contain only a few pages. This means they can be implemented using a simple linked list rather than a more efficient, but more complex, data structure.

Finally, the Module State Table entries store the key and hash used to protect the module's secure storage area (see Chapter 6) and a pointer to the module's vMap. This is a map of all pages the module is allowed to access, as defined by the security segment. In this map, the address of each virtual page is associated with the address of the corresponding machine page and the access rights granted to the module for that page. It also contains the `i_bit` and `c_bit` used to selectively activate encryption and integrity verification for the page. We call this map the *Module vMap* since it contains one entry per virtual page belonging to the module.

Module vMap: The hypervisor keeps one *vMap* per module, specifying the virtual pages belonging to the module, the machine pages to which they map, the module's access rights and the related `i_bit` and `c_bit`. The Bastion hypervisor enforces that no module should request an encrypted page that is not also integrity-protected, in order to defeat the attack on page reuse described in Section 8.1 of Chapter 8. A vMap entry is depicted in Figure 5.2. The hypervisor uses the vMap to detect illegal memory access to module memory by outside software and illegal memory access to outside memory by the module itself. The hypervisor also uses the information in the vMap to detect malicious remapping of module virtual pages by a guest OS during runtime. Module virtual pages that have been swapped to disk are marked as such in the vMap (using the "*swapped?*"

Boolean flag). In systems using shadow page tables, vMap entries are scattered in the shadow page table for the application or OS hosting the module.

In systems using nested page tables (where only guest-physical-to-machine translations are kept by the hypervisor), this map of virtual-to-machine translations can be a simple list, a single-level page table, or a multi-level page table, depending on how many pages belong to the module. Small modules formed of only a few pages can use a compact list for a vMap, which can be traversed quickly whenever the hypervisor needs to know whether a given virtual page belongs to the module. Larger modules should use a full-blown page table, either single or multi-level, to organize their page translations into a data structure that can be looked up efficiently. To speed up the lookup of sparsely populated multi-level page tables and reduce their storage requirements, the guarded page table technique [Liedtke 1995] should be implemented for modules with several pages scattered across a large virtual address space, e.g., 64 bits. Guarded page tables enable efficient traversal and memory allocation for sparsely populated multi-level data structures by skipping levels of indirection where possible. Regardless of the data structure type with which it is implemented, the Module vMap should be aligned to allow for encoding of the data structure type in the least significant bits of the vMap pointer in Module State Table entries.



virtual page address	*swapped?*	*machine page address (swapped=0) or dMap pointer (swapped=1)*	*RWX Rights*	*i_bit* *c_bit*

Figure 5.2. A vMap Entry

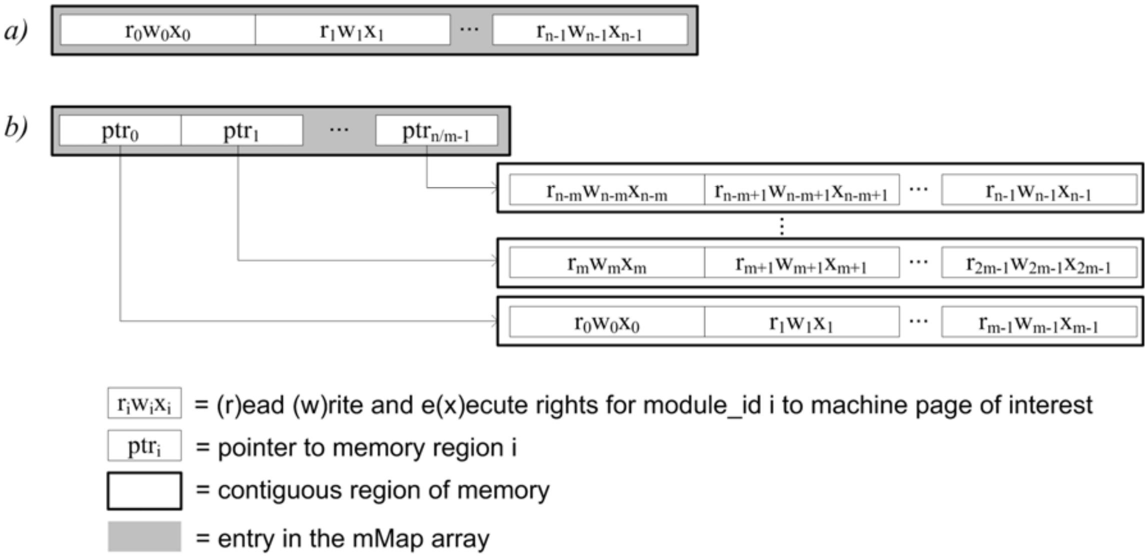

a)
$r_0w_0x_0$
$r_1w_1x_1$
…
$r_{n-1}w_{n-1}x_{n-1}$
b)
ptr_0
ptr_1
…
$ptr_{n/m-1}$
$r_{n-m}w_{n-m}x_{n-m}$
$r_{n-m+1}w_{n-m+1}x_{n-m+1}$
…
$r_{n-1}w_{n-1}x_{n-1}$
⋮
$r_mw_mx_m$
$r_{m+1}w_{m+1}x_{m+1}$
…
$r_{2m-1}w_{2m-1}x_{2m-1}$
$r_0w_0x_0$
$r_1w_1x_1$
…
$r_{m-1}w_{m-1}x_{m-1}$
$r_iw_ix_i$ = (r)ead (w)rite and e(x)ecute rights for module_id i to machine page of interest
ptr_i = pointer to memory region i
= contiguous region of memory
= entry in the mMap array

Figure 5.3. An mMap entry for a `module_id` space of n modules. (a) single-level entry for a small n, (b) 2-level entry for a larger n (memory regions with all `module_id`'s unused or all rights set to zero can remain unallocated, with corresponding pointer set to null)

mMap entry	*integrity tree page root*

Figure 5.4. A dMap Entry

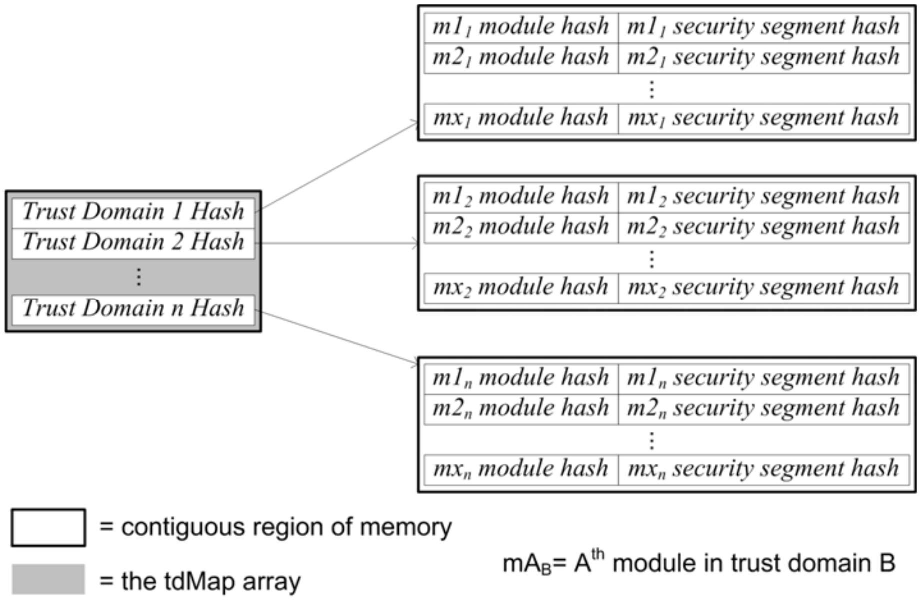

$m1_1$ module hash
$m1_1$ security segment hash
$m2_1$ module hash
$m2_1$ security segment hash
mx_1 module hash
mx_1 security segment hash
Trust Domain 1 Hash
Trust Domain 2 Hash
Trust Domain n Hash
$m1_2$ module hash
$m1_2$ security segment hash
$m2_2$ module hash
$m2_2$ security segment hash
mx_2 module hash
mx_2 security segment hash
$m1_n$ module hash
$m1_n$ security segment hash
$m2_n$ module hash
$m2_n$ security segment hash
mx_n module hash
mx_n security segment hash
= contiguous region of memory
= the tdMap array
mA_B= A^{th} module in trust domain B

Figure 5.5. A tdMap for n active trust domains (domain i is formed of x_i modules)

Machine mMap: The hypervisor uses a single *mMap* to cover all of machine memory. For each machine page, this data structure lists the modules that are allowed access to the page. It is used by the hypervisor upon SECURE_LAUNCH to detect whether pages allocated to a module overlap with other modules. It is also used on TLB misses to confirm that the machine page about to be mapped in the TLB belongs to the module currently executing. The machine memory space is small enough for each mMap entry to be addressable using a single level of indirection, as opposed to the multiple level of indirections required in multilevel page tables for large (i.e., 64-bit or wider) virtual address spaces. Although the page-to-module bindings defined in the mMap are also found in module vMap structures, the mMap is necessary to ensure that these bindings can be looked up quickly in a single data structure, without having to traverse every single vMap. For example, to confirm, using vMaps, that machine page P is currently allocated exclusively to module X, the hypervisor would have to lookup every entry of every running module's vMap to check whether any module virtual page is mapped to P. In comparison, a single lookup to the mMap entry for P tells the hypervisor whether X is the only module with access rights to P.

Figure 5.3 depicts an mMap entry for a `module_id` space of n modules. For each possible `module_id` on the system, every mMap entry stores three RWX access right bits reflecting the module's rights to the machine page. To accommodate a large `module_id` namespace without having to reserve memory for every module, the mMap entries must be implemented as multilevel structures similar to guarded page tables. This avoids allocating memory for `module_id`'s that have not been assigned yet.

Disk Swap dMap: The hypervisor uses a single *dMap* to cover all protected module pages that have been swapped to disk by guest operating systems. When the hypervisor detects a module page was relocated from machine memory to disk, it deallocates the page's mMap entry and allocates the page a dMap entry. Each dMap entry contains all the information that was in the mMap when the page was in machine memory. It also contains the root of an integrity sub-tree covering the page. Since the dMap is stored in protected hypervisor memory, storing the root of a tree covering the page in the dMap effectively protects the page's integrity while it sits on disk (the usage of the dMap is detailed in Section 5.3). The dMap can be expanded during runtime to adapt to an increase in the number of swapped module pages. The only constraint on the size of the dMap is that it must fit within machine memory at all times. This ensures the page tree roots remain protected by the hardware memory integrity mechanism covering hypervisor memory state. The dMap stores mapping and integrity information related to on-disk module pages, but it does not keep track of the position of module pages within a guest OS swap file (this information is typically unavailable to the hypervisor anyway). The hypervisor only needs to know that a protected module page is on disk, it does not need to know about its exact location in the swap file. As for the vMap, the internal structure of the dMap can be adapted to the number of pages it needs to cover. It can be a simple list or a page-table-like data structure, with one or more levels of indirection.

Trust Domain tdMap: Every time the hypervisor encounters a new trust domain—i.e. a module requests to be launched as part of a trust domain different from that of previously launched modules—, it registers the new trust domain in the tdMap data structure, depicted in Figure 5.5. This map is keyed with the trust domain hash, which points to a memory region containing the trust domain descriptor: the list of (module hash, security segment hash) for every module in the trust domain. With the information contained in this map, it is possible to reconstruct the identity of a module X in the trust domain, by hashing together the trust domain hash with X's module hash and security segment hash.

5.1.2 SECURE_LAUNCH Hypercall

The SECURE_LAUNCH hypercall is invoked by OS and application modules to request a secure execution compartment from the Bastion trusted computing base. It differs from the `secure_launch` instruction for hypervisor launch in that it is handled by the hypervisor rather than an on-chip routine, and it allows for customization of TCB protections. The definition of a module and the custom TCB protection it requests are specified in each module's security segment, which is passed to the hypervisor's Secure Launch handler via a pointer argument to the hypercall. A second argument to the hypercall is the trust domain descriptor, listing the identities of modules trusted by the module being launched. This descriptor is used by the hypervisor to determine whether sharing interfaces should be established with previously launched modules.

Identity Computation: In its processing of the hypercall, the SECURE_LAUNCH handler first allocates a Module State Table entry for the module, and then measures the module's identity. The identity of a module is a cryptographic hash computed over the concatenation of three cryptographic hashes: the hash over the module's initial state (*module hash*), the hash over the module's security segment (*security segment hash*) and the hash over the module's trust domain descriptor (*trust domain hash*). To compute a module's identity, the hypervisor's SECURE_LAUNCH handler first reads the security segment and the trust domain descriptor into its protected memory space and computes the security segment and trust domain hashes. Guided by the module definition in the security segment, the handler then reads the module's initial code and data state to compute the module hash. Finally, the handler computes the module identity, a compounded hash over these three hash values, and stores it in the module's Module State Table entry. The standalone trust domain hash is also stored in the table so it can be used to reference the module's full trust domain descriptor in the tdMap. A trust domain hash which does not correspond to any tdMap entry means a new trust domain is being requested by the launched module. In this case, the hypervisor allocates a new tdMap entry, where it copies the trust domain descriptor and hash.

Runtime Protection Setup: As described in Chapter 3, the security segment specifies how module pages must be protected by the hardware memory integrity and confidentiality engines. Using their memory-mapped command interface, the hypervisor directs the on-chip engines to apply these protections to module state, as it is fetched on-chip. The hypervisor asks the engines to encrypt module pages with a `c_bit` set to 1 in the security segment and add to the integrity tree's coverage the module pages with an `i_bit` set to 1, a procedure described in Section 5.3. This takes place in parallel with the computation of the module identity so the initial module memory state considered as valid by the memory protection mechanism is identical to that included in the module identity computation.

`module_id` Assignment: In the processor hardware and within hypervisor data structures, modules are identified by their `module_id`, a shorthand for their full identity hash. The hypervisor allocates a module being launched its own `module_id` by allocating it a Module State Table entry and writing its module identity to that entry. A `module_id` is valid for at most one power cycle—the period of time between two platform resets. The hypervisor assigns each module it securely launches the next available `module_id`. The first module launched is assigned `module_id` 1, the second module is assigned `module_id` 2, etc. When a module is properly terminated via the new SECURE_RETIRE hypercall, its `module_id` can be reassigned to another module during a subsequent invocation of SECURE_LAUNCH. The `module_id` is used in hardware structures so its bit width must be much smaller than that of a full hash (typically between 128 and 512 bits), but it must also be wide enough to accommodate a large number of concurrent modules; between 8 and 20 bits should be sufficient. The reserved `module_id` 0 is assigned by default to all software that has not been securely launched. All unprotected OS or application software on the platform is thus conceptually part of the untrusted *module zero*, with its data and code pages assigned `module_id` 0 in hardware and in hypervisor data structures. Since module zero spans so many different software entities, it is not assigned an entry in the Module State Table. The *i*th entry of the Module State Table is assigned to module_id *i*, starting with the first entry being assigned to module_id 1

Shared Interfaces Setup: As part of SECURE_LAUNCH, the hypervisor also sets up the memory protection and sharing interfaces requested by the module. As it does so, the handler checks whether the module's definition and requested protections are compatible with previously launched modules—i.e., the private pages of different modules do not overlap and trusted software modules sharing a page agree on one another's identity. For each machine page P that M_L (the module being launched) requests as private (i.e. not shared), the hypervisor checks that no previously launched module was given access to P. This is done by looking up the mMap for P. Since all pages are by default accessible to module zero, the hypervisor must also revoke the default module zero access.

In Bastion, any two trusted software modules sharing a page must be part of a common trust domain. Using the tdMap and the module hashes specified in the security segment's Shared Pages directives, the hypervisor first checks that M_L is only requesting sharing of pages with either module zero or with a trusted software module within M_L's trust domain. For each machine page P_S that M_L wants to share with modules in the set $M_S = \{M_1, M_2, \ldots, M_n\}$, the hypervisor then checks, using the mMap, that only modules in M_S have access to P_S. Then for each previously launched M_i in M_S, it checks, in M_i's Module State Table entry (Shared Interfaces field), that M_i had requested P_S as shared with $(M_S \setminus \{M_i\}) \cup \{M_L\}$[17]. When module zero is not in M_S, the hypervisor must also revoke the default module zero access to P_S. Modules in M_S that have not been launched yet are ignored since potential conflicts will be detected upon handling the SECURE_LAUNCH hypercall for these modules.

Committing Module State to Hypervisor Data Structures: When all private and shared pages in a module pass the checks above, the hypervisor commits the module's definition to its Bastion data structures: the Module State Table, the module's vMap, the machine mMap. If any check fails, the module's module_id is revoked (i.e., its Module State Table entry is wiped out and marked as invalid and the Secure Retirement procedure is applied to allow for reuse of the module_id), the SECURE_LAUNCH hypercall returns and the software module executes as module zero, without any access to Bastion security services. On a successful SECURE_LAUNCH, the hypervisor first populates the Module State Table entry for the module with the authorized entry points and the shared memory interfaces defined in its security segment. It also writes the current virtual address space identifier in the *context_id* field. The hypervisor sets the module's authorized entry point list and private module stack pointer based on the security segment, and zeroes out the other fields of the *Transition Registry* since no transition has occurred yet for this module. Finally, the hypervisor looks up its own secure storage area to see if the module being launched is the owner of an existing storage area. If so, it stores the related key and hash in the module's Module State Table entry. Bastion secure storage is discussed in detail in Chapter 6.

[17] Set theory notation: the \ is the relative complement operator (A \ B means the set formed of all elements of A that are not also in B) and ∪ is the set union operator.

When Nested Paging is used, the hypervisor allocates memory for a module's vMap and writes a pointer to that memory in the vMap pointer field. It then writes in that vMap the virtual-to-machine translations (with access rights) for all pages to which the module has access. To obtain the machine address of each virtual page, the hypervisor must first get the guest physical address for the page and translate it to a machine address. To get the guest physical address, the hypervisor can either 1) walk guest page tables or 2) force the guest OS to service a fake TLB miss and intercept its write of the virtual-to-guest-physical translation to the TLB. The method to use depends on the underlying hardware architecture. In x86 [Intel 2009] for example, the hypervisor can read all translations directly from the guest page tables while in UltraSPARC hyperprivileged edition [Saulsbury 2008], the hypervisor only has access to a cache of most frequently accessed translations.

When shadow page tables are used, the module's vMap pointer is the base address of the shadow page table for the virtual address space containing the module. This shadow page table was constructed by regular hypervisor logic upon the launch of the application or OS containing the module. To bind the module to its pages in the vMap, the hypervisor writes the module's `module_id` in shadow page table entries relating to the module's pages. In all cases, the `c_bit` and `i_bit` for module pages, read from the security segment, are also copied in the vMap. We describe initialization and runtime usage of the vMap in more detail below.

5.2 Runtime Memory Protection Against Software Attacks

5.2.1 Shadow Access Control

Bastion compartments are protected against software attacks using a mechanism we call *Shadow Access Control*. We define a software attack against a module's compartment as a read or write to one of the module's memory pages by unauthorized software—i.e., software that was not explicitly given read or write access to the page during the module's secure launch. The attack vector in this case is an instruction fetch (e.g., resulting from a jump instruction or a program counter incrementation) or a data load or store instruction executed by unauthorized software on the processor. We deploy a component of our defense mechanism in the processor core itself to detect these unauthorized memory operations at the source and prevent them from corrupting or leaking protected module state. The processor enforces the module memory bindings committed to the mMap and module vMaps by the hypervisor's SECURE_LAUNCH hypercall. As a result, a module's private pages are only accessible to the module itself, and its shared pages are only accessible to the modules explicitly identified in the module's security segment.

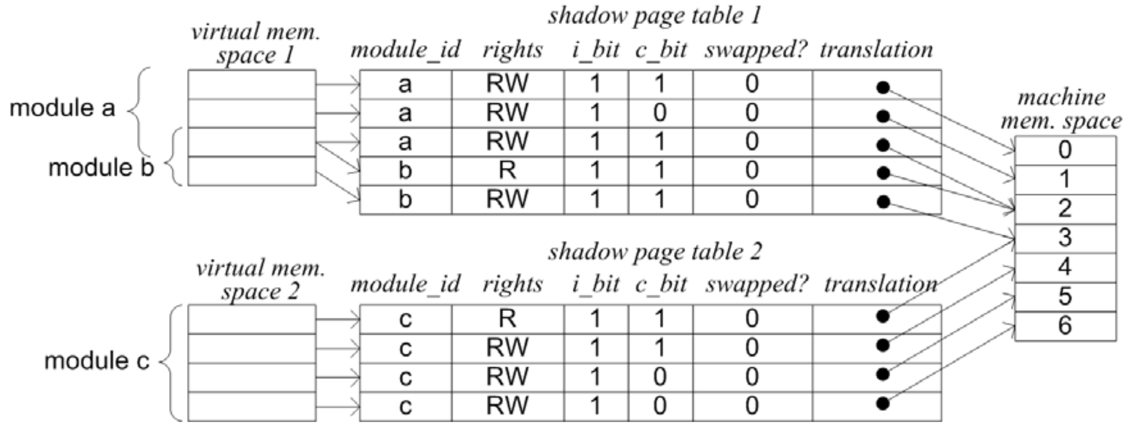
In the processor core, Shadow Access Control is implemented as a simple access control scheme, enforced on every access to memory. A module is only allowed to access a memory page if the page is associated with the module's `module_id` in the TLB. A new *current_module_id* register in the processor identifies the module currently executing using its module_id. Each instruction and data TLB entry is also extended with the module_id of a module authorized to access the page. To enforce module memory compartmentalization—i.e., protect modules against software attacks—the Bastion processor only executes memory operations that access instruction or data pages tagged with a module_id equal to the module_id in the *current_module_id* register. Bastion integrates this Shadow Access Control check as part of the TLB lookup performed on every memory access. Our extended hardware MMU logic only triggers a TLB hit condition if a TLB entry for the missing virtual page is tagged with the module_id in *current_module_id*. The hypervisor component of Shadow Access Control is responsible for managing the *current_module_id* register and the module_id tags in the TLB. The hypervisor sets these tags so they express the module-to-memory-page bindings defined in the mMap and module vMaps.

Upon module entry or exit, the secure inter-module transition mechanisms described in Section 5.4 allow the Bastion hypervisor to intervene and set the *current_module_id* register accordingly. module_id tags in the TLB are set as TLB miss events are serviced, including the new miss events caused by a TLB mismatch. On a platform with software-handled TLB misses, the hypervisor services TLB miss events. On platforms with hardware-handled TLB misses, they are serviced by a hardware page table walker. The module_id assigned to each TLB entry comes from module vMaps in hypervisor memory space. The method for forming a complete TLB entry—the missing page's virtual address, the corresponding machine address, access rights and the module_id—thus depends on the technique used to virtualize memory, either shadow page tables or nested paging. The idea of providing metadata via the TLB was detailed in [Champagne et al. 2009].

5.2.2 Shadow Page Table Support

In hypervisors using shadow page tables, a module's vMap is the shadow page table created to keep track of virtual-to-machine address translations for the application or OS containing the module. When an application or OS contains multiple modules, the Module State Table vMap pointer fields for these modules all point to the common shadow page table. The entries of all shadow page tables are extended with a module_id field binding the virtual page either to a module explicitly authorized to access the page by SECURE_LAUNCH, or to the default module zero. To allow for shared pages, where the modules in the set M_S are all bound to a shared page P_S, Bastion replicates the shadow page table entry of P_S. Each replica is tagged with the module_id of a module in M_S and may be associated with different access rights, as specified in each module's security segment. A shared machine page mapped in two virtual address spaces may also be associated with different access rights. This allows for the establishment of shared memory regions where modules have asymmetric access rights, e.g. a one-way producer-consumer buffer between two modules.

Our extended shadow page table scheme is depicted in Figure 5.6, where modules a and b share machine page 2 and modules b and c share machine page 3, with asymmetric access rights. This is a conceptual view of our extended shadow page tables: in practice, duplicate entries for sharing in the same virtual address space should be organized as a linked list with a head entry in the table and a tail allocated outside the table. This ensures that the table is fully associative and can be indexed using the page virtual addresses.



module_id	rights	i_bit	c_bit	swapped?	translation
a	RW	1	1	0	●
a	RW	1	0	0	●
a	RW	1	1	0	●
b	R	1	1	0	●
b	RW	1	1	0	●

module_id	rights	i_bit	c_bit	swapped?	translation
c	R	1	1	0	●
c	RW	1	1	0	●
c	RW	1	0	0	●
c	RW	1	0	0	●

Figure 5.6. Our Extended Shadow Page Table

On a TLB miss, the shadow page table for the missing program is looked up by the hypervisor or the hardware page table walker. Hardware walkers need to be adapted to support the duplicate table entries for shared pages. The goal of the lookup is to find an entry which contains the virtual-to-machine address translation for the missing page and which bears the `module_id` of the currently executing module. If such an entry is found, it is inserted in the TLB to service the miss event and execution of the missing program is resumed. Failure to find such an entry means that either (1) the missing virtual page was never mapped to a machine page by the hypervisor, or (2) the virtual page has indeed been mapped to a machine page, but was never associated to the `module_id` of the currently executing module.

The first condition can only occur when software in module zero tries to access a page in module zero, since all trusted software module pages are mapped in the shadow page tables during SECURE_LAUNCH. In this case, which is confirmed by checking that the *current_module_id* register is set to zero, the virtual page is mapped to its machine page in the shadow page table, and associated with the default `module_id` 0. The software that caused the TLB miss event is restarted and the memory access completes successfully since a TLB match can now be found.

The second condition occurs when executing software tries to access a memory page it is not authorized to access. Either the executing software is in module zero and tries to access a private module page, or executing software is part of a trusted software module and tries to access another module's page to which it was not given access. The hypervisor then denies or restricts the request, depending on the requestor. A guest OS is given access to an encrypted and hashed version of the page 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, as discussed in Section 5.2.5. Encryption and hashing are provided by the runtime memory protection mechanisms described in Section 5.3. Unauthorized accesses by applications are denied by resuming the program without executing the offending memory operation, at the instruction following the faulting access[18].

5.2.3 Nested Page Table Support

Hypervisors using nested paging create one vMap per module, mapping all the module's virtual pages to their machine pages. During the SECURE_LAUNCH hypercall for the module, each vMap is populated with the module's virtual-to-machine translations, and the associated access rights, and i_bit/c_bit. During runtime, TLB misses are handled by the hypervisor or by the hardware page table walker as usual, except for an extra final step. With nested paging, a regular TLB miss involves looking up the guest page table for the offending program in parallel with the hypervisor's nested page table for that guest. The result of these parallel table walks is a pair of address translations—one virtual-to-guest-physical and one guest-physical-to-machine—that are combined together to produce a virtual-to-machine translation for the missing TLB entry.

In Bastion, the hypervisor is invoked in an extra step to add the module_id, i_bit and c_bit and to the TLB entry at the end of a successful walk. When a hardware page table walker is used, a new trap must be generated by the hardware upon a successful walk to trigger hypervisor intervention. The hypervisor looks up the offending module's vMap to confirm that the translation in the newly formed TLB entry is valid for that module, and to add the module's module_id, i_bit and c_bit to the entry. The extended TLB entry can then be inserted in hardware, and the triggering program resumed. When the missing virtual page is absent from the module's vMap, or when the mMap specifies the module is not allowed access to the page, the hypervisor denies or restricts the access. The access is simply denied if requested by an application. The access is restricted to an encrypted and hashed copy of the page if requested by an OS, as described in Section 5.3.

5.2.4 Dynamic Module Memory Allocation

During runtime, a Bastion module may push data on the stack using the stack pointer, stored either in a dedicated register defined in the ISA or in a register reserved by convention. This stack pointer initially points to a memory region allocated for the stack by the untrusted OS loader. From one invocation of the Bastion module to the next, the value of the stack pointer may differ, depending on the exact function call sequence that preceded the invocation. This means that a given module does not always use the same set of stack pages. As a result, the security segment cannot statically define a set of pages dedicated to the module within the larger application or OS stack. The same logic applies to module requests for heap memory, which may be satisfied with any chunk of memory within the application or OS heap.

[18] Where necessary for debugging, the hypervisor could also trigger a processor-specific segmentation fault or I/O error to halt the program and allow a debugger to examine the offending operation. Bastion support for debugging is left to future work.

To support dynamic memory allocation for modules isolated in their compartments, Bastion provides each module with its own heap and stack. Module programmers need to specify heap and stack pages in the security segment so they are treated as module pages by Shadow Access Control. This ensures that during runtime, module data stored to the stack and heap are protected against software and hardware attacks just like the statically allocated part of the module's data space.

Stack: The security segment of each protected Bastion module has a *Stack Pointer* field with a value pointing to the top of a virtual address space region dedicated to the module's stack. This region should be reserved by the compiler to avoid having any data or code mapped to it during compilation. Upon processing the SECURE_LAUNCH hypercall for a module, the Bastion hypervisor registers the module's stack pointer in the Module State Table.

As it branches to the module's code during an authorized CALL_MODULE transition, the hypervisor substitutes the module's private stack pointer for the current stack pointer obtained from the Module State Table. The substituted stack pointer, which may have been pointing to an untrusted OS or application stack or to another protected module's stack, is saved in the Module State Table, in the *Saved Stack Pointer* field. The module thus starts executing using a private stack within its own compartment. When the module completes its execution and invokes RETURN_MODULE, the hypervisor restores the stack pointer that prevailed before CALL_MODULE, using the value in the *Saved Stack Pointer* of the Module State Table.

To exchange data parameters between two trusted modules via the stack, both modules must be defined so that they share the same stack. To keep the rest of their stacks inaccessible to one another, the modules must use the shared stack only for parameter passing and switch to a private stack themselves immediately after they are invoked. In future iterations of the Bastion architecture, the semantics of the CALL_MODULE hypercall could be extended to include an option for having the TCB copy invocation parameters on the callee module's private stack. This would make passing of parameters via the stack more transparent for both the caller and callee.

Heap: The security segment should also include a set of pages reserved for a private module heap, to be added to the module's compartment during SECURE_LAUNCH. Memory on the module heap is allocated and freed using the same functions as for a regular heap—e.g. `malloc`, `realloc`, `free`, `sbrk`, etc. However, each module must have its own copy of these functions[19], as they perform security-sensitive operations that cannot be entrusted to outside software. These functions must be compiled to use the private module heap pages rather than the untrusted heap pages, typically located at the end of the data segment. To share heap data between two Bastion modules, both modules must be compiled so that they share the same heap. Wrappers to heap management functions can also be written to allow a module to handle a single private heap and multiple shared heaps for the modules with which it collaborates.

[19] This amounts to only 746 lines of code in our implementation, presented in Chapter 7.

5.2.5 Runtime Relocation of Pages

As part of their management of the guest physical memory space, guest operating systems may relocate module pages within guest physical memory or write them to disk temporarily, in an on-disk *swap file*. These relocations are legitimate OS behavior, as long as the OS does not change the contents of the module's page, and properly updates the memory mappings in guest page tables. Bastion allows these relocations to take place by giving guest operating systems restricted access to a hashed and encrypted version of protected module pages. An OS can thus read a module page at a given memory address and copy it to the address where it is being relocated, either in memory or on disk. The Bastion hypervisor detects these relocations and updates its dMap, mMap and vMaps accordingly. It does not immediately check the integrity of relocated pages. Rather, it instructs the memory integrity checking engine to keep track of the page's integrity in its new position (described in Section 5.3). This way, any illegitimate change to the relocated page will be detected by the on-chip engine if and when the module accesses the affected page data.

Bastion needs to detect the relocation of protected module pages to enable a continuous protection of their integrity by the on-chip engine. Detection need not happen immediately, as the OS starts relocating the page. However, relocation must be detected prior to the module's first access to the page following its relocation. This allows the hypervisor to intervene and instruct the on-chip memory integrity engine to expect the module's virtual page in its new location. Hypervisors using shadow page tables detect relocations by intercepting every write the OS makes to its page tables. Hypervisors using nested page tables, however, do not track guest page table writes. Therefore, we add a check at the end of a successful nested page table walk. If the virtual-to-machine address translation in the TLB entry constructed by the walk is different from the translation currently in the module's vMap, the hypervisor considers the page was relocated. Malicious relocation is detected using the procedures described in Section 5.3.2.

A virtual module page V_A may be relocated to a machine page frame M that contains a different virtual module page V_B. In this case, the OS first relocates V_B out of M to an available machine page, or to the disk file. The OS can then relocate V_A to M without losing the information in V_B. However, if the relocated V_A is accessed before the relocated V_B, the Bastion hypervisor may detect the move of V_A into M but not the move of V_B out of M. To keep track of V_B despite being unaware of its current location, the Bastion hypervisor treats it as if it was on-disk, where it cannot be accessed directly by the module. The hypervisor thus assigns V_B a disk page in the dMap. When the module does access V_B again from its new machine page frame, the hypervisor deallocates the dMap page and updates the mMap to move V_B from its disk page to its new machine page. As it moves a page to machine memory, the hypervisor instructs the memory integrity engine to keep track of the page in its new position in RAM. This is described in further detail in the next section.

5.3 Runtime Memory Protection Against Hardware Attacks

Bastion provides runtime memory protection against hardware attacks using two on-chip engines: an encryption engine for memory confidentiality and an engine maintaining the Bastion integrity tree for memory integrity. The Bastion integrity tree covers the memory of both the hypervisor and an arbitrary number of protected OS and application modules. While the hypervisor layer is thin enough to fit entirely in a reserved region of machine memory, the set of modules that need to be protected at a given point in time may be too large to fit in machine memory. Guest operating systems deal with this shortage of memory resources by using on-disk swap files as an extension of their guest physical memory spaces. To protect hypervisor and module integrity, the Bastion integrity tree must thus cover not only the machine memory space in the RAM chips but also the module virtual pages swapped to disk by guest operating systems. The Bastion integrity tree accomplishes this by covering hypervisor and module pages in machine memory, including the special pages containing the dMap. Protecting the integrity of the dMap—which contains a fingerprint of each page currently sitting on-disk—effectively protects the integrity of modules pages on disk.

5.3.1 Protection Setup

A module's memory protections are activated by the hypervisor during SECURE_LAUNCH for the module. The hypervisor sends every module page with a `c_bit` set in its security segment to be encrypted by the on-chip encryption engine, cache line by cache line. The cache lines in pages with a set `i_bit` are also sent to the on-chip integrity engine, where they are fingerprinted as part of the Bastion integrity tree scheme. By doing so, we say that the hypervisor *adds the module pages to the integrity tree*. To add a new page to the Bastion integrity tree's coverage, the hypervisor sends a command to the on-chip integrity tree engine. This command tells the engine that a page worth of data is about to be read on-chip, and that a page tree should be computed over this data. The engine computes recursive hashes over the cache lines in this page until a single page root is obtained. The engine then substitutes, in the existing tree, the null page root corresponding to this page with the newly computed page root. This replacement triggers updates to the tree's top cone such that from this point on, the on-chip tree root reflects the state of the module page just added to the tree.

As the integrity engine computes a new page tree, not all of the tree's leaves (cache lines) or intermediate nodes can be kept in the engine's buffers. The page itself may contain tens of kilobytes or even megabytes of data, and the intermediate nodes of the page tree can require memory capacity up to fifty percent of the page size[20]. Allocating on-chip buffers to hold an entire page and its tree may thus lead to unacceptable costs in on-chip memory capacity. The same is true of the initial page cone computed during platform initialization. The Bastion memory integrity engine must thus be able to compute trees without holding all the leaves and intermediate nodes on-chip. We adopt a technique proposed in [Gassend et al. 2003], where on-chip memory is freed by evicting

[20] The formula to compute the tree overhead O, depends on the arity A of the tree. $O = 1/(A\text{-}1)$ [Elbaz et al. 2009]

from internal buffers the child nodes of computed parent nodes. Parent nodes fingerprint their children so the parent nodes kept on-chip are effectively the roots of a set of hardware-rooted integrity trees covering their descendents (their children, and the children of their children, if any). Note that when caching of intermediate integrity nodes is enabled (see Appendix A for a discussion of this topic), the hardware integrity engine can use the L2 cache as a buffer.

5.3.2 Runtime Protection

During runtime, cache lines from protected module pages are decrypted or integrity verified by the on-chip engines as they are read from external memory. Similarly, protected cache lines are encrypted and fingerprinted by the on-chip engines before being written back to external memory. The engines determine which cache lines to process by inspecting the two new tag bits, the `i_bit` and `c_bit`, associated with each L2 cache line. These tag bits are written when the L2 cache line is allocated upon the L2 miss for the line. The value of the bits is a copy of the `i_bit` and `c_bit` found in the Bastion-extended TLB entry of the page containing the line, depicted in Figure 5.7. The TLB entry is looked up as part of the memory access that caused the L2 miss.

module_id	*virtual page address*	*machine page address*	*i_bit*	*c_bit*

Figure 5.7. Our Extended TLB Entry (processor-specific bits not depicted, e.g. read-only, privilege level, cacheable)

We first discuss the detailed semantics of the `i_bit` and `c_bit`, which not only provide trusted software modules with a private and tamper-evident memory space, but also enable restricted OS access to protected module pages, to allow for physical memory relocation and swapping of virtual pages to disk. As explained in Section 5.2.5, the hypervisor detects that virtual page relocation has taken place whenever it notices that the machine page associated with a virtual page has changed. These mappings are kept in each module's vMap, where the hypervisor can look up the last valid virtual-to-machine mapping and update it if it deems the relocation to be legitimate. We end this section with a discussion of the three basic forms of relocation handled by the hypervisor: memory-to-disk, memory-to-memory and disk-to-memory.

Restricted OS Access: The operating system's role as a manager of guest physical memory involves swapping virtual pages back and forth between a disk-based page file and guest physical memory. It may also require relocating virtual pages within guest physical memory. Since some of these relocated virtual pages may belong to Bastion-protected modules, the OS must be allowed to access protected pages (i.e., pages with either the `i_bit` or `c_bit` set to 1) to enable relocation. The OS memory manager is untrusted, however, so the Bastion TCB must only give the OS *restricted access* to protected pages—i.e., access to an encrypted and tamper-evident version of the protected pages. This is achieved by having the hypervisor manage the `i_bit` and `c_bit` so that they serve two roles: in addition to serving as encryption/hashing flags, they also serve as permissions for executing software.

As per the `c_bit`'s "permission" role, executing software is given access to (i.e., has the permission for) a decrypted version of a page P only if it has a TLB entry for P with a `c_bit` set to 1. As per the `c_bit`'s "encryption flag" role, only cache lines from a page with a `c_bit` set to 1 are encrypted upon a write from the CPU back to main memory. Similarly, as per the `i_bit`'s "permission" role, the software can legitimately modify a page (i.e., have the permission to reflect a modification into the integrity tree) only if it has a TLB entry for P with an `i_bit` set to 1. As per the `i_bit`'s "hashing flag" role, only cache lines from a page with an `i_bit` set to 1 are integrity-verified against the hardware-rooted tree upon read from main memory.

To give a trusted software module M full access to an encrypted and integrity-protected page P, the hypervisor maps P in the TLB with an `i_bit` and `c_bit` set to 1. The TLB entry is also tagged with M's `module_id`. To give an untrusted OS memory manager in module zero restricted access to P, the hypervisor maps P in the TLB with both the `i_bit` and `c_bit` set to zero, and tags the TLB entry with `module_id` zero. The OS can thus access P but without triggering neither encryption/decryption nor integrity tree verification/updates. Data accessed by the OS from P are fetched encrypted from main memory and sent directly to the caches. Our cache logic, enhanced to handle the new `i_bit` and `c_bit`, ensures that any plaintext copy of a line from P is first evicted before the encrypted version is brought in. The OS can thus read all the data in P to relocate P within physical memory or to write it to a disk swap file. The OS can also modify P, but its modifications are not reflected in the integrity tree.

Relocation within guest physical memory is detected by the hypervisor whenever M accesses P again, as described below. Upon access to the relocated P, the TLB maps P in the TLB with the new machine address, with the `i_bit` and `c_bit` set to 1 and with M's `module_id`. Once again, M can access the plaintext content of P, and all its accesses trigger integrity verifications. Any modification carried out by the OS during relocation is detected as a violation of P's integrity, since the on-chip integrity tree used to verify P does not reflect OS modifications to P. In other words, corruption of P by the OS during relocation is detected as soon as M accesses the corrupted data. Any corruption of module pages detected by the on-chip engine is reported by the hardware to the hypervisor via a trap. The hypervisor then applies the SECURE_RETIRE procedure on the module to lock it out of its secure storage and prevent it from carrying out the attestation procedure described in Chapter 6.

Memory to Disk Relocation: When the hypervisor detects relocation of a virtual module page to disk, it first requests the page's current page root from the on-chip memory integrity engine. It saves this value and tells the engine to replace the page root in the tree with a null page root. The hypervisor then allocates a new dMap entry for the page, where it copies the page's mMap entry and the saved page root. Finally, the hypervisor unmaps the module virtual page from the mMap by wiping out the mMap entry and updating the module's vMap so it points to the dMap entry. The integrity tree, which protects the integrity of all machine memory pages including the dMap, now protects the on-disk page's page root as a leaf and thus still protects the page itself. This is depicted in

Figure 4.2, in Chapter 4, where the dMap (located within the fourth page) protects the integrity of the fifth and sixth pages, which have been swapped to disk.

Memory to Memory Relocation: The hypervisor may also detect relocation of a virtual module page from a *source* machine memory page to a *destination* machine memory page. Again, the hypervisor requests the source page's page root from the on-chip memory engine and asks the engine to substitute it with a null page root. When the machine mMap indicates that the destination machine page is free, the hypervisor simply copies the source mMap entry to the destination mMap entry, before wiping out the source entry. The hypervisor also updates the source page's vMap entry so it points to the new machine page. To update the tree so it reflects the page's new position, the hypervisor also asks the engine to replace the destination page root with the source page root it just read from the engine. When the machine mMap indicates that the destination machine page already contains a protected virtual page, the hypervisor assumes this virtual page has undergone a relocation that has not been detected yet (this can only happen with nested page tables, since all relocations are detected on-the-fly with shadow page tables). Therefore, before it updates the mMap, vMap and Bastion integrity tree as above, the hypervisor applies the memory to disk relocation procedure to this virtual page. Hence the integrity of the page is protected until it is accessed again by a module, from a different machine memory location.

Disk to Memory Relocation: When the hypervisor detects a *source* virtual page has been relocated from disk to a *destination* machine memory location, it first looks at the destination machine page's entry in the mMap. If the machine page already contains a protected virtual page, it performs a memory to disk relocation procedure for that page. In all cases, the hypervisor re-activates integrity verification of the disk page in its new memory location. To do so, it copies the source page's page root from the dMap to the integrity tree, by asking the on-chip engine to overwrite the destination page root. The hypervisor then copies the source page's dMap entry (except for the page root) to the destination page's mMap entry. Finally, the hypervisor wipes out the source page's dMap entry.

5.4 Secure Inter-Compartment Transitions

Bastion compartments cannot provide execution security if they are not entered and left securely. Bastion compartments are entered when the module they contain are invoked and left when module execution completes. We call these *module invocation transitions*. A compartment may also be left when module execution is preempted by a guest operating system, on an interrupt or a trap, or by the hypervisor itself, on a virtual machine switch. The compartment containing a preempted module is reentered when module execution resumes from its preempted state. We call these *module preemption transitions*. Securing module invocation and preemption involves protecting sensitive register state across transitions, activating or deactivating module-specific memory protections and providing cross-authentication to modules calling one another.

In this section, we first define transition security more formally and provide some background on secure transitions in past computer architectures. We then present our approach to securing compartment transitions due to module invocation and preemption, within the Bastion threat model. Finally, we explain how modules can leverage these secure transition mechanisms, and other Bastion features to establish what we call *secure inter-module collaboration*.

5.4.1 Problem Statement

The main goal of the Bastion architecture is to protect the integrity and confidentiality of data manipulated by software modules running in Bastion compartments. The first step to achieving this security goal is to provide, using the mechanisms described in Sections 5.2 and 5.3, the module with a confidential and tamper-evident memory space. Module data stored in this secure memory environment cannot be directly read or written (snooped or corrupted) by attackers. For hardware attackers, memory encryption obfuscates reads and the memory integrity tree detects tampering via writes. For software attackers, Shadow Access Control prevents malicious code from performing a load or store to protected memory. Compartments are thus protected against *direct memory access* by attackers, where a memory datum is read or written 1) via a processor load or store instruction, 2) via access to external buses or 3) via fake memory transactions.

However, these mechanisms do not protect compartments against *indirect memory access*. Attackers can perform an indirect memory access by snooping on or corrupting memory state temporarily stored in processor registers by protected software, while this software is preempted. Attackers can also trick protected software into revealing or corrupting its own data. Within our threat model, there are two possible avenues for such attacks. Malicious software can invoke a protected module by jumping into its code at an illegitimate entry point, e.g., to skip security checks done at the legitimate entry point. Malicious software could also impersonate trustworthy software and invoke a protected module, asking it to release sensitive data to unprotected memory. These attacks are specific to the functionality of the software modules. Some modules may be written such that they never release or corrupt sensitive data, regardless of where they are entered or who requests the operation.

Invocation and preemption of protected software modules within an untrusted software stack thus require special mechanisms for secure compartment transitions. We define three elements of transition security:

Entry Point Authorization: The execution of code in a protected software module must start at an entry point defined as legitimate by the programmer of the module. There may be multiple authorized entry points, for example, that each correspond to a function made available by the module to outside software. When an entire application is defined to be a module, there is typically a single entry point corresponding to the address of the `main` function or the address of a special `_start` function initializing certain data variables and environment settings before `main` is invoked. In addition to those initial entry points, module code may be reentered during runtime, following a preemption by the OS or the hypervisor. The only entry point that is authorized on reentry is the address of the module instruction whose execution was preempted. When a caller module invokes a callee module, the caller's code is also reentered at runtime, as the callee completes its execution and returns into caller's code. In this case, the only entry point that should be authorized on reentry is the address of the caller instruction that followed the branch instruction into the callee[21].

Mediated Preemption: The register state of preempted modules must be protected from snooping and corruption by the software given CPU control following preemption. The register state of the preempted module must thus be inaccessible or unintelligible to preempting software. In addition, when the preempted module is resumed, its register state must be identical to what it was upon preemption. A mechanism for achieving this security objective in a hardware-only security architecture was introduced in the SP user-mode architecture [Lee et al. 2005] and adopted in the SP authority mode architecture [Dwoskin and Lee 2007]. In this thesis, we leverage the software component of our TCB to achieve mediated preemption with a hardware-software mechanism.

Authenticated Invocation: To allow software modules within a trust domain to collaborate in carrying out a security-critical task within an untrusted software stack, they must be able to authenticate one another. Caller modules must know which module they are calling, callee modules must know which module is calling them. There may be sensitive data in parameters passed to callees upon invocation or in the results returned to callers. The identity of callers must thus be made explicit to callees so that the latter can refuse to return results if it does not trust the former. Similarly, callers must be able to specify the identity of their callees to ensure that sensitive invocation parameters are made accessible only to callee modules they trust. This also allows callers to trust the results they receive from their callees.

5.4.2 Past Work on Secure Transitions

Virtual Memory Compartments: Computing platforms typically load each application in its own virtual address space. These address spaces can be seen as execution compartments isolated from one another by the operating system's virtual memory manager. The address of entry into an application's compartment is specified in the header of the application's binary file—e.g., an Executable and Linkable Format (ELF) file—parsed by the OS loader upon launching the application. This entry point, specified by the programmer of the application, typically corresponds to the address of `main` or `_start` functions. Hence when the OS is trustworthy and binary files are protected from attacks, the code base of applications can only be invoked via the `main` or `_start` functions.

[21] Certain Instruction Set Architectures (ISAs) allow for branch delay instructions. These follow the branch in the assembly source code and in the binary, but they execute before the first instruction of the branch target, while the processor is busy resolving the branch target. When branch delay instructions are used, the authorized reentry point into caller code must be the address of the first instruction following the branch delay instructions.

During runtime, the OS specifies the current execution compartment by telling the hardware which set of virtual-to-physical memory translations to use. As the application is preempted, the OS is in charge of saving and restoring the application's register state to avoid the leakage of information across compartments. An application cannot directly invoke application code in a different compartment, since there is no user space mechanism for switching virtual address spaces. Applications must instead go through the OS to invoke one another, using special Inter-Process Communication (IPC) mechanisms. The security of transitions across traditional virtual address space compartments thus depends on the OS. However, current commodity OSes do not measure the cryptographic identity of the software they load, so even a trustworthy OS is unable to provide the level of assurance required by entry point authorization, mediated preemption and authenticated invocation.

Memory Segmentation Compartments: Operating systems could also use segmentation on top of memory virtualization, in order to create smaller execution compartments, each associated with specific subsets (called *segments*) of the application's code and data. As opposed to virtual memory compartments, these smaller compartments can invoke one another, as long as they have been authorized to do so by the operating system. On Intel's x86 [Intel 2009], these self-contained application components are called *tasks*. User code can transition between tasks using special *far jump* or *call* instructions. These instructions take a segment selector as an operand, which points to a *task gate* data structure in memory describing the task being called. Upon executing the instruction, the processor validates the task gate (and the task descriptor it points to) and checks whether the caller has sufficient privilege to invoke the callee. If so, the processor carries out the call by jumping to the entry point specified in the task descriptor.

When the execution of a task is preempted by an interrupt, the processor saves the task register state in a task descriptor data structure in memory. By making task descriptor and task gate data structures accessible only to specific tasks, the operating system can control which task can be invoked by a given caller task. It can also specify authorized task entry points by writing them to the task descriptor data structures.

As with virtual memory compartments, however, task compartments are vulnerable to a compromised OS and cannot provide cryptographic guarantees as to the identity of software they contain. We also note that although x86 is ubiquitous, its hardware task management is very seldom used, presumably because of the high complexity and poor portability of the code required to set it up.

Virtual Machine Compartments: A Virtual Machine (VM) managed by a hypervisor is a coarser form of execution compartment. Similar to virtual address spaces, the virtual machine is a compartment defined by a zero-based address space, the guest physical address space. Software within a VM usually starts executing at the physical address corresponding to the underlying processor's reset vector. During runtime, this software may create multiple virtual memory compartments for the software it loads.

On traditional virtualized platforms, the hypervisor is only responsible for saving and restoring the state of preempted virtual machines, not the state of preempted virtual memory compartments within a VM. Software in a VM cannot directly invoke software in another VM by jumping to it. If applications want to collaborate together across VMs, they must request a sharing channel from the hypervisor, e.g. virtual network sockets or messaging mailboxes. Not all hypervisors allow for inter-VM communication.

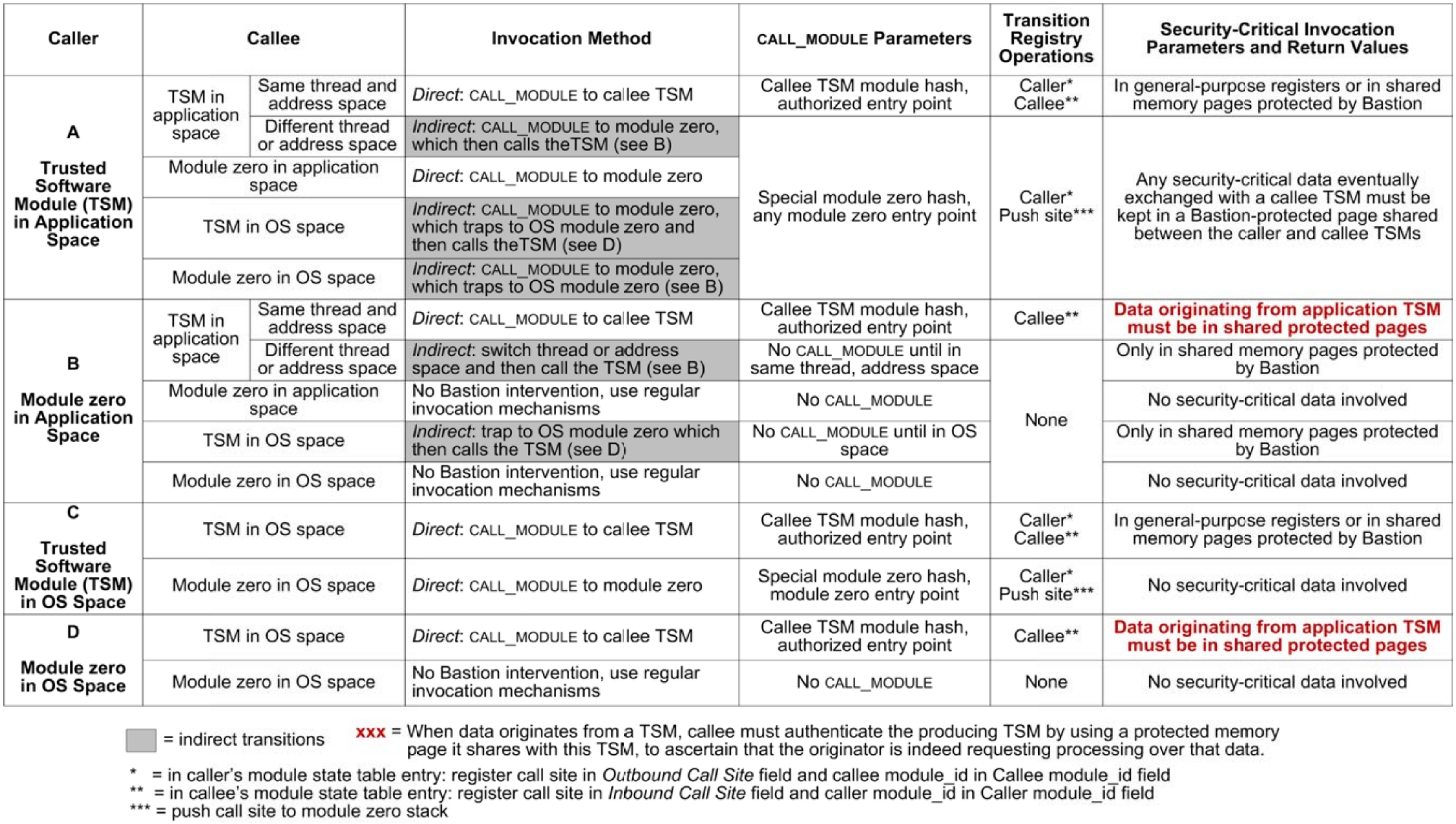
Although some hypervisors can measure the cryptographic identity of VM software (e.g. [Garfinkel et al. 2003]), existing hypervisors do not allow for collaborating VMs to authenticate one another. In any case, the hypervisor keeps track of its virtual machines in memory-based data structures. This means inter-VM transitions are vulnerable to hardware adversaries when the hypervisor runs on traditional hardware, which does not protect the virtualization layer's memory space against physical attacks. In addition, user-space hypervisors such as the Java Virtual Machine are also vulnerable to attacks by compromised or malicious OS code.

Hardware Compartments: Application compartments created by hardware security architectures such as XOM [Lie et al. 2000], AEGIS [Suh 2005] and SP [Lee et al. 2005, Dwoskin and Lee 2007] are protected against attacks from both hardware adversaries and untrusted OS code. In SP, the hardware itself provides a form of mediated preemption by protecting the register state of secure compartments with encryption and hashing. On an interrupt, registers are encrypted with an on-chip key and the resulting ciphertext is written back to the registers. A cryptographic hash is also computed over the ciphertext, and stored on-chip to detect tampering with registers during preemption. When CPU control switches back to the compartment, the processor checks the integrity of registers and decrypts them to allow the execution of the software in the compartment to resume. XOM hardware offers a similar service, but requires the untrusted operating system to explicitly request the encryption and decryption of individual registers.

In AEGIS [Suh 2005], the trusted part of the operating system, the *Security Kernel*, is responsible for saving and restoring the registers of protected compartments so they are unavailable to untrusted software. In XOM [Lie et al. 2000], collaboration across compartments must take place using unprotected shared memory, where data must be encrypted and authenticated using keys shared by collaborating software components. In AEGIS [Suh 2005], the OS must set up a shared memory region between two collaborating compartments. SP [Lee et al. 2005, Dwoskin and Lee 2007, Levin et al. 2009] hosts a single trust domain at a time where runtime sharing between compartments can be achieved via SP's secure storage mechanism.

Bastion Compartments: The Bastion architecture presented in this thesis allows for software modules in different compartments to invoke one another and exchange invocation parameters and results via authenticated shared memory interfaces. Mediation of module preemption by the hardware-protected hypervisor ensures that register state cannot be read or modified by hardware adversaries or software outside the compartment. Table 5.1 summarizes Bastion's handling of the possible types of inter-module transitions, described in more detail below.

Table 5.1. Overview of the Bastion Inter-Module Transition Mechanism



Caller	Callee		Invocation Method	CALL_MODULE Parameters	Transition Registry Operations	Security-Critical Invocation Parameters and Return Values
A **Trusted Software Module (TSM) in Application Space**	TSM in application space	Same thread and address space	*Direct*: CALL_MODULE to callee TSM	Callee TSM module hash, authorized entry point	Caller* Callee**	In general-purpose registers or in shared memory pages protected by Bastion
		Different thread or address space	*Indirect*: CALL_MODULE to module zero, which then calls theTSM (see B)	Special module zero hash, any module zero entry point	Caller* Push site***	Any security-critical data eventually exchanged with a callee TSM must be kept in a Bastion-protected page shared between the caller and callee TSMs
	Module zero in application space		*Direct*: CALL_MODULE to module zero			
	TSM in OS space		*Indirect*: CALL_MODULE to module zero, which traps to OS module zero and then calls theTSM (see D)			
	Module zero in OS space		*Indirect*: CALL_MODULE to module zero, which traps to OS module zero (see B)			
B **Module zero in Application Space**	TSM in application space	Same thread and address space	*Direct*: CALL_MODULE to callee TSM	Callee TSM module hash, authorized entry point	Callee**	**Data originating from application TSM must be in shared protected pages**
		Different thread or address space	*Indirect*: switch thread or address space and then call the TSM (see B)	No CALL_MODULE until in same thread, address space	None	Only in shared memory pages protected by Bastion
	Module zero in application space		No Bastion intervention, use regular invocation mechanisms	No CALL_MODULE		No security-critical data involved
	TSM in OS space		*Indirect*: trap to OS module zero which then calls the TSM (see D)	No CALL_MODULE until in OS space		Only in shared memory pages protected by Bastion
	Module zero in OS space		No Bastion intervention, use regular invocation mechanisms	No CALL_MODULE		No security-critical data involved
C **Trusted Software Module (TSM) in OS Space**	TSM in OS space		*Direct*: CALL_MODULE to callee TSM	Callee TSM module hash, authorized entry point	Caller* Callee**	In general-purpose registers or in shared memory pages protected by Bastion
	Module zero in OS space		*Direct*: CALL_MODULE to module zero	Special module zero hash, module zero entry point	Caller* Push site***	No security-critical data involved
D **Module zero in OS Space**	TSM in OS space		*Direct*: CALL_MODULE to callee TSM	Callee TSM module hash, authorized entry point	Callee**	**Data originating from application TSM must be in shared protected pages**
	Module zero in OS space		No Bastion intervention, use regular invocation mechanisms	No CALL_MODULE	None	No security-critical data involved

▨ = indirect transitions **xxx** = When data originates from a TSM, callee must authenticate the producing TSM by using a protected memory page it shares with this TSM, to ascertain that the originator is indeed requesting processing over that data.

* = in caller's module state table entry: register call site in *Outbound Call Site* field and callee module_id in Callee module_id field
** = in callee's module state table entry: register call site in *Inbound Call Site* field and caller module_id in Caller module_id field
*** = push call site to module zero stack

5.4.3 Module Invocation

In securing module invocation transitions, Bastion provides **entry point authorization** with two new hypercalls—CALL_MODULE and RETURN_MODULE. These hypercalls are used by modules to request inter-compartment transitions from the hypervisor. The hypervisor checks the legitimacy of the requested transition and updates the *current_module_id* register in hardware.

Software in virtual machines always starts executing as module zero, in the default, unprotected compartment. The *current_module_id* register is set to zero and code and data pages accessed by software must be tagged with `module_id` 0. Within a given virtual machine, the only way to transition out of module zero's compartment is to invoke the new CALL_MODULE hypercall. This hypercall takes two parameters: the module hash of the callee module and the virtual address of the requested entry point into the callee module. The Bastion hypervisor services the CALL_MODULE hypercall by first checking whether the requested entry point into the callee is legitimate. To do so, the hypervisor looks up, in the callee's Module State Table entry, the list of entry points authorized during SECURE_LAUNCH for the callee, identified by its module hash. If the entry point is authorized, the hypervisor saves the caller's reentry point and `module_id` in the callee's Transition Registry, in its Module State Table entry (*Inbound Call Site* and *Caller module_id* fields in Figure 5.1). This reentry point corresponds to the address of the caller instruction following the CALL_MODULE hypercall.

The hypervisor then writes the callee's `module_id` to the *current_module_id* register in hardware using the new `bastion_write` instruction. The hypervisor returns from the hypercall by jumping to the entry point into callee's code specified as an operand to CALL_MODULE. When the callee module completes its execution, it needs to invoke the new RETURN_MODULE hypercall in order to return to its caller. This hypercall does not take any parameter: the requested reentry point into the caller module's code is implicitly chosen to be the one saved upon the invocation of CALL_MODULE, so no further checks on its legitimacy are required (as explained below, modules are non-reentrant). The hypervisor writes the caller's `module_id` into the *current_module_id* register (obtained from the *Caller module_id* field in the callee's Module State Table entry) and returns from the hypercall by jumping to the caller's reentry point.

During their execution, callee modules may themselves call other modules using the CALL_MODULE hypercall. For calls to module zero, the calling site is registered in the caller's *Outbound Call Site* field of the Module State Table (see Figure 5.1), and the value zero is stored in the *Callee module_id* field. The calling site is also pushed on the module zero stack. This allows the hypervisor to distinguish between concurrent calls to module zero when handling an invocation of RETURN_MODULE by module zero. For calls to trusted software modules, the calling site is registered in the caller's Module State Table entry (*Outbound Call Site* field) as well as in the callee's (*Inbound Call Site* field), along with the caller and callee `module_id`'s. We do not register calls in a Module State Table entry for module zero, since module zero consists in an amalgam of many different pieces of software and can thus be the target of multiple simultaneous CALL_MODULE or RETURN_MODULE by different protected modules. Return addresses for *calls to* module zero are thus registered in the caller's Transition Registry. Similarly, return address for *calls from* module zero are registered in the callee's Transition Registry.

Due to the untrusted, unprotected nature of module zero, no guarantees can be obtained as to exactly which piece of code will execute when control is transferred to module zero. Therefore, the hypervisor allows all entry points in calls to the untrusted module zero. Enforcing specific entry points on calls to module zero would not provide additional security against a physical attacker or a corrupted OS. For example, if the hypervisor was to enforce that module zero be entered only via virtual address 0x2A00000, a physical attacker, or corrupted module zero code in the OS could overwrite the code at 0x2A00000 (which is part of the *unprotected* module zero) with arbitrary malicious code. It is up to the programmer of a trusted software module to write the module such that its security objectives cannot be thwarted by calls to compromised or outright malicious module zero code. In the worst case, a call to corrupted module zero code should result in denial of service (i.e., module zero code never invokes RETURN_MODULE to return to the caller module).

The Bastion architecture currently defines protected modules as being non-reentrant. This means that a given module must be exited before it can be called again. This ensures that our single Transition Registry per protected module will be sufficient to keep track of all pending transitions for a given module. Reentrance into protected module code is a topic left to future research. At first glance, it appears to require extending our current authenticated invocation mechanism and replacing the Transition Registry with a stack of Transition Registries.

5.4.4 Module Preemption

Software modules protected by Bastion may be preempted by interrupts—e.g., from an external device or the system timer—or by hardware traps due to a special condition occurring as the module executes, e.g. a page fault. To lower the overheads of virtualization, hardware platforms often let some of these traps and interrupts be routed directly to guest OS handlers, and specify only a few that must be handled by the hypervisor itself. An interrupt or trap condition may thus cause the CPU to jump directly from protected module code to untrusted guest OS code, without having the hypervisor intervene.

Shadow Access Control ensures that Bastion-enabled platforms invoke the hypervisor on such events, to avoid breaching compartmentalization (hence providing **Mediated Preemption**). This is due to the fact that when the processor jumps from a protected module (non-zero `module_id`) to an OS interrupt handler (`module_id` 0), the *current_module_id* register does not change. Fetching OS code pages tagged with `module_id` 0 thus causes a TLB miss trap due to a `module_id` mismatch. The performance overhead caused by the hypervisor's intervention are small, as shown in Chapter 8.

TLB miss traps are always handled by the hypervisor since they are essential to the hypervisor's ability to virtualize platform memory. As it handles the TLB miss trap, the hypervisor detects that the trap was due to the preemption of the current module by untrusted OS code. It saves the current module's registers in its Module State Table entry (*Saved Registers* field in Figure 5.1) and zeroes out the module's register state from the hardware. It also saves the current program counter of the module in its Transition Registry (*Preemption Site* field in Figure 5.1). Finally, the hypervisor writes the value zero to the *current_module_id* register and completes its servicing of the TLB miss by retrying the OS instruction that caused the TLB miss condition. This time, the instruction does not cause a TLB miss since there no longer is a mismatch between the OS code page `module_id` and the value in the processor's *current_module_id* register. The transition from the compartment containing the preempted module to the module zero compartment containing the OS interrupt (or trap) handler is thus complete.

Eventually, the operating system restores the preempted module's (zeroed) registers and jumps back to the module. This resumption may happen immediately after the OS completes its servicing of the trap or interrupt that caused the transition. The OS could also schedule a different program before it decides to schedule the preempted module again. In any case, the operating system's jump back into preempted module code causes a TLB miss due to `module_id` mismatch: the *current_module_id* register contains the value zero, while the module's code pages are tagged with its non-zero `module_id`.

The hypervisor intervenes to handle this miss event and detects that it corresponds to a resumption of the preempted module (by comparing the current Program Counter and *context_id* to the saved values in the Module State Table). It restores the module's program counter and registers from the Module State Table, and writes the module's `module_id` to the *current_module_id* register. The hypervisor completes its servicing of the TLB miss by retrying the module instruction that had initially been preempted. Again, the module instruction does not cause a TLB miss since there no longer is a mismatch between the module's code page `module_id` and the value in the processor's *current_module_id* register. The transition from the operating system's module zero compartment to the compartment containing the resumed module is thus complete.

5.4.5 Secure Inter-Module Collaboration

In providing **Authenticated Invocation**, the Bastion TCB offers a novel security service: *secure collaboration* between trusted modules in a trust domain, as well as with the untrusted module zero (which includes the commodity OS). Modules can collaborate within applications, within an OS, across applications, across privilege levels and across virtual machines. Any interaction a trusted software module has with protected modules from another trust domain is treated as collaboration with module zero.

We consider collaboration between two modules A and B as consisting of three types of interactions:

1) A invokes B

2) A transfers invocation parameters to B

3) B returns computation results back to A

Invocation parameters and return results can be exchanged between modules using one or more memory pages they share together. This is a secure interface for passing data that modules can specify. Shadow Access Control protects these shared pages from access by other modules, and runtime memory protection mechanisms ensure they cannot be observed or tampered with by physical attackers. To support existing software conventions, where function call parameters and return results are often passed via processor registers, our CALL_MODULE and RETURN_MODULE hypercalls leave register contents intact to allow for register passing of values. Thus, there are two ways to exchange data between caller and callee: registers and shared memory.

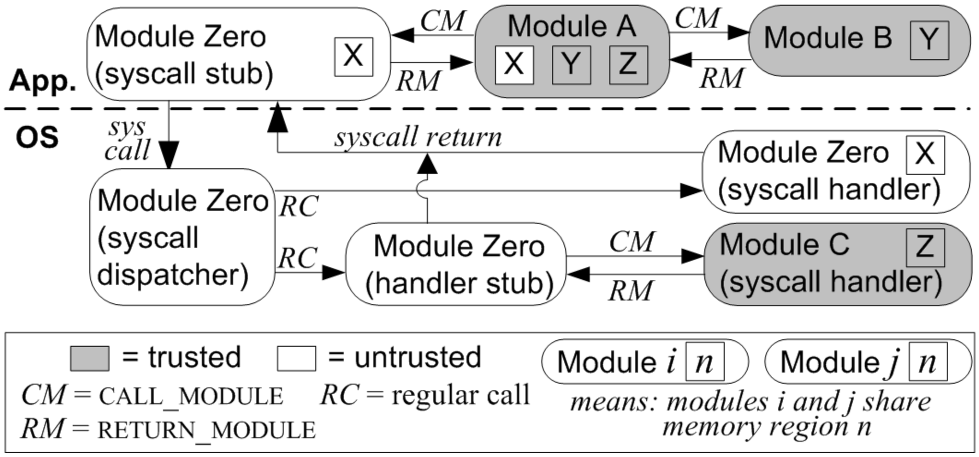

Module Zero (syscall stub)
X
CM
RM
Module A
X
Y
Z
CM
RM
Module B
Y
App.
OS
sys call
syscall return
Module Zero (syscall handler)
X
Module Zero (syscall dispatcher)
RC
RC
Module Zero (handler stub)
CM
RM
Module C (syscall handler)
Z
= trusted
= untrusted
CM = CALL_MODULE
RC = regular call
RM = RETURN_MODULE
Module i n
Module j n
means: modules i and j share memory region n

Figure 5.8. Secure Inter-Module Collaboration

Figure 5.8 depicts a trust domain composed of three trusted software modules within an untrusted software stack. Modules A and B are trusted modules in the same untrusted application running in user mode, while module C is a trusted system call handler within an untrusted commodity OS. To interface with untrusted code, modules A and C require stubs (the “syscall stub” and the “handler stub”, respectively), which are small code routines in the untrusted module zero. Module A collaborates with module B in application space, and it also makes system calls to an untrusted system call handler in module zero and to Module C, a trusted system call handler.

As an example, module A may be a Digital Rights Management (DRM) plug-in in a media player application, B is a crypto library, C contains the device driver for trusted display hardware (e.g., using High-Bandwidth Digital Content Protection, HDCP, technology [Lyle 2002]) and the untrusted system call handler is the file system manager used to access the untrusted disk. The owner of the content protected by the DRM module trusts A, B and C to enforce its security (DRM) policy on media files.

Untrusted System Call: When A needs to access files from the untrusted disk, it invokes the untrusted file system manager via a system call. To do so, module A must invoke the module zero system call stub with a CALL_MODULE hypercall, with invocation parameters in registers or in the memory region X it shares with the stub. The stub then carries out the system call, which gets handled by the untrusted handler. This handler returns into stub code, which resumes module A execution by invoking RETURN_MODULE. Status codes should be returned via registers while large data items produced by the system call handler for Module A are returned via the memory region X it shares with A.

Trusted Application-level Call: Assume that module A has a Message Authentication Code (MAC) key and a decryption key. It could have been provided to A using the secure I/O mechanisms described in Chapter 6. With these keys, A can decrypt the data it fetches from the untrusted disk, and verify their integrity. To perform decryption and verification, module A invokes functions from the cryptographic library in module B. The two modules are in the same thread and the same address space, so they can use the CALL_MODULE and RETURN_MODULE hypercalls to perform direct jumps to one another's code base. Some invocation parameters can be securely exchanged via registers, e.g., a constant identifying the type of cryptographic operation requested. Larger data items like the ciphertext data are passed between modules via the protected memory interface Y they both share.

Trusted System Call: Operating system components performing security-sensitive operations are encapsulated in new trusted OS modules. These OS-based modules must be accessible via a system call to allow for (trusted or untrusted) application-level software to request their services. In our example, module A requests the display of content-protected media frames from the trusted device driver in module C. In this case, invocation parameters that are not security-sensitive—e.g., the system call number—are passed to the untrusted "syscall stub" via registers. Parameters that are passed via registers are subject to corruption by the software in module zero. Hence any security-sensitive invocation parameter or computation result exchanged between A and C must be passed via the shared memory region Z, protected against software and hardware attacks.

An important design goal of our secure module collaboration mechanisms is to preserve compatibility with existing software conventions. To this end, we allow for both register and memory passing of arguments within virtual address spaces. We do not try to implement a new mechanism for direct transition between an application-based module and an OS-based module. This avoids breaking the existing system call control flow—i.e., an application triggers a software interrupt, the OS interrupt handler jumps to the system call dispatcher, the system call dispatcher jumps to the system call handler, the handler returns to application space using a "syscall return" directive. Rather, we use the existing system call path and add untrusted stubs to take care of transitions between trusted and untrusted domains. The only change we require is in the passing of parameters and return data between trusted software modules: they have to be written such that security-critical data is passed via shared memory interfaces protected by Bastion.

We can support OS-based trusted modules by loading them and their stubs via existing mechanisms such as Linux loadable kernel modules and Windows kernel extensions [Oney 2003], without having to modify the core OS code base.

Having to go through module zero to reach an OS-based handler gives malicious software the opportunity to drop or re-route the system call. From the point of view of the application, this amounts to denial of service, which we do not consider in this thesis.

Other Types of Calls: Several other collaboration scenarios follow the logic just discussed. For example, calls between two trusted modules across threads follow a similar path as for trusted system calls, with module zero stubs in both threads. The main difference is that a thread switch rather than a system call (privilege switch) must occur to transfer control from the caller module's stub to the callee module's stub. The same logic applies to inter-application and inter-VM calls, where the module zero stubs must trigger (or wait for) an application switch (e.g., a remote procedure call) or a VM switch (e.g., the VM containing the callee module gets scheduled by the hypervisor).

Module Cross-Authentication: An essential security objective in module collaboration is to ensure that callers and callees can authenticate one another to avoid impersonation. In our example, modules B and C must be able to tell whether they have been invoked by the trusted software module A or by some untrusted code in module zero. If impersonation were to happen, B and C might operate over corrupted invocation parameters, produce erroneous results or even be tricked into writing sensitive data to untrusted space. Similarly, A must be assured that its invocation parameters are accessible only to B (or C), and that the return results it receives were indeed produced by B (or C).

When invocation parameters and return results are exchanged via shared memory, impersonation is prevented by our Shadow Access Control mechanism, set up by the SECURE_LAUNCH hypercalls for both modules. Sharing is set up for specific memory pages only if both modules agree on one another's identity, as defined by their security segments. This means that at runtime, A can write to memory region Y and be assured that only module B can read the data. If module B is not launched, or if a corrupted version of it is launched, the shared memory interface will not be set up by the hypervisor, so the data written by A will remain accessible only to A. Similarly, A can read data from memory region Y and be confident that it was produced by module B, the only software entity, other than A, which has access to Y. This is also true for module B. The same properties hold for memory regions shared across privilege levels.

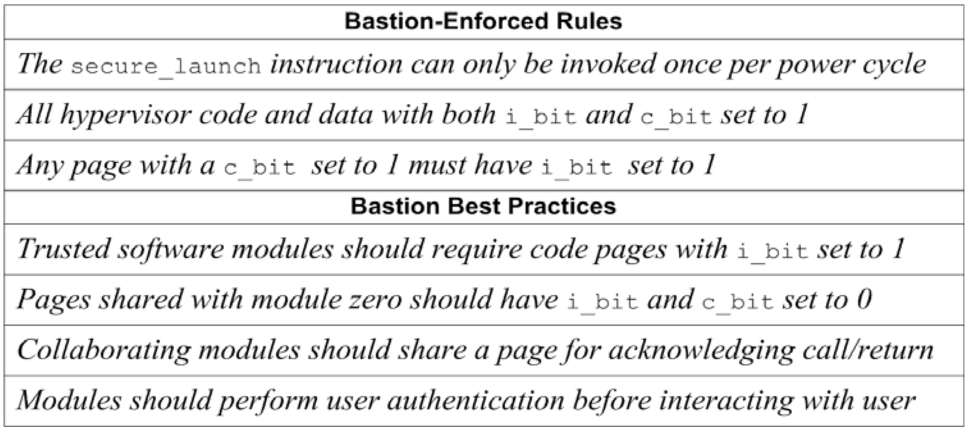
Invocation parameters and return results can only be exchanged via registers when the two collaborating modules are within the same address space, with the same privilege level. In this case, impersonation of the callee is prevented by the hypervisor as it processes the CALL_MODULE hypercall. The module hash used by the caller as an argument to CALL_MODULE refers to a specific module within the caller's trust domain. The hypervisor guarantees that the CALL_MODULE will either transfer control to the specified callee or return immediately to the caller (e.g., if the callee has not yet been launched). Hence, sensitive register values can only be revealed to the caller or the callee.

For calls to module zero, the module hash used in CALL_MODULE is the reserved module zero hash. It is up to the caller module to ensure that no sensitive information is left in the registers prior to the invocation of CALL_MODULE.

For the callee, authenticating the caller is less "automatic". The hypervisor does not restrict invocations of CALL_MODULE: any module, including module zero, can use CALL_MODULE to invoke a callee module, as long as the caller uses, when calling a module other than zero, an entry point authorized in the callee's security segment. It is up to the callee to identify the caller, and accordingly, determine whether it wants to satisfy the caller's request for processing. Identification of the caller by the callee can be done using the authenticated memory interfaces. In our example, module A can confirm that it is the caller asking for a specific operation by writing a pre-determined "I called you, to do XYZ, at what point" message to memory region Y. Module B can then confirm it was invoked by module A by reading this message from Y before doing any processing on the register values it received upon invocation. This also allows trusted software modules to adapt their computations based on the identity of their callers.

In summary, secure module collaboration consists of protecting invocation parameters, data and returned computation results from corruption, observation or impersonation. Corruption and snooping of memory-based data are prevented with Shadow Access Control and runtime memory protections provided to interfaces in shared memory pages. Caller and callee impersonation is prevented by a combination of Shadow Access Control and the SECURE_LAUNCH, and CALL_MODULE hypercalls. Register values are protected from corruption and observation by the hypervisor's handling of CALL_MODULE and RETURN_MODULE hypercalls. Table 5.2 provides a summary of important Bastion rules and best practices to clarify the difference between what invariants Bastion enforces and what trusted software modules should do to ensure they can execute securely to completion.

Table 5.2. Bastion Rules and Best Practices



Bastion-Enforced Rules
The `secure_launch` *instruction can only be invoked once per power cycle*
All hypervisor code and data with both `i_bit` *and* `c_bit` *set to 1*
Any page with a `c_bit` *set to 1 must have* `i_bit` *set to 1*
Bastion Best Practices
Trusted software modules should require code pages with `i_bit` *set to 1*
Pages shared with module zero should have `i_bit` *and* `c_bit` *set to 0*
Collaborating modules should share a page for acknowledging call/return
Modules should perform user authentication before interacting with user

5.5 Secure Module Retirement

A new SECURE_RETIRE hypercall can be invoked by modules wishing to tear down their secure execution compartment and make inaccessible all protected resources associated to it, in registers, VM memory and hypervisor data structures. To service this hypercall, the hypervisor first walks the module's vMap to iterate over every module page. For each virtual page in machine memory, the hypervisor checks whether the corresponding machine page is shared with other modules, by looking up the mMap. If the page is swapped on-disk (as indicated by the vMap's *swapped?* flag), the hypervisor checks for sharing by looking up the dMap. For a shared page, the hypervisor simply removes the access rights of the module being retired in the mMap or dMap entry of the shared page, and then deletes the module's vMap entry for the shared page. Other modules that had access rights to the shared page in their vMap keep their rights, ensuring they maintain access to any shared data they are authorized to access. This is useful, for example, to allow a consumer module to access the last outputs of a retiring producer module. For pages private to the module, the hypervisor deletes the mMap or dMap entry, as well as the vMap entry. Finally, for each private page with an `i_bit` set to 1 (which includes every private page with a `c_bit` set to 1), the hypervisor asks the on-chip memory integrity engine to replace the page's page root in the integrity tree with a null page root.

Once all module page access rights have been removed from the vMap, mMap and dMap, the hypervisor flushes all TLB and cache lines associated with the module's `module_id` in the processor. It also wipes out any processor register state belonging to the module. For modules that had requested a secure storage area, the hypervisor commits the latest state of the module's area, fingerprinted in the Module State Table, to the hypervisor's own processor-rooted secure storage area (see Chapter 6). Finally, the hypervisor zeroes out the module's Module State Table entry and returns from the hypercall, with the *current_module_id* register set to value zero. Following this procedure, the `module_id` just retired can safely be reused by the hypervisor, for assignment to a trusted software module in a subsequent invocation of SECURE_LAUNCH. Indeed, following this procedure, the Module State Table entry associated with this `module_id` is empty and all traces of this `module_id` have been removed from hypervisor data structures. The security of the secure retirement procedure is further analyzed in Chapter 8.

5.6 Compatibility Issues

5.6.1 Demand Paging

For the sake of simplicity, the Bastion architecture presented in this thesis requires that all trusted software module pages be mapped to a machine page upon SECURE_LAUNCH. This means that every heap or stack page that might get used during runtime needs to be allocated in guest physical memory before the module can start executing. While this requirement is not a problem for simple modules using small heaps and stacks, it may lead to large amounts of wasted guest physical memory when a module pre-allocates a large heap or stack, but uses only a small fraction of it during runtime. In modern operating systems, this problem is solved with the use of *demand paging* [Tanenbaum 2007], where physical memory is allocated only when a page is actually accessed for the first time. Unfortunately, the purpose of demand paging is defeated by our simple SECURE_LAUNCH since every single module page gets accessed before the first module instruction executes, in order to compute the module's identity and add its pages to our integrity tree.

Extending Bastion to support demand paging for trusted software modules is left to future work. However, we do note that a possible approach consists in forcing all initialized module memory (i.e., code, static data and initialized data or BSS, Block Started by Symbol) to be mapped to machine memory upon SECURE_LAUNCH, but leaving unmapped the virtual pages reserved for the uninitialized heap and stack. In this case, the module hash component of module identity can be computed over initialized memory and over a new architected descriptor specifying the location of heap and stack pages. In hypervisor data structures, the heap and stack pages can then be marked as unmapped, until they are accessed for the first time. Upon first access, the pages should be checked to be completely zeroed out, and then sent to on-chip encryption and hashing engine to be covered by Bastion memory protection mechanisms. The module hash computed initially thus reflects the initial state of module memory space from the module's point of view since these pages, described in compact form in the new descriptor, are guaranteed to be filled with zeroes upon first access. A similar idea was first introduced by the author of this thesis in [Champagne et al. 2008].

5.6.2 Multiple Page Size Support

For clarity, this thesis does not consider Multiple Page Size Support (MPSS) functionality, often implemented by modern operating systems. With MPSS operating systems, virtual page sizes may vary across applications, while Bastion hypervisor data structures and memory management hardware assume fixed size pages. Adding MPSS functionality would increase the complexity of hardware and software structures and obscure the fundamental role played by each architectural feature. While MPSS in Bastion is left to future work, we note that current Bastion structures and procedures are not incompatible with MPSS functionality. The fixed page size of current data structures can be chosen to be the smallest page size supported by an MPSS system. Hypervisor routines can then be enhanced to allocate multiple vMap and mMap entries to cover each large page.

5.6.3 No-Translation Address Spaces

In certain cases, the software running within a virtual machine may execute out of the guest physical address space, without the need for virtual address translation. This is the case with software running on an Intel x86 processor where "protected mode" [Intel 2009] has not been enabled; this software executes in "real mode", without virtual address translation. The processor architecture may also allow for some operating system code to execute out of guest physical memory, even though the applications it runs execute out of virtual memory. The Bastion hypervisor needs to be extended to handle modules running within such “no translation” address spaces. This could be done by adding a flag bit specifying whether virtual addresses in data structures like the vMap are actually guest physical addresses. Nested page tables and the mMap are not affected by this since there still is a need for guest-physical-to-machine address translations. Extension of Bastion to support modules executing within no-translation address spaces is left to future work.

5.6.4 Processor-Specific Conventions[22]

In this section, we discuss processor-specific conventions that may require extensions to the Bastion mechanisms described in this chapter. We do not architect such mechanisms into Bastion to remain independent from special features in the underlying processor hardware.

Processor architectures with hardware support for virtualization define hypervisor traps and guest OS traps. Events that must be handled by the hypervisor to avoid breaking virtual machine compartmentalization (e.g., TLB miss traps) are architected to be hyperprivileged traps, delivered to the hypervisor. Events that do not affect virtual machine compartmentalization (e.g., interrupt caused by a timer set by a guest OS) are architected as privileged traps, delivered directly to the guest OS. Because Bastion secure module compartments are effectively small light-weight virtual machines within a virtual machine, giving the Bastion hypervisor control over hyperprivileged traps is sufficient to ensure module compartmentalization is maintained. With the help of hyperprivileged traps, the hypervisor ensures TLB mappings enforce VM memory compartments and it also saves VM register state on a VM switch. Similarly, with Bastion’s extended semantics for the TLB miss (which now also considers the `module_id`), the hypervisor enforces module memory compartmentalization and saves module register state on module exit.

Virtual machines and Bastion module compartments differ in that the memory and register state of a module may be accessed from outside the module whereas the state of a VM usually is only accessed by the VM itself. For memory, this can happen when a module shares a page with outside software or when a guest OS relocates physical memory page frames. Mechanisms for handling these two common scenarios have been architected into the Bastion hypervisor, namely Shadow Access Control (Section 5.2) and Restricted OS access (Section 5.3.2). Callee and caller modules exchanging data via registers is also a common use case, which is handled by the CALL_MODULE and RETURN_MODULE hypercalls (Section 5.4.5).

[22] In this section, we use the hyperprivileged edition of the UltraSPARC architecture to guide our discussion about trap types. However, the same concepts apply to other processor architectures with hardware support for virtualization, albeit with some variation in the implementation of trap delivery mechanisms.

However, specific features of a processor's architecture may introduce new types of events where module memory or register state needs to be accessed from outside the module. In the UltraSPARC architecture for example, a new set of application registers, called a register window, is allocated on a function call from a hardware-based stack of register windows. When the resources of the hardware stack are depleted, the processor triggers a privileged register window *spill* trap, which is to be handled by the OS. To service this trap, the OS handlers must read application register state from the hardware and write (spill) it on the application's stack, in memory, in order to free a register window in hardware.

Similar trap events are architected for restoring (or *filling*) a hardware window with the register previously spilled on the stack, and for zeroing (*cleaning*) a register window that previously contained register state belonging to a different software context. The OS is unable to service such a register window spill, fill, or clean trap occurring during module execution. This is because the Bastion mechanisms defined in this chapter do not allow the untrusted OS trap handlers to read register state and write it to the private module stack or vice versa. A solution to this issue, which we applied in our implementation (described in Chapter 7), is to handle register window spill, fill and clean traps directly in the hypervisor, which has full access to module registers and stacks.

The Bastion TCB presented in this thesis aims to be independent of the underlying processor architecture. This is why Bastion does not include architected mechanisms to handle architecture-specific events such as the register window traps described above. Before porting Bastion to a specific processor, its architecture must be analyzed to determine whether some of its instructions, mechanisms and conventions break Bastion module isolation. If so, the Bastion hypervisor must be extended with processor-specific mechanisms to handle the offending scenarios. In most cases as in the register window example above, adding new event handlers in the hypervisor should be sufficient to maintain Bastion module compartmentalization. A full compatibility analysis of existing processor architectures is outside the scope of this thesis. We only provide support for the mechanisms and conventions that are constant across the different mainstream microprocessors available today.

5.7 Chapter Summary

In this chapter, we described in detail the five mechanisms Bastion uses to provide protected execution compartments for security-critical tasks. A secure launch procedure is used to define execution compartments to the hypervisor, who then sets up mechanisms for protecting runtime memory state in the compartment against software and hardware attacks. We also presented the Bastion approach to secure inter-compartment transitions and our secure retirement procedure for tearing down secure execution environments. In the next chapter, we present two architectural mechanisms used to provide Bastion compartments with secure I/O capabilities.

Chapter 6

Support for Secure Input and Output[23]

This chapter presents our approach to securing the input and output of trusted software modules running in Bastion-protected compartments. We first define key concepts for secure I/O and state our contribution to tackling part of this important research challenge. A complete solution is deemed outside the scope of this thesis, as the full problem involves significant interaction with the "real world", with its analog signals and human users. After some background on attestation, the chapter then discusses how our Tailored Attestation and Secure Persistent Storage primitives leverage the TCB security services described in Chapters 4 and 5 to provide generic secure I/O services to trusted software, that apply to user, disk and network I/O. We present a solution where Human Interface Devices (HIDs) can perform input and output operations on behalf of both trusted and untrusted software running on the platform. For example, we do not require the user to have two sets of keyboards and display monitors.

6.1 Problem Statement

Programs must have Input and Output (I/O) capabilities to be of practical use. Their I/O can be to and from the user, disks (and other forms of persistent storage), the network or other peripheral devices such as data acquisition cards and Universal Serial Bus controller cards. Inputs consist in commands telling the program what to compute, or in data over which the program is to carry out a computation. Outputs are either the data results of the program's computation or commands requesting computations from other (local or remote) software or hardware entities. This simple definition of I/O makes it clear that the protection of software modules provided by the Bastion Trusted Computing Base must include *secure I/O* capabilities for the module to achieve any meaningful security objective. Secure I/O mechanisms protect the confidentiality and integrity of a module's inputs and outputs between their source and destination. This includes having the source and destination of the data authenticate one another to ensure that security-critical data is not revealed to or accepted from untrusted entities. While the secure inter-

[23] Some of the ideas presented in this chapter first appeared in prior publications [Champagne and Lee 2010], [Champagne and Lee 2010b], [Lee and Champagne 2010]

module collaboration primitive presented in Chapter 5 addressed the problem of secure I/O between local software components, this chapter looks at securing I/O between a local Bastion-protected software module and an I/O endpoint which is not protected by the Bastion TCB.

Securing I/O with local and remote devices and users, across both digital and analog channels is a very complex problem that this thesis does not solve in its entirety. Rather, Bastion provides processor and hypervisor primitives that can enable I/O security when some tamper-resistant I/O hardware and informed users are in place. This chapter thus focuses on establishing secure channels between Bastion-protected software and trusted I/O endpoints. The following section defines important aspects of the secure I/O problem to better identify the contributions of Bastion mechanisms to its solution.

6.1.1 Secure I/O Definitions

Because one of the endpoints is a human being rather than a computing device, the secure I/O literature makes a distinction between user I/O and other forms of I/O. To achieve secure user I/O, a *trusted path* is needed, while secure I/O between computing devices requires a *secure channel*. Paths and channels both start at the computing device running the program of interest, i.e., the program considered to be the source of outputs and the target of inputs. A secure channel ends at a computing device communicating with the program of interest, while a trusted path ends, *conceptually*, in the brain of the user interacting with the program of interest. In both cases, the endpoint of the communication needs to be able to verify that the data outputs produced by the program of interest are genuine. The endpoint also needs to ensure that the program of interest can authenticate the data inputs it receives from the endpoint. This cross-authentication is necessary since the devices and buses on the way from the program of interest to the endpoint are outside the Bastion security perimeter (i.e., the processor chip), hence they are untrusted.

A trusted path can be seen as a *digital* channel from the processor to a Human Interface Device (HID) followed by an *analog* secure channel between the HID and the user. Trusted paths thus require two security mechanisms: 1) a secure channel between the processor and the HID; 2) a mechanism for a human user (i.e., the *brain* of the human user) to authenticate the HID and its outputs. This authentication is necessary to avoid having a malicious HID impersonate a trusted device. When such impersonation is successful, the user may be tricked into inputting sensitive information into the bad device, which might then leak the information. The user may also take an incorrect decision based on an output (e.g., information displayed on a monitor) from a malicious device. Unfortunately, traditional cryptographic methods such as public key signature verification are not possible with humans, who are extremely slow when it comes to computing functions such as the RSA exponentiation. Establishing a trusted path thus requires methods that are usually outside the scope of mathematically provable security, and more in the realm of laws and social conventions like reputation and interpersonal trust.

6.1.2 Secure I/O Example

An example of a trusted path is the link between a bank customer and the computer program inside an Automated Teller Machine (ATM). The HIDs involved in this scenario are the keyboard and card reader (inputs) of the ATM and its display (output). When customers approach the ATM, they must authenticate it before they start using it. This authentication could be based on a combination of many possible non-cryptographic factors: location (e.g., an ATM within a bank building is most likely genuine), appearance (e.g., the layout, shape and fonts of ATM components are recognizable, and may be registered trademarks or difficult to forge), history (e.g. the customer has successfully carried out transactions with this ATM before), context (e.g. many people are queuing up to use this particular ATM, looking confident that it is genuine), etc. Customers engage genuine ATMs to input sensitive data (e.g., their debit card's Personal Identification Number, PIN) and take security-critical decisions (e.g., transfer money between accounts) based on information displayed by the ATM. The ATM uses the customer's debit card and PIN to authenticate the customer. Both the user and the ATM trust that the components between the ATM's HIDs and the ATM's computer are out of reach for attackers. When it is not the case, encryption and message authentication protocols need to be set up between the HIDs and the computer.

In this scenario, we assume authenticating the ATM's HID devices is equivalent to authenticating the ATM's processor and software: genuine HIDs on the frontend mean there is a genuine processor and genuine software in the ATM on the backend. We can also assume that genuine HIDs guarantee the existence of a secure channel between these devices and the genuine processor. The idea is that a genuine, trustworthy HID will only communicate with a trustworthy processor via a secure channel. If it cannot find such a processor and secure channel, it will not display any outputs or accept any inputs. However, like most things in the physical world, the customer's authentication of the HID is not guaranteed to succeed. In the worst case, a failure of authentication means a customer will consider as genuine HIDs that are not genuine. The problem of human authentication of HIDs or other computing devices is outside the scope of this thesis. It is still an active field of research, where many approaches have been proposed, from visual cryptography [Naor and Shamir 1994] to using an authentication proxy such as a trusted cell phone [McCune et al. 2009]. More background on the secure I/O literature as it applies to Bastion is provided in the appendix to this chapter, in Section 6.A

6.2 Overview of Bastion Secure I/O

6.2.1 Secure I/O Trust Models

Our secure I/O trust model is depicted in Figure 6.1, and described in detail below. In this thesis, we consider that users are able to successfully authenticate the HIDs they interact with one hundred percent of the time (i.e., we do not address the problem of machine-to-user authentication). These HIDs are trusted I/O endpoints, as defined in the threat model in Chapter 2. They can produce and verify attestation reports in the process of establishing a secure channel with the software they communicate with. As with the processor chip, attackers cannot snoop on or tamper with the computation and storage elements within these I/O devices (i.e., trusted I/O endpoints are axiomatically trustworthy). Similarly, we assume that some form of user authentication is in place so that the Bastion-protected software receiving user inputs from the HIDs is able to identify the user interacting with the platform. Bastion does not offer architectural mechanisms for this user-to-machine authentication. It is up to security-critical software to ensure that proper user identification and authorization is carried out.

As opposed to the special-purpose ATM setup that only runs trusted software, our generalized computing model requires HIDs that can perform input and output operations on behalf of both trusted and untrusted software running on the platform. (We do not require the user to have two sets of keyboards and display monitors for example.) This requires HIDs that can identify the software currently controlling the devices, to allow the user to decide whether to trust the outputs of the software and whether to entrust it with user inputs. We assume that HIDs and users can perform this type of software identification. Identifying trusted software to human users is a component of the machine-to-user authentication process (which we do not consider in this thesis) and remains an open research area. We refer readers to the literature for a description of proposed solutions to this problem: e.g., dedicated screen regions [Ye and Smith 2006], human attestation [Toegl 2009]. The goal of the Bastion secure I/O mechanisms is to ensure that, given a tamper-resistant HID that can be authenticated by the user, the Bastion processor can identify a piece of trusted software to the HID and establish a secure channel between that piece of software and the HID. This in turn allows the HID to identify to the user the piece of trusted software he is currently interacting with.

As opposed to HIDs, the disks and other persistent storage devices are not considered to be trusted I/O endpoints in Bastion. They are just passive storage devices like main memory, and they are considered untrusted like main memory. Network interface cards and other local networking equipment and cables are also considered untrusted. Only remote networked devices considered as trusted I/O endpoints should be able to send and receive secure network I/O. Everything between the Bastion processor chip and a networked trusted I/O endpoint is subject to attack. For simplicity, we assume that all local I/O devices other than persistent storage devices and network interface cards are trusted I/O endpoints. To support secure I/O operations, Bastion must thus establish a secure communication channel (encrypted and authenticated) between Bastion-protected software (running on the processor chip identified in Figure 6.1) and trusted I/O endpoints. As described in more details below, this is achieved using a combination of our tailored attestation and secure storage services. Attestation allows Bastion-protected software to:

1) identify itself to a trusted I/O endpoint and

2) exchange shared secure channel keys.

Secure storage allows this software to:

1) store cryptographic materials for verifying the identity of the trusted I/O endpoint,
2) and store the secure channel keys exchanged for future interactions between itself and the endpoint.

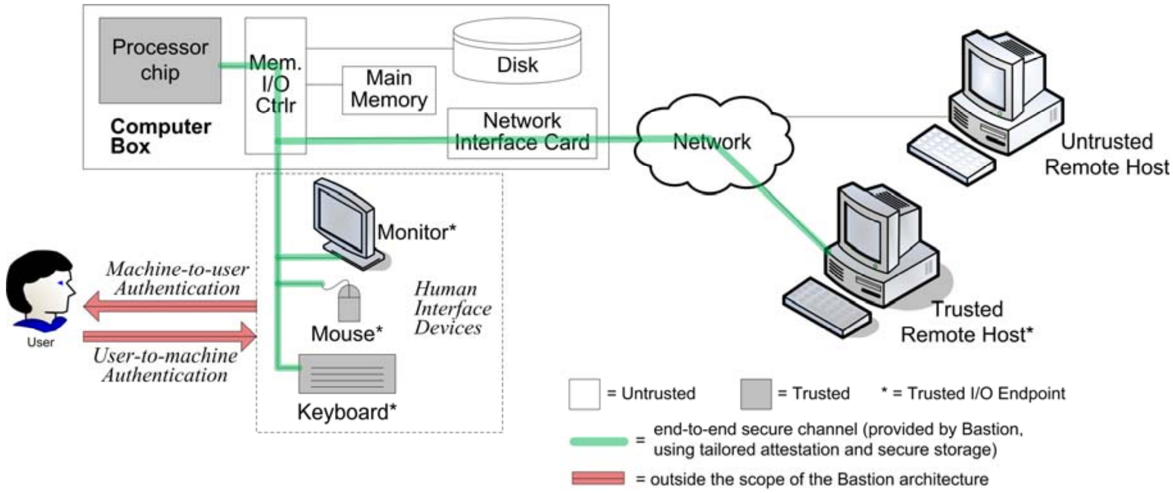


Figure 6.1. Trust Model for Bastion Secure I/O

6.2.2 Bastion Secure I/O Primitives

The Bastion architecture provides two primitives, *Tailored Attestation* and *Secure Persistent Storage*, to support secure user, disk, local and network I/O for protected software modules. Attestation allows each software module to identify itself to software or hardware entities outside the trusted Bastion microprocessor chip. Secure persistent storage provides a software module with a confidential and tamper-evident non-volatile area to store security-critical information that is available across platform resets. Protected software modules use these two primitives to carry out cross-authentication with trusted I/O endpoints. On the one hand, cryptographic keys or other secret data stored in secure persistent storage allow a module to authenticate messages coming from a trusted I/O endpoint. On the other hand, the endpoint itself can authenticate a software module by examining an attestation report the software module produced using the attestation mechanism. As part of this cross-authentication procedure, the trusted I/O endpoint and the software module exchange symmetric keys that are then used to establish an encrypted and authenticated secure channel between one another. The only runtime difference with regular I/O is thus that data on a secure I/O channel is encrypted rather than in plaintext, and accompanied with some form of Message Authentication Code (MAC). While this method of securing I/O transactions increases the amount of data that transits on the I/O channel (because of the MACs), it does not require changes in the traditional mechanisms used by software to communicate with I/O devices, as described below.

6.2.3 Support for Existing I/O Models[24]

Bastion supports I/O transactions performed via DMA. It does not require a change to the hardware or software TCB to do so, but it does require software modules to allocate buffer memory differently for storing DMA I/O data. In memory mapped I/O, every read and write of I/O data goes through the processor and is requested by the software module itself. As depicted in Figure 6.2-a, a Bastion-protected module executing on the

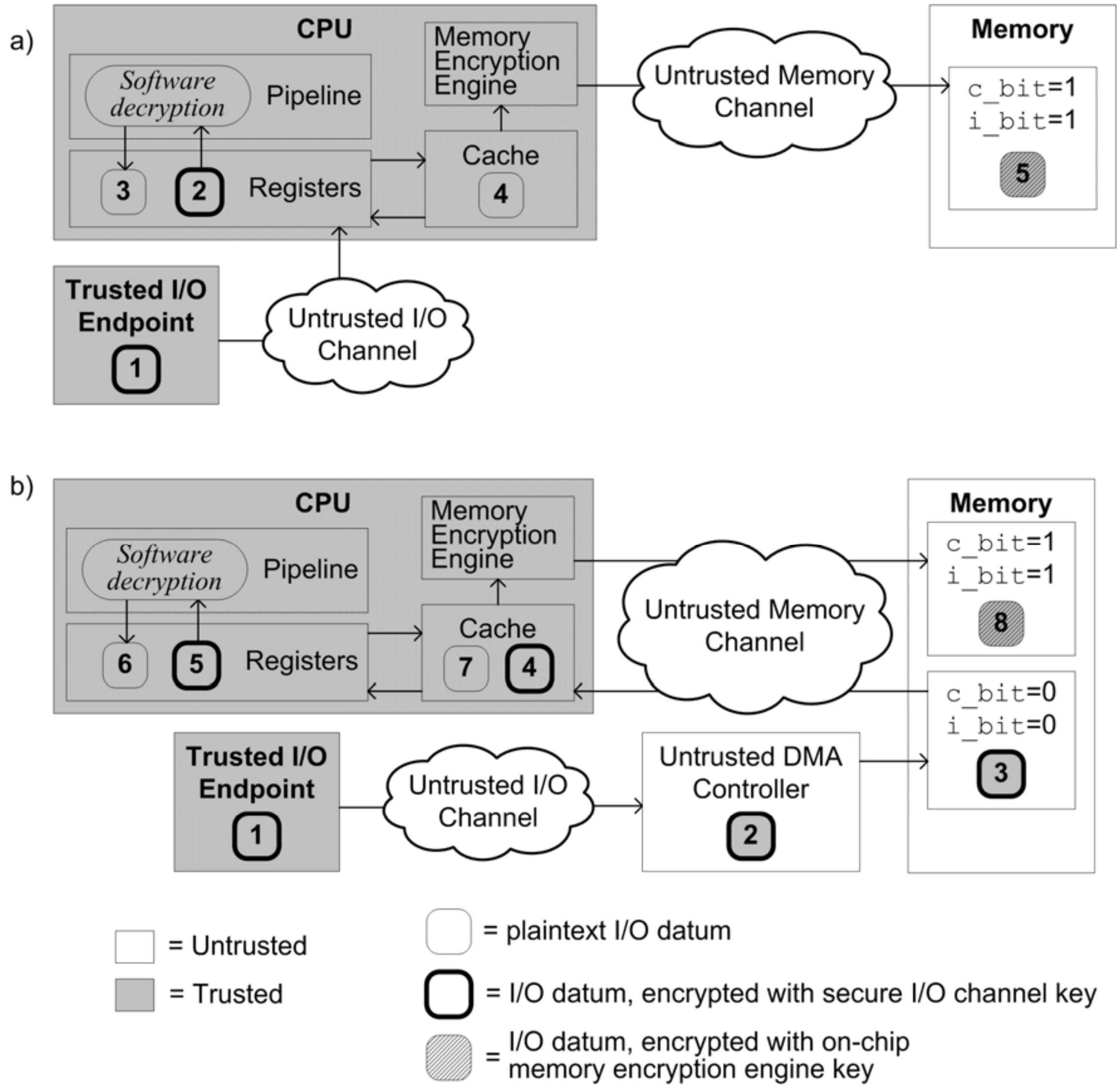


Figure 6.2. Reading encrypted I/O data into a Bastion-protected memory page (`c_bit`=1), a simplified view (no depiction of the integrity verification mechanism). An I/O datum is numbered sequentially as it progresses through the data path. We show a typical data path for memory-mapped I/O (or dedicated I/O instructions) in (a), and for a DMA transaction in (b)

[24] Background on basic I/O mechanisms is provided in the appendix to Chapter 6, in Section 6.B

processor can read encrypted data from the secure I/O channel into the processor registers (data numbered 1=>2). The module can then decrypt the data using the secure channel key it shares with the trusted I/O endpoint (data numbered (2=>3). At this point the module can also verify the integrity of the I/O data using a secure I/O channel key, or otherwise validate the data received from the I/O endpoint (not depicted). Finally, the module writes the I/O data to its encrypted page (also integrity-protected), via the cache subsystem (data numbered 3=>4=>5). Memory buffers containing I/O data can then be located within encrypted and integrity-verified module pages (i.e., with set i_bit and c_bit), since I/O data is read and written by the module itself.

Difficulties arise with DMA transactions (with devices other than the disk, which is never a trusted I/O endpoint), where I/O data in main memory is read or written directly by untrusted hardware, without going through the processor chip. In this case, I/O data buffers in main memory cannot be located within pages that have their i_bit or c_bit set. The reason for this restriction is that data read and written directly from and to memory is not decrypted or encrypted by the on-chip memory engines. In addition, direct writes to machine memory do not trigger integrity tree updates by the on-chip engine. Without the proper cryptographic processing, the data in these pages becomes unusable. The trusted I/O endpoint is unable to understand the data it receives from a page with a set c_bit, since they are encrypted with the processor's memory encryption key, which is only available to the on-chip memory encryption engines. Data written via DMA to I/O buffers with an i_bit set trigger an integrity verification error when read by a module since the current valid state of the integrity tree does not reflect the data written to memory by the DMA controller.

In Bastion, the only memory pages that can be written by untrusted entities without triggering integrity verification errors are pages tagged with an i_bit set to zero. Similarly, the only pages that are not obfuscated by encryption and thus can be read by untrusted entities are the pages with a c_bit set to zero. Software modules using DMA must thus allocate I/O buffers in pages that have both bits set to zero. As illustrated in Figure 6.2-b, the DMA controller can write to these pages the I/O data it relays from a trusted I/O endpoint, encrypted and MAC'ed for the secure channel (data numbered 1=>2=>3). Modules then read this data via the cache (3=>4)[25], into processor registers (4=>5), where secure channel decryption and verification can take place (5=>6). The decrypted data is then written to the module's encrypted page via the cache (6=>7=>8). As it is written off-chip, the memory integrity engine (not depicted) updates the current state of the integrity tree to reflect the page's new contents.

In the reverse direction (not depicted), module data to be written to the secure channel is first encrypted and MAC'ed within a protected page, where the data and operations are shielded from snooping or corruption by outside entities, including the DMA controller. The encrypted and MAC'ed data is then copied to an unprotected page, where they can be read by the untrusted DMA controller as part of the DMA transaction.

[25] No integrity tree verification takes place at this stage, since the page has its i_bit set to zero.

6.3 Bastion Secure I/O Services

This section presents the Tailored Attestation and Secure Persistent Storage mechanisms, two services needed for secure input and output for software modules protected by Bastion. In Section 6.4, we detail how Bastion-protected software can apply these primitives to secure user, network, disk and other forms of I/O.

6.3.1 Tailored Attestation

In this section we first provide some background on the concept of attestation and its implementation as a mechanism in existing commercial and research platforms. We then describe the design and implementation of the Bastion Tailored Attestation mechanism for the hypervisor and VM-based software modules.

Background on Attestation: Attestation is the reporting by a computer of the software it is running to a remote party. The remote party must then make a *trust decision*, to determine whether the reported software can be trusted to carry out a given security-critical computation. The trust decision is based on whether the reported software can be trusted to 1) protect the data inputs it is provided with, 2) carry out the security-critical computation correctly, within a protected execution environment and 3) protect the results of the computation, which may be stored locally, passed on to another piece of local software or reported back to the remote party. An attestation mechanism must thus not only report the identity of the software it is running, but it must also report on the protection properties provided to this software by the platform. In addition, the mechanism should also enable the establishment of a secure channel between the remote party and the reported software running on the attesting platform. This channel will ensure the confidentiality and integrity of any data input or output exchanged between the two parties.

The most widely deployed attestation mechanism is the one provided by the Trusted Platform Module (TPM) chip [TCG 2006]. To simplify this discussion, we will take as an example the TPM implementation targeted for Personal Computers (PCs) and assume the computer only has a single processor. The TPM, a discrete chip separate from the microprocessor chip, can store the measurements of local software running on the processor. The TPM itself does not measure local software. During bootstrap of the processor, each layer of software that runs measures the identity of the next layer and asks the TPM to store this measurement in its dedicated registers. Therefore, the very first layer of software to run, often called the BIOS boot block, is not measured. In the TPM threat model, this piece of software is assumed to be trustworthy and immutable. During runtime, local software can ask the TPM for an attestation report, to satisfy an attestation request by a remote party. The TPM creates the attestation report by signing, with its private key, a message containing stored measurements. This signed report is returned to local software, which can then forward it to the remote party (along with a platform certificate binding the TPM signing keys to the platform and certifying the TPM chip was correctly mounted). All interaction between local software and the TPM chip is done through standard I/O mechanisms, reading and writing data on the Low-Pin Count (LPC) bus where the TPM chip is attached.

A major problem with the vanilla TPM approach described above is that an entire software stack must be reported in order for a remote party to make a trust decision about an application. This typically includes the BIOS, the OS loader and the entire operating system code base, in addition to the code base of the application of interest. Hence a remote party must not only determine whether the application can carry out a given security-critical computation, but it must also determine whether this software stack can be trusted to protect the application. To make this determination, the remote party can either inspect the source code of the reported software stack or check the reported measurements against a database of measurements known to correspond to trustworthy software. The latter approach can be seen as equivalent to an inspection of the source code since some entity has to inspect the code before it vouches for it in the database. Unfortunately, inspecting the millions of lines of code forming an operating system is a daunting task in practice, as evidenced by the high number of exploitable vulnerabilities found in most software systems, even after thorough verification by their developers. Therefore, the security guarantees provided by the basic TPM attestation mechanism are weak. In practice, the remote party is unable to determine with certainty whether a security-critical application can run uncorrupted on the attesting platform.

Given the difficulties associated with inspecting a full-blown commodity software stack, an attestation mechanism should try to report only the software that directly participates in the completion of a security-critical task. We call this capability *functional attestation*. To allow a remote party to make a trust decision despite partial software stack information, the platform must guarantee that reported software executes in isolation from surrounding untrusted software. Therefore, functional attestation not only requires measuring and reporting individual software components, but it also requires mechanisms isolating this software from the unreported part of the software stack. Some proposals in the literature achieve a form of functional attestation by combining virtualization technology with the TPM. A trusted hypervisor loads a commodity software stack (including a full-blown operating system) in one virtual machine, and security-critical software in a different virtual machine, running a smaller, trusted OS [Garfinkel et al. 2003]. The strict isolation guarantees provided by virtualization ensure that untrusted software cannot interfere with the software in the trusted VM. Attestation on these platforms excludes the software in the commodity VM to report only the identity of the BIOS, the boot-time loader software, the hypervisor and the software stack in the trusted VM.

Technologies such as Intel's Trusted eXecution Tecnology (TXT) [Intel 2007] can also be leveraged to establish what TPM calls a *Dynamic Root of Trust for Measurement (DRTM)*, to further reduce the amount of software reported on attestation. Although DRTM technology was initially designed to allow for a trusted hypervisor to be "slipped" dynamically under an already running untrusted OS, it can also be used at boot-time to remove the need for reporting BIOS and boot-time loader software during attestation. The DRTM is a trusted routine written and digitally signed by the processor manufacturer. It can be invoked using a new processor instruction, GETSEC[SENTER] on Intel [Intel 2009]. The DRTM is authenticated and executed within the processor chip—i.e. no state is evicted to external memory and most processor interrupts are disabled. It verifies the work done by software that ran prior to DRTM invocation (e.g., the BIOS and the boot-time loader), and it sets up a secure environment for a trusted hypervisor. The DRTM routine also measures the identity of the loaded hypervisor and stores it in the TPM chip. Attestation reports from a platform using the DRTM can thus be reduced to identifying the trusted hypervisor and the software running in a trusted VM, completely ignoring untrusted VMs and software that ran prior to DRTM invocation. This is a step closer to functional attestation, but it still requires attesting to the whole software stack in the trusted VM, including an operating system.

The Flicker security system [McCune et al. 2008] takes a different approach and uses DRTM technology to establish a minimalist runtime environment for a critical *Piece of Application Logic* (PAL). Hardware virtualization support is used to isolate the PAL's execution environment from the untrusted software stack from which it was invoked. This removes the need for a trusted hypervisor and a trusted VM, and allows Flicker to report merely to the trusted PAL and its runtime. When each PAL maps to a security-critical function, this approach is a near-ideal embodiment of the functional attestation concept, as only the identity of a PAL and its runtime support routines are reported. However, due to the absence of a trusted context manager such as a hypervisor, the Flicker approach requires the DRTM to be invoked every time CPU control is to switch from untrusted software to the PAL. This can lead to high overheads, on the order of hundreds of milliseconds [McCune et al. 2008], due to the slow speed of TPM operations.

Another research proposal aiming for functional attestation is property-based attestation [Sadeghi and Stüble 2004], where the platform reports properties rather than software measurements. A major thrust of this research is to protect the privacy of the user by ensuring that remote parties receiving an attestation report are unable to determine exactly what software configuration the user's platform is running. The attesting platform needs to run special software that can infer properties from the running software stack and get the TPM to sign a report committing to these properties. This report thus confirms a number of statements about the platform's capabilities rather than the traditional bit identity of software components. The recipient of such an attestation report then makes a trust decision based on whether the attested properties are sufficient to ensure the protected execution of a given security-critical task. Although it may simplify the making of the trust decision in itself, property-based attestation still requires a trusted third party to inspect a full software stack to confirm the reported properties.

PRIMA [Jaeger et al. 2006] adopts an approach similar to property-based attestation to reduce the amount of software reported by an attesting platform. A PRIMA agent running on the platform analyzes the software stack based on a formal integrity model to derive information flow properties. Based on these properties, the PRIMA agent can trim certain components from attestation reports that are judged to be isolated from the software of interest. However, the approach still requires reporting the identity of the PRIMA agent and the entire SELinux operating system on which it is based. A similar approach, also based on SELinux, was developed specifically to report the enforcement of a simplified version of the Clark-Wilson integrity policy [Shankar et al. 2006].

Having the TPM chip as a discrete chip separate from the processor not only slows down security operations, it also prevents the TPM from gaining any visibility into the software context. This makes it impossible for the TPM to detect whether software measured at launch is being changed during runtime. As a result, approaches that rely on the TPM are vulnerable to Time Of Check to Time Of Use (TOCTOU) attacks. In such an attack, a legitimate software stack is loaded and measured into the TPM at boot time, but corrupted during runtime—e.g., by a hardware adversary modifying software state in memory. If a remote party asks the TPM to report the state of the software stack, the TPM attests to the load time state of software, not to its current corrupted state. The trust decision made by recipients of TPM attestation reports may thus be wrong. This lack of a *runtime attestation* capability makes TPM unsuitable for scenarios where physical attacks are considered in the threat model. Hardware-based security architectures like AEGIS and SP [Dwoskin and Lee 2007] can offer a form of runtime attestation, as they have the ability to detect runtime corruption of measured software. However, SP can only handle one piece of trusted software at any given time and AEGIS requires modification to the operating system kernel to support attestation. In addition, it does not support functional attestation as it systematically reports all OS modules supporting at least one critical application rather than only those depended upon by the application of interest.

The Bastion attestation mechanism offers what we call *Tailored Attestation*, a functional, runtime and resilient form of attestation. Bastion attestation strikes a compromise between the minimalist, but high overhead TCB of Flicker and the generalist approach of traditional attestation, where vast amounts of software need to be reported. Bastion attestation is functional, as it can report the identity of the hypervisor alone, or combined with the identity of a subset of Bastion-protected modules. Bastion memory integrity protection mechanisms enable runtime attestation to prevent TOCTOU attacks. Similarly to PRIMA [Jaeger et al. 2006], Bastion attestation enables resiliency in attestation, as attesting to critical functions will not be prevented by compromises in unrelated parts of the system. We first describe the `attest` instruction, Bastion's hypervisor attestation primitive. The hypervisor is the software component of the Bastion Trusted Computing Base (TCB) so it must be reported on every attestation. Then we show how the hypervisor leverages the `attest` instruction to produce attestation reports for the Bastion-protected modules in a trust domain. In Section 6.4, we show how these mechanisms can be used to provide secure network and user I/O.

Bastion Hypervisor Attestation: As in TPM, there are two roots of trust in the Bastion hypervisor attestation mechanism. The root of trust for measurement, responsible for measuring the identity of the hypervisor, is the Secure Launch routine, invoked via the `secure_launch` instruction. The measured identity is stored in the hypervisor identity register (*hv_identity*), on the processor chip. The root of trust for reporting, responsible for compiling and digitally signing attestation messages, is the attestation routine invoked by the hypervisor, via the new `attest` instruction. The message to sign consists in a concatenation of the hypervisor's identity, read from the *hv_identity* register, and a datum called a *hypervisor certificate*, passed to the attestation routine via the attest instruction's *certificate pointer* operand. The digital signature is computed in software by the attestation routine, which runs in a protected on-chip environment, mapped to a dedicated range of machine memory. The routine compiles the attestation message and signs it with the processor's private key, read from the CPU Private Key register (*cpu_key*) using the `bastion_read` instruction. The attestation routine then returns the signed attestation report to the software that invoked `attest`. It does so by writing memory via the *report pointer* operand to the `attest` instruction.

The `attest` instruction can only be invoked by the hypervisor software itself. As it executes `attest`, the processor saves the address, within hypervisor code, of the `attest` instruction into a general-purpose register to enable return back to hypervisor code. The on-chip attestation routine can then jump back to the hypervisor instruction following `attest` when it completes execution. Since the hypervisor executes out of machine memory, both pointer operands to `attest` refer to addresses within the machine memory space. *hv_identity* is initially zeroed on processor reset. If the `secure_launch` instruction is not invoked, or if its invocation fails, this register remains filled with zeroes. The `attest` instruction carries out the Bastion attestation procedure even if the register is zeroed, as it allows the platform to report that it is not running a securely launched hypervisor. A Bastion platform thus only produces non-nil attestation reports when a hypervisor has been securely launched and is running in its protected memory space.

The certificate passed to the `attest` instruction is used by the hypervisor to bind a piece of data to the attestation report. The certificate pointer can either reference the data directly or it can reference a cryptographic hash computed by the hypervisor over the data. The attestation routine does not need to know whether the data referenced by the certificate pointer is a hash or not. It simply reads a fixed amount (corresponding to the width of a cryptographic hash) of data from the referenced address, concatenates it with the hypervisor identity hash and then computes a digital signature over the concatenated items. The resulting attestation message effectively binds the processor (CPU private key), the specific hypervisor (hypervisor identity) and the data generated by the hypervisor (the hypervisor certificate). The certificate can thus be used for a variety of purposes, e.g., ensuring message freshness, binding a computation result to a specific hypervisor or binding a public key to a specific hypervisor.

To ensure freshness during runs of an attestation protocol, the requestor of the attestation report must send an unpredictable Number used ONCE (NONCE) to the platform, which the hypervisor includes in the computation of the certificate it passes to `attest`. The attestation routine then includes that nonce in the signature, which provides the remote party with a guarantee that the attestation report was freshly generated. The certificate can also be used to provide a guarantee that a given computation result was produced by the current hypervisor. This is necessary to prevent certain TOCTOU attacks. For example, the platform may boot hypervisor X, attest to hypervisor X and then reboot with hypervisor Y. A remote party receiving a computation result from hypervisor Y might then think the result actually comes from X, which just replied to an attestation request. This form of impersonation is possible because of the delay between the check of the hypervisor's identity and the production of a computation result. To avoid this TOCTOU vulnerability, the remote party can ask the hypervisor to deliver its result as part of an attestation message, including it in the hypervisor certificate.

In scenarios where the platform frequently sends data to the remote party, having to generate an attestation report for each datum may lead to unacceptable overheads in processing time. To avoid these overheads, the hypervisor can use the Bastion attestation mechanism once to exchange an authentication key with the remote party, and use this key to authenticate future communications. To do so, the hypervisor first needs to generate a public/private key pair and provide the public part of the key as the certificate for the `attest` instruction. The remote party then uses this public key to encrypt an authentication key and sends the encrypted key to the attesting platform. The platform decrypts the authentication key with the private part of the key pair and can use it to compute Message Authentication Codes (MACs) on future messages sent to the remote party.

The initial attestation report effectively certifies that the reported hypervisor executing on a specific Bastion processor has generated a public/private key pair whose public component is bound to the attestation message. The remote party can be confident that the private part of the key, hence the authentication key as well, is secure because the Bastion processor only attests to hypervisors that are running in a protected memory space (where both keys are stored). If the platform were to reboot, all secrets within the protected hypervisor memory space would be lost (because the memory encryption key changes on a reboot). Therefore, data authenticated with the MAC key can only come from the hypervisor that was identified during attestation. Using the MAC key thus allows the hypervisor to bind computation results to its identity without having to invoke the `attest` instruction every time it communicates with the remote party. The public/private key pair can also be used to set up a symmetric encryption key allowing for bulk encryption of messages exchanged between the hypervisor and the remote party.

Tailored Attestation: Our Tailored Attestation procedure is depicted in Figure 6.3. Software modules protected by Bastion attest to their identity using the new ATTEST hypercall. As for the `attest` instruction, the ATTEST hypercall takes a *certificate pointer* argument and a *report pointer* argument. The former points to a piece of module-specific data to be bound to the attestation report (called the *Module Certificate*), while the latter points to the memory region where the hypervisor is to write back the newly created attestation report (not depicted). The hypervisor handles the ATTEST hypercall by first compiling a report binding the identities of modules in the requesting module's trust domain with the module certificate passed as an argument. In Bastion, a module's identity includes its module hash and its security segment hash, reflecting not only the module's initial data and code state, but also the protection and sharing interfaces provided to the module during runtime. The report compiled by the hypervisor is thus a cryptographic hash computed over the concatenation of the certificate and the module and security segment hashes of every module identified in the trust domain descriptor. The hypervisor then invokes the `attest` instruction, with the report as part of the certificate passed as an operand. In addition to the report, the hypervisor can include a nonce, or any other type of data in the certificate.

Modules can use the Module Certificate just like the hypervisor's `attest` certificate: to introduce freshness in the attestation report or to bind a key or a computation result to the attestation report. The attestation report returned by the ATTEST hypercall binds modules not only to the specific Bastion processor they are running on, but also to the identity of the hypervisor running on the platform. The attestation report R and the session's public encryption key EK are returned (not depicted) to the requester, along with metadata identifying the processor (i.e., the processor's signature verification key), the hypervisor (HV, the hypervisor's identity), and the modules (their full trust domain descriptor).

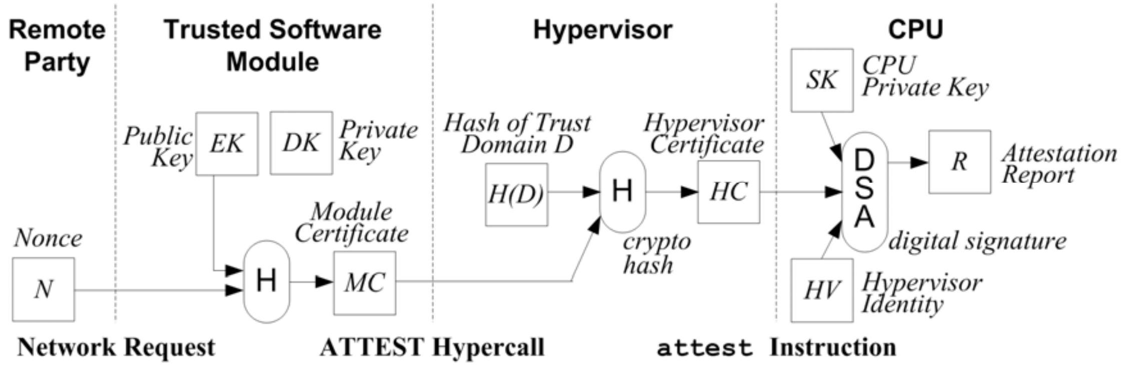


Figure 6.3. The Tailored Attestation Procedure

As for all Bastion security functions offered to modules, the module attestation mechanism cannot be trusted if the underlying hypervisor is untrusted. When it determines the hypervisor is trusted, the recipient of an attestation report can be assured that the module runs in a protected memory compartment. This is turn guarantees that the private component of a public/private key pair certified by module attestation is accessible only to the module itself. As for hypervisor attestation, this key pair can be used to establish encryption and MAC keys to protect the integrity and confidentiality of messages exchanged between the module and a remote party.

6.3.2 Secure Persistent Storage

The Bastion processor provides the hypervisor with its own *secure persistent storage* area, rooted in three new hardware registers: the Secure Storage Hash register (*storage_hash*), the Secure Storage Key register (*storage_key*) and the Secure Storage Owner register (*storage_owner*). These three registers are non-volatile, as they must protect the secure storage area across processor resets. As in SP's authority mode [Dwoskin and Lee 2007], the storage hash register contains a cryptographic hash computed over the latest version of the hypervisor's secure storage area, which is encrypted with the symmetric key stored in the storage key register. To avoid SP's requirement for physical presence of an authority to initialize these registers, Bastion allows any hypervisor to independently create its own secure storage area and root it in hardware. The storage owner register keeps track of the identity of the hypervisor that created the area currently protected by hardware. During execution of the `secure_launch` instruction, the processor only unlocks the current secure storage area if the hypervisor being loaded is the one that created the area. Otherwise, the loaded hypervisor is barred from reading the registers: it can only write them to establish a new secure storage area bound to its identity.

The Bastion hypervisor leverages its hardware-protected secure storage to provide an arbitrary number of protected software modules with their individual secure persistent storage areas (see Figure 6.4). For each module requesting a secure storage area via hypercalls, the hypervisor creates the same three storage anchors that were created by the processor for the hypervisor's secure storage area: a module storage hash, a module storage key and a storage owner. Rather than storing these anchors in dedicated hardware registers on the trusted processor chip, the hypervisor stores them in its own storage area, protected by the processor hardware. The processor hardware—the only trusted component on a platform reset—thus protects the hypervisor's secure storage area, which in turn protects the storage areas of software modules.

This chain of trust enables scalability of the Bastion secure persistent storage mechanism since the three hardware registers can protect a hypervisor storage area of an arbitrary size, which in turn can protect an arbitrary number of module storage areas. This concept is depicted in Figure 6.4. With module secure storage, the hypervisor SECURE_LAUNCH hypercall (not the hardware `secure_launch` instruction) is responsible for checking whether the module being loaded has in the past created a secure storage area. If so, the module can request its storage key and hash via hypercalls. If not, the module can only request a new, empty storage area from the hypervisor.

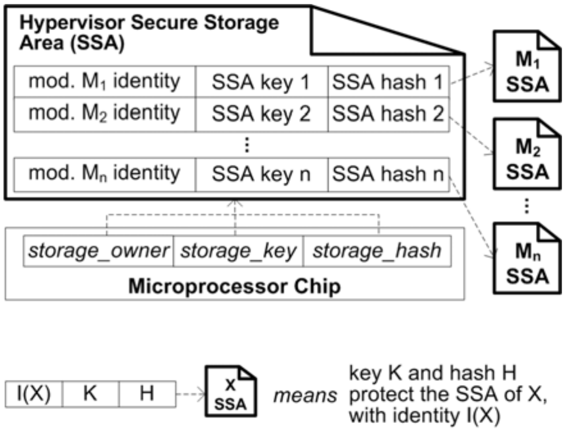


Figure 6.4. Bastion Secure Hypervisor and Module Storage

Hypervisor Secure Storage: The Bastion processor is manufactured and delivered without a hypervisor secure storage area. On the first processor reset, *storage_hash*, *storage_key* and *storage_owner* are filled with zeroes. To create a new secure storage area for itself, a hypervisor must first invoke the `secure_launch` instruction, to be measured by the processor and loaded into a protected execution environment. As part of `secure_launch`, the processor detects that no storage area exists and so it sets the volatile *storage_lock* bit to 1 to indicate it cannot be accessed, as in SP authority mode [Dwoskin and Lee 2007]. It also stores the hypervisor's computed identity in the dedicated *hv_identity* register. Within its protected memory space, the hypervisor can then allocate and fill up a data structure containing the initial *secrets* to be stored in the secure storage area. Using the on-chip hardware random number generator (RNG), the hypervisor then generates a symmetric encryption key K_{HV} that it uses to encrypt the contents of the data structure. It then computes a cryptographic hash H_{HV} over the encrypted data structure.

To create a hardware-anchored secure storage area, the hypervisor must write H_{HV} and K_{HV} to *storage_hash* and *storage_key* respectively. These register writes are carried out using the new `bastion_write` instruction. The processor interprets the first write to either register as a request for the creation of a new secure storage area by the current hypervisor. To satisfy this request, it first wipes out any secure storage area that previously existed by zeroing out the *storage_hash*, *storage_key* and *storage_owner* registers. It then binds the current hypervisor to the storage hash and key registers by copying the contents of *hv_identity* into *storage_owner*. Finally, it carries out the write requested by the `bastion_write` instruction and sets the *storage_lock* bit to zero to allow the hypervisor read access to its storage key and hash.

Once H_{HV} and K_{HV} are securely stored in the processor, the hypervisor can write the ciphertext of its data structure to untrusted persistent storage. This can consist of a disk, a disk array, a flash memory chip or any other form of persistent storage. The channel used to reach the untrusted storage medium can be untrusted as well since the confidentiality and tamper-evidence of the hypervisor secrets are maintained by the hardware-protected H_{HV} and K_{HV}. The hypervisor can thus write its data structure to disk or flash using I/O instructions outputting data on untrusted buses. It can also use a virtual machine-based helper program that has access to an OS disk device driver. This allows the hypervisor to store its secrets on a disk device without having to host possibly complex disk device drivers within its code base. In this case, the hypervisor maps its encrypted ciphertext to a machine page that is also mapped within the virtual address space of the helper program. In all cases, the encrypted hypervisor secrets written to storage form the hypervisor's secure storage area. This area could consist of a special partition of the storage medium where raw ciphertext bytes are written without any special format (other than respecting an endianness convention). The area could also be a chunk of data within a file respecting the conventions of the file system where it is written.

As it executes, the hypervisor may need to modify the data in its storage area or generate more data. In both cases, it carries out a procedure similar to the one initializing the area. The hypervisor loads its current area in a memory data structure, modifies or extends it, and then encrypts and hashes it as was done initially. This time, however, the key used for encryption is read from the *storage_key* register using the `bastion_read` instruction, rather than generated anew. Committing the new state of the area to the processor thus requires only a single register write to *storage_hash* using `bastion_write`. If the secure storage area gets too large to be conveniently loaded in its entirety to hypervisor memory, the hypervisor can use a hash tree scheme rather than a single hash to protect the content of the area. In this case, the value written to *storage_hash* would be the root of the hash tree covering the area. As for the memory integrity tree, this scheme allows the hypervisor to fingerprint changes to its secure storage area without having to fetch the entire area. Such a scheme has been proposed for SP authority-mode [Dwoskin and Lee 2007].

Upon processor reset, uncorrupted hypervisors automatically gain access to their secure storage areas. During `secure_launch`, the processor unlocks the *storage_hash* and *storage_key* registers for reading only if the identity of the loaded hypervisor, measured into the *hv_identity* register, is the same as the hypervisor identity specified in *storage_owner*. When the identities match, the processor sets the *storage_lock* bit to 0 to allow the loaded hypervisor full access to the secure storage key and hash register. The hypervisor can then fetch its secure storage area from untrusted storage, verify its integrity using the hash in *storage_hash* and decrypt it using the key in *storage_key*.

When there is a mismatch, the processor sets the *storage_lock* bit to 1 to prevent read access to these registers. In that case, the hypervisor can only create a new, empty secure storage area using the same procedure as was used to establish the very first such area. Creating a new secure storage area wipes out the hash and key of any storage area previously rooted in hardware. When no storage backup or migration mechanisms are in place, the data in the previous hypervisor secure storage area is unrecoverable since the key needed to decrypt it was lost. Ways to achieve secure storage backup and migration in the Bastion architecture are discussed in Section 6.5.

Module Secure Storage: As part of its handling of the SECURE_LAUNCH hypercall, the hypervisor checks whether the module being launched has created a secure storage area in the past. To do so, the hypervisor compares the module identity computed during SECURE_LAUNCH against the list of module identities with secure storage areas, found in the hypervisor's own secure storage area. When there is a match, the hypervisor copies the appropriate module storage hash and key to the Module State Table entry of the module being launched. When no match is found, the module is assumed not to have a secure storage area so the hypervisor zeroes the storage hash and key fields of the Module State Table entry. Following the completion of SECURE_LAUNCH, a module associated to an area can read or write its hash and key using the READ_STORAGE_HASH, READ_STORAGE_KEY, WRITE_STORAGE_HASH and WRITE_STORAGE_KEY hypercalls.

A module without a secure storage area can create one using a procedure similar to that of hypervisor area creation. The module must first generate, within its protected memory space, a data structure with the area's contents. Then it must invoke the hardware RNG to generate a symmetric key with which it then encrypts the data structure. The module also computes a cryptographic hash over the ciphertext. Finally, it commits its new secure storage area to the hypervisor by writing the hash and key using, respectively, the WRITE_STORAGE_HASH and WRITE_STORAGE_KEY hypercalls. Further updates to an existing area are committed using the WRITE_STORAGE_HASH hypercall. The hypervisor handles the hypercalls by writing the hash and key to the module's entry in the Module State Table.

Similarly to the hypervisor, modules write their encrypted storage area to a disk, a disk array, a flash memory or another persistent storage device. When a module runs in a virtual machine containing an operating system with storage device drivers, it can store its area and retrieve it using standard library functions invoking I/O system calls, e.g., *open*, *write*, *read*, *close*. In that case, the module's storage area is sent to storage as part of a file within a file system. It can also use other methods for carrying out I/O, even if channels are untrusted, since the area's data is protected by the hash and key securely stored within the hypervisor's memory space. On each invocation of WRITE_STORAGE_HASH by a module, the hypervisor commits the new state of the module's secure storage area by updating the module's entry in its own hardware-rooted area. The hypervisor periodically commits these updates to the processor hardware by writing the Secure Storage Hash register. When the module retires using the SECURE_RETIRE hypercall, the hypervisor wipes out its storage key and hash from the Module State Table. This ensures that the storage area remains inaccessible until the module is launched again in a future invocation of the SECURE_LAUNCH hypercall.

6.4 Usage Scenarios

In this section, we describe how the Bastion Tailored Attestation and Secure Persistent Storage primitives can be used to provide secure disk, network and user I/O. We also discuss other forms of local I/O and the impact of virtualized devices.

6.4.1 Disk I/O

Encrypted Files: Securing disk I/O on a Bastion platform can be a straightforward application of the secure storage primitive. In this case, every protected module has its own encrypted and hashed secure storage area in a file, which can be requested via the untrusted I/O mechanisms of the operating system.

Encrypted File Systems: The Bastion secure storage primitive can also enable more elaborate secure disk I/O schemes, where the entire file system is encrypted. In such a scenario, the file system manager, either in OS or application space, can be defined as a Bastion-protected module. This module's secure storage area can contain a file system key, a file system hash and critical file system metadata. All files in the file system are then encrypted either directly by the file system key or indirectly, by a key derived from the file system key. The current valid state of the file system is fingerprinted with a hierarchical hashing structure similar to a memory integrity tree, with its top hash as the file system hash. When the file system module is loaded via SECURE_LAUNCH in a protected memory space, the file system key, hash and metadata become available via the Bastion secure storage primitive. All file system requests by application and OS software can then be routed through the Bastion-protected file system module to be authenticated and decrypted.

Oblivious Disk Data Transfers: Under certain threat models, the software making secure disk I/O requests may also want to obfuscate the disk request patterns in addition to the disk data. In this case, a trusted file system module must not only maintain a file system key and hash to protect the data, but also establish a secure communication channel with the hardware disk controller. The cylinder, head and sector for a data request can then be sent encrypted and hashed over that secure channel such that they are unavailable to an attacker snooping on I/O buses. This requires a trusted hardware disk controller that can authenticate itself to and establish a secure channel with software running on the platform.

The design of such a controller is outside the scope of this thesis. We simply note that a trusted file system module could be distributed with the public keys of disk controllers known to be good embedded in its data space. Using public key exchange protocols, this could be leveraged to set up key materials between the file system module and the hardware controller. These key materials could be stored in the module's secure storage area as file system metadata, so the channel can be reestablished across reboots without the need for authenticating the controller every time. More work on the subject of storage request obfuscation can be found in the literature on oblivious data transfers, e.g. [Goldreich 1987].

6.4.2 Network I/O

Secure network I/O may require any combination of the following security functions: *inbound authentication*, *outbound authentication*, message integrity protection and message confidentiality protection. The last two functions can be achieved by establishing a secure communication channel between the remote network party and local software.

Outbound authentication—a term initially coined for the predecessor of TPM, the IBM 4758 secure crypto-coprocessor [Dyer et al. 2001]—can be achieved with a direct application of the Bastion attestation primitive, where local software attests to its identity for a remote party. As described in Section 6.3.2, this attestation primitive also allows local software to establish a secure communication channel with the remote party. Bastion attestation thus takes care of both outbound authentication and secure channel establishment. As explained next, the other security function involved in secure network I/O—inbound authentication—can be achieved with a combination of the Bastion secure storage and attestation primitives.

Inbound authentication is the authentication of a remote network party by a piece of software running on the local platform. To secure inbound authentication, Bastion must enforce the binding between the identity of a piece of software and the inbound authentication keys it was assigned. This in turn ensures that the software will only trust the remote parties that can be successfully authenticated using these keys. The keys can either be assigned statically by the developer of the software, or dynamically, by a remote party interacting with the software. In the former case, the developer embeds the inbound authentication keys in the data space of the software, in its binary file. Since Bastion does not protect the confidentiality of software binaries, these keys should be public signature verifications keys. Exposure of such keys to untrusted parties during software distribution does not compromise inbound authentication. This approach was taken by most mainstream web browser applications, which are distributed with public keys from online parties such as Verisign (see [Verisign 2008]), Microsoft and others.

Inbound authentication keys could also be assigned dynamically, by parties wishing to ensure a given piece of software only interacts with a certain set of trusted remote parties. In this case, the remote party first identifies the software by asking it to perform outbound authentication. Then the inbound authentication keys can be transferred from the remote party to the local software via a secure communication channel. Finally, the local software can store these keys in its Bastion-protected storage area to ensure they are available across reboots. Because the communication channel and storage area are encrypted, the inbound authentication keys in this case do not have to be public. For example, they can be (secret) symmetric keys used to compute message authentication codes. For the sake of clarity, the rest of this section assumes public keys are used for inbound authentication.

Making a trust decision about a piece of software requires knowing the set of remote parties that are considered trusted by the software. This can be done by having the software report the set of public inbound authentication keys it uses. As stated in Chapter 2, we assume that any public key infrastructure needed to certify public keys is already in place. With static inbound authentication keys, the identity of the software itself reflects the set of keys embedded in the software's data space. The Bastion attestation mechanism thus automatically reports on the set of parties trusted by a piece of software assigned static inbound authentication keys. With dynamically assigned keys, the software has to explicitly report the set of inbound authentication keys it is using. This can be achieved using the Bastion attestation mechanism and including the keys as part of the certificate passed to the attestation instruction or hypercall. The keys can also be reported via a secure channel previously established with the remote party inquiring about the keys.

6.4.3 User I/O

As discussed in Section 6.2.1, secure user I/O requires a combination of any of the following three security functions: user-to-machine authentication, machine-to-user authentication and secure HID-to-CPU channel. Solving the problem of cross-authentication with the user is outside the scope of this thesis. Bastion focuses on solving one part of the problem: cross-authentication with the HID device. Achieving secure I/O in Bastion is thus very similar to achieving secure network I/O. The main difference is that instead of interacting with a trusted network entity, software interacts with a trusted HID device. We assume the HID device is a trusted I/O endpoint as defined in Chapter 2, which can authenticate itself to software via public key cryptography. It can also receive a software attestation report and determine whether to allow this software access to its secure input or output paths. Providing secure I/O to a piece of software is thus reduced to providing that software with the same security functions as in network I/O: outbound authentication, inbound authentication and secure channel establishment.

6.4.4 Other Local I/O

In addition to user, disk and network I/O, Bastion computers may also be equipped with other local I/O devices such as printers, data acquisition cards or sound cards. As for user

I/O devices, Bastion does not attempt to provide security for the analog signals going through these devices. If a local I/O device is a trusted I/O endpoint, Bastion primitives can enable cross-authentication and secure channels, to protect the digitized data exchanged between the device and software running on the platform. In most cases, however, such devices are likely to require some form of physical tamper-resistance if they are to operate within the Bastion threat model, where attackers can physically tamper with most components of the computer system. Most security policies that require integrity of analog signals captured by a computer probably cannot be enforced within the Bastion threat model. For example, it is hard to see how a computer monitoring temperatures and pressures within a nuclear power plant could protect the integrity of the analog signals it monitors if hardware adversaries can have physical access to the environment. Bastion can secure the processing, storage and reporting of data, but it cannot substitute for physical security measures such as armed guards and motion detectors for detecting or deterring intruders.

6.4.5 Virtualized I/O

Rather than assigning each device to a particular virtual machine, certain hypervisors may virtualize some of the I/O devices. The hypervisor is usually the only software entity that can communicate with a virtualized I/O device, via a special device driver located in hypervisor space. This driver is a trusted piece of software since it is part of the hypervisor code base. There are two alternatives to achieving secure I/O with such virtualized devices. If the device can support multiple simultaneous secure I/O channels, the hypervisor driver can simply relay to the device all commands sent by VM-based software. The device then takes care of establishing separate contexts for each piece of software it communicates with. (In the literature, e.g. [Saulsbury 2008], separate contexts are sometimes called *device functions*.) When the device can only have a single secure I/O context, the secure I/O channel must be established with the hypervisor device driver, who then virtualizes this channel for each VM. In this case, the hypervisor driver emulates the device to the VM-based software during cross-authentication and secure I/O channel establishment.

6.5 Migration, Backup and Software Updates

In this section, we present a solution to the secure storage migration problem, where the availability of a hypervisor’s secure storage area is maintained despite loading a new or updated hypervisor on the Bastion processor.

Bastion secure storage is tied to a specific hypervisor running on a specific processor. This means that if two hypervisors create secure storage areas on a given processor, the second area effectively destroys the first one. It also means that migrating a hypervisor and its secure storage from one Bastion processor to the next is not immediately feasible. The same restrictions apply to module secure storage areas, which are anchored in the hypervisor’s secure storage area. This section proposes a solution for secure storage backup and migration which does not require any extension to Bastion hardware. Our solution is similar to that proposed for migration in TPM, where a trusted remote migration authority can be designated to temporarily be in charge of TPM secrets.

6.5.1 Platform-to-Authority Transfer

In Bastion, a trusted migration authority responsible for moving the secure storage of a given piece of software can be designated by the hypervisor or by each individual software module. As for inbound authentication, the authority is identified by its public key, embedded in the data space of the software that might eventually request migration or backup. In both cases, the authority is temporarily in charge of a secure storage {hash, key, owner} triple, until the triple is reassigned to a platform, either the same (backup) or different (migration). Note that a migration can be done between two very similar pieces of software (e.g., two versions of the same software, separated by an incremental update) or two completely different pieces of software (e.g., a Sun Hypervisor and a VMware hypervisor).

The first step to backup or migration is to establish a secure channel with the migration authority. As in TPM, we assume this authority is a dedicated server reachable via the network. As in a secure network I/O operation, the hypervisor or software module requesting backup or migration performs inbound authentication on the authority. This authentication, which ensures the software is interacting with a trusted authority, is carried out with the migration authority key embedded in the program's initial data space. Using the Bastion attestation mechanism, the software then attests to its identity and establishes a secure communication channel with the authority.

On this channel, the software can then securely send the hash and key it wants to backup or migrate. The software then specifies to the authority whether it is requesting a backup or a migration. If it is requesting a backup, the authority labels the {hash, key} pair it just received with the identity of the requesting software, to create the {hash, key, owner} triple. If a migration is requested, the software sends along the identity of the software that is destined to receive the secure storage area being migrated. The authority then creates the {hash, key, owner} triple using the received identity. After the request has been placed to the authority, the {hash, key, owner} triple on the local platform can be wiped out safely, e.g., to load a different hypervisor creating its own secure storage area.

6.5.2 Authority-to-Platform Transfer

To recover a backed up secure storage area or gain access to a storage area being migrated, hypervisors and software modules must prove their identity to the migration authority, using the Bastion attestation mechanism. The software must also perform inbound authentication on the authority to make sure it receives the storage area it expects. The authority only releases a migrated or backed up storage area if the attesting software has the same identity as the owner identified in the {hash, key, owner} triple. If so, the authority sends the hash and key to the requestor via a secure channel established during attestation.

The requestor can then use Bastion instructions or hypercalls to anchor its new secure storage area in the Bastion TCB. This completes migration or recovery of the roots of a secure storage area. Bastion does not define what happens to the actual ciphertext data covered by these roots. In the case of a backup and subsequent recovery on the same platform, the ciphertext does not need to move from its persistent storage device. For migration across different platforms, the ciphertext could either be sent along with the storage roots to the migration authority, or it could be sent directly via an untrusted network channel between the two platforms.

6.6 Chapter Summary

In this chapter, we introduced two Bastion mechanisms—namely Tailored Attestation and Secure Persistent Storage—which are used as primitives in providing secure Input and Output (I/O) capabilities for protected execution compartments. We defined our secure storage and attestation primitives and detailed the architectural features and protocols they require. Finally, we present usage scenarios which leverage these two primitives to provide secure disk, network and user I/O. In the next chapter, we present an implementation of the Bastion TCB on the OpenSPARC project.

Appendix to Chapter 6

6.A Related Work in Secure I/O

Secure input and output is a broad problem that has been addressed within a variety of threat models. In this section, we present past work on several aspects of the secure I/O problem and specify the underlying threat model assumptions. Some proposed solutions treat input and output separately, but most are designed to secure both input and output with the same hardware and software. We divide approaches according to whether they provide secure network, disk or user I/O, although some proposed approaches address multiple aspects of the secure I/O problem.

6.A.1 Network I/O

Secure Sockets Layer (SSL): The Secure Sockets Layer (SSL) protocol [Freier et al. 1996] (and its successor the Transport Layer Security (TLS) protocol [Dierks 1999]) is widely deployed to secure network I/O in modern computers. It allows a local program and a remote computer to authenticate one another and establish a secure channel over an insecure network. The mutual authentication part of the protocol is carried out using asymmetric cryptography, and allows each party to authenticate itself to the other using its own public-private key pair. A public key exchange protocol then establishes shared secrets that can be used to set up an encrypted and tamper-evident communication channel between the two parties. The secure communication channel is guaranteed to be opened only to host X and Y, as long as each host's hardware and software are not compromised. If X has malicious software or hardware, however, it can leak its public-private key pair and the shared secret such that another host M is able to impersonate X in its communication with Y. This is due to the fact that in the SSL threat model, the network is untrusted but hosts are assumed to be good. And so, in Bastion (as in TPM described below), we provide an attestation primitive allowing hosts to describe their hardware and software configuration so their counterpart can analyze their trustworthiness.

Trusted Platform Module (TPM): TPM [TCG 2006] can increase the security of network I/O by making sure SSL keys are only available to software known to be good. It can also provide stronger assurance in authentication by allowing software to attest to its identity before a secure network channel is established. Tightening the SSL threat model, the TPM assumes that host software may be compromised by offline attacks on the disk or by installation of untrusted software. On a typical SSL setup, keys are stored in plaintext on the platform's disk or on a user token (e.g., a USB stick). A compromised software stack can thus obtain the keys by reading them from the disk or requesting them from an unwitting user.

A TPM chip can encrypt the SSL keys such that they can only be decrypted by this very same TPM chip, only if the known-good software stack is running. The ciphertext of SSL keys can then safely be stored on the disk or on an external token. Compromised software is unable to access the plaintext of the SSL keys since the TPM refuses to decrypt their ciphertext when compromised software is running. As a result, compromised software is unable to authenticate itself with the keys reserved for good software. In addition, its inability to read the plaintext of the SSL keys ensures it will not be able to leak the keys to other hosts to enable impersonation. TPM can also help networked hosts authenticate one another for the first time before public keys are exchanged. Indeed, the attestation capability of the TPM allows a host to report the identity of its software stack so that its counterpart can determine whether or not the host can be trusted to store its SSL keys securely. Secure network I/O in TPM still has security shortcomings, however, addressed by Bastion and other security hardware solutions, as described below.

Processor-Based Solutions: The problem with TPM-based protection of SSL (or other types of secure networking) keys is that they are bound to an entire software stack, typically including a large, complex and frequently updated operating system code base. It is hard to inspect such a code base in order to determine whether or not it contains software vulnerabilities that could potentially be exploited to undermine the security of the SSL keys. In practice, TPM-based systems are unable to guarantee that the "known-good" stack to which the SSL keys are bound is actually trustworthy. A solution to this problem is to bind critical key materials to a much smaller piece of software protected directly by enhanced processor hardware.

This is the approach of the SP processor [Lee et al. 2005], which can encrypt key materials such that they can only be decrypted by an SP processor running a small, easy to verify Trust Software Module (TSM). The processor makes decrypted keys available only to the TSM, which it runs in a hardware compartment. This compartment is protected against snooping and corruption from both hardware adversaries and malicious software in surrounding applications or in the underlying operating system. The security of SSL keys is improved as a result, since the keys are made available to a much smaller amount of software. Although the user-mode version of SP [Lee et al. 2005] cannot report the identity of its TSM to a networked party, the authority-mode version of SP [Dwoskin and Lee 2007] offers some TSM attestation capabilities to help establish initial key materials with remote parties over the network.

AEGIS [Suh 2005] is another processor-based security architecture which can attest to smaller pieces of software running in protected compartments. Its attestation protocol can be used to establish shared secrets, but AEGIS does not provide secure persistent storage to protect keys across platform reboots.

Virtualization-Based Solutions: Another way to reduce the amount of software to be trusted for protected network operations is to host security-critical networking software in a virtual machine separate from commodity operating systems running in other virtual machines. The idea is to run a network stack in a trusted VM running a minimal OS environment and give that VM exclusive control over the hardware Network Interface Card (NIC). Other VMs access the physical network through this trusted VM. As demonstrated in the Vault architecture [Kwan and Durfee 2007], critical SSL operations can be carried out in the trusted VM itself, protected from runtime attacks by commodity software. The SSL keys can also be protected from offline attacks while they sit on the disk by binding them to the trusted VM using TPM technology (e.g., as is done in the Terra architecture [Garfinkel et al. 2003]).

Bastion: The approach taken by Bastion to secure network I/O is similar to that of SP authority-mode [Dwoskin and Lee 2007], where a small hardware-protected software module can attest to its identity and establish a secure network channel. This channel remains secure despite the software module being surrounded by untrusted software, including an untrusted OS. Bastion improves on SP by making this approach more suitable to general-purpose computing platforms. A crucial difference is that, by introducing a private CPU key for attestation, Bastion devices can establish secure network channels with remote parties they have never met before, whereas SP requires a remote authority to be in physical possession of the SP device at some point in order to inject it with secret key materials.

In addition, Bastion scales up the SP model by making it possible for several modules in different trust domains to establish separate secure network channels with different remote parties. Limited SP hardware resources mean only one trust domain at a time can be provided with secure network I/O. Bastion also enables resiliency of I/O by ensuring that a protected software module’s secure network channel remains usable even during an attack on platform integrity; Bastion halts any software module for which it detects tampering, but TCB mechanisms continue protecting a module and its network channel as long as the hypervisor and the module itself are not affected by corruption.

6.A.2 Disk I/O

In this section, we use the generic term disk I/O to refer to any data exchange between software and the hardware disk device, either via raw accesses or formatted file system accesses.

Traditional File Systems: A basic form of disk I/O security consists in the permission bits associated to each file in most file systems. These bits typically (e.g., in Unix) specify an Access Control List (ACL) formed of three sets of Read, Write and eXecute (RWX) rights: for the user owning the file (by default, the creator), the group the user is a part of, and the rest of the world. During runtime, the file system manager, either an OS module or a user-space program, enforces that programs running on behalf of the current user be only allowed the access rights specified in the ACLs of the files they access. Although this method protects from malicious applications trying to access another program’s files, it does not provide any security when the operating system is corrupted or compromised. Even with a genuine OS, this permission scheme can be easily defeated by an offline attack, where the attacker accesses the disk directly, either by booting a different OS that bypasses the permission scheme or by gaining physical access to the disk.

Encrypted Disk: Disk encryption schemes have been proposed to defend against offline attacks on disk contents. Encryption can be performed by the disk controller itself (e.g., [Seagate 2006]) or by the file system software, e.g., [PGP 2009]. In both cases, the key used for encryption has to be kept off the disk to make it inaccessible to an offline attacker. A common way to protect the disk encryption key is to have the user provide it anew every time he logs onto the platform. This can be done by having the user carry the actual key around on a hardware storage device such as a USB key. Another possibility is to derive the key on-the-fly from a user password or passphrase. Although these methods keep the key away from an offline attacker, they still do not prevent a compromised OS from requesting the key from an unwitting user. As was the case for secure network I/O, this problem can be solved using the TPM chip to seal the disk encryption key to a known-good version of the software stack. This strategy has been adopted in both commercial [Microsoft 2009] and open source [Halcrow 2007] disk security software.

Hardware-Anchored Storage: The security mechanisms described above rely on the OS code base being trustworthy. As for network I/O, this assumption seldom holds. To circumvent this problem, SP authority-mode [Dwoskin and Lee 2007] anchors a piece of persistent secure storage in a pair of processor registers accessible only to a verified and runtime-protected Trusted Software Module (TSM). One of the registers contains a Storage Root Key (SRK) and the other contains a Storage Root Hash (SRH). To protect an arbitrary amount of long-lived secrets, a TSM can create an arbitrary size data structure in its protected memory space and write all the secrets to this structure. It can then encrypt the data structure using the SRK, compute a cryptographic hash over the ciphertext and store that hash (the SRH) in the dedicated processor register. The TSM can then flush the ciphertext, its *secure storage area*, to disk via the unprotected, untrusted I/O mechanisms offered by the operating system. The drawback of the SP approach is that a single secure storage area can exist at any point in time, corresponding to a single trust domain, due to a limitation in the number of dedicated processor registers. However, SP does not require a hypervisor as Bastion does.

Bastion: In providing secure disk I/O, Bastion scales up the SP approach to secure storage without increasing the number of required processor registers. A single pair of registers protects a secure storage area for the Bastion hypervisor, which in turn leverages this area to provide each protected software module with its own secure persistent storage area. The hypervisor ensures that modules cannot access one another’s area. As for network I/O, Bastion provides resiliency in secure disk I/O by allowing protected modules to use their secure storage area even when tampering has been detected in other Bastion modules.

6.A.3 User I/O

User I/O is arguably the hardest type of I/O to secure, as it involves interacting with a human user, often via analog means. We divide the secure user I/O problem into three components: 1) user-to-machine authentication, 2) machine-to-user authentication and 3) secure HID-to-CPU channel establishment.

User-to-Machine Authentication: Most computer systems authenticate the users they interact with. A well-known taxonomy of user authentication divides authentication factors into three classes: What You Know (WYK), What You Have (WYH) and What You Are (WYA). In WYK authentication, the user proves his or her identity by demonstrating knowledge of some information only he or she could know, e.g., a password, a passphrase or a social security number (*n.b.*, a provably weak authentication factor [Acquisti and Gross 2009]). In WYH authentication, the user is asked to prove his or her identity by providing a physical token that only he or she could possess, e.g. a door key, an uncloneable chip or badge or a private key. Finally, in WYA authentication, an intrinsic characteristic of the user is examined to determine his or her identity, e.g., fingerprint, eye retina patterns, gait, etc.

On computer systems, WYK authentication is usually carried out via the keyboard while WYH and WYA are carried out through specialized peripheral hardware devices. To improve security, certain systems adopt two-factor authentication and require the user to provide two different proofs of his or her identity. For example, in many smartcard-based systems, the user is only authenticated after presenting the physical smartcard (WYH) and an accompanying PIN or password (WYK). Two-factor authentication can also be carried out over a remote channel. For example, remote login systems enhanced with SecurID technology [RSA 2009] require the user to provide a password formed of two components: a user-memorized alphanumeric password (WYK) and a multi-digit number obtained from a special hardware token unique to the user (WYH). Because the number on the token changes constantly based on an unpredictable cryptographic function, the user or an adversary cannot memorize a given number to use it later. A successful remote login requires physical possession of the token to ensure the freshest number is used. Note that this technology has been shown to be vulnerable to cryptanalysis [Biryukov et al. 2003].

Regardless of the user-to-machine authentication mechanism used, however, the user should not provide any sensitive data via the HID devices until he or she has completed some form of machine-to-user authentication. Such authentication gives the user some assurance that he or she is not disclosing secret information to malicious hardware or software.

Machine-to-User Authentication: Machine-to-user authentication is the process trough which a user ascertains that he or she is communicating with a well-behaving (i.e., non malicious) hardware system running uncompromised software. Past work on machine-to-user authentication does not try to provide strong, mathematically provable guarantees about the correctness of hardware and software, but rather concentrates on thwarting the most likely threats within a set of scenarios under consideration. The most frequently used form of machine-to-user authentication is probably the Microsoft Windows login mechanism. Through the years, users have been trained to use a special escape sequence when logging into a platform running a Windows NT-based operating system. The well-known sequence of keys, *Control-Alt.-Delete*, is a *Secure Attention Key* (SAK) triggering the display of a dialog box where the users are asked for their login credentials. An SAK can be defined as a key or a combination of keys reserved for the invocation of a security mechanism—in this case, the login process in Windows NT. Users are expected to know that a login dialog box displayed without invocation of the SAK is probably a sign of a malicious piece of software trying to steal their login credentials. When this happens, the users should refrain from further interaction with the machine since it is likely running malicious software.

SAK-based login is far from being a silver bullet. When the operating system kernel itself is compromised, the software handling the SAK mechanism can be changed to display a malicious login dialog box upon invocation of the SAK. In this case, the login procedure appears perfectly genuine, even to a trained user, but the login software may leak the user's secret login information. Such a compromise of privileged software is not unlikely since the OS code base usually consists of millions of lines of code and that are likely to contain several hundred exploitable software vulnerabilities. To protect user credentials without having to rely on operating system correctness, some researchers propose linking the SAK with a trusted login process running in a virtual machine separate from the virtual machine containing the commodity operating system [Kwan and Durfee 2007, Borders and Prakash 2007]. In such a system, the hypervisor monitors user keyboard interaction so to intercept the SAK and route it to the login VM. In this approach, user login security depends on a much smaller code base—the combined code base of the login VM and the hypervisor versus the entire OS code base—hence it is more likely to be achieved.

SAKs are based on local keyboard hardware triggering an interrupt to be handled by the operating system kernel. This mechanism thus is not suitable for remote login procedures where the user himself, not his local machine, is responsible for authenticating the remote machine. To address this problem, remote servers can use a form of What You Have authentication towards the user. In a typical implementation, the remote server displays, as part of the user's login dialog box, an image belonging to or chosen by the user in a trusted registration phase. The user registers the image only once with a trusted server and in later login attempts, expects to see the image displayed as a proof that he or she is interacting with the trusted server, e.g., [Pering et al. 2003]. The idea is that a malicious server trying to impersonate the trusted server would be unable to produce the user's image and the user would refrain from logging in, protecting his or her credentials.

The methods discussed above do not allow for full-blown machine-to-user authentication since the user must trust at least some part of the local machine's software stack. To avoid trusting local software, many research efforts suggest having the user employ a small trusted computing device as a proxy to either authenticate or bypass the local machine. The architecture presented in [Garriss et al. 2007] enables *trustworthy kiosk computing* by having the user authenticate an untrusted local machine by using a trusted mobile device and the TPM chip on the local machine. Through the mobile device, the user requests a TPM attestation report from the local machine. The mobile device forwards this report to a trusted remote server that examines it to determine whether the local machine runs approved software. The user is notified of an approved software stack by the trusted mobile device. If the software is approved, the local machine is considered as trustworthy and the user interacts with it, i.e., the user entrusts it with sensitive information, and trusts its outputs. If the software is not approved, the user refrains from further interaction and simply walks away from the local machine.

A similar approach is taken in the BitE architecture [McCune et al. 2006], where the user's mobile device also requests an attestation report from the TPM chip attached to the local machine. BitE aims to establish separate keyboard input channels for each trusted application the user is interested in. The separate channels prevent applications from spying on one another's keyboard traffic. The OS and applications of interest are attested to separately, and the correctness of the software measurements reported by the TPM are verified by the mobile device itself. During runtime, sensitive data can be displayed on the trusted mobile device's display to avoid output spoofing by unverified applications.

Other approaches involving trusted proxy devices actually remove the need for machine-to-user authentication. They do so by partially bypassing the local machine in order to secure user inputs and outputs. Instead of having the local machine attest to the trusted mobile device, the approach in [Oprea et al. 2004] has the user interact with the local machine only for input and output operations that are not security-critical. The trusted mobile device is used for critical keyboard inputs and display outputs. The driving scenario behind this approach is a user trying to communicate with his or her home PC while at a public terminal. The HID devices of the untrusted terminal provide convenience with their large sizes and richness of features, while the mobile device provides security for sensitive operations. The architecture proposed in [Sharp et al. 2006] focuses on the confidentiality of outputs displayed to the user on a public terminal. In this case, the mobile device is used to automatically redact sensitive information on the public display and reroute them to the display of the trusted device, which is less susceptible to shoulder surfing attacks.

Secure HID-to-CPU Channel: Traditional computing platforms typically do not protect the data traffic between human interface devices and the CPU. This allows physical attackers and unauthorized software to snoop on or corrupt sensitive user keyboard inputs or confidential data sent to displays, for example. Computing platforms that do protect user I/O traffic do so by establishing an encrypted and integrity protected communication channel between an HID and the software running on the CPU. In BitE [McCune et al. 2006], the trusted computing device proxy secures separate user input channels to different applications by having the device exchange different keys with each application during attestation. The trusted mobile device is inserted between the keyboard and the machine of interest. The keyboard, equipped with special keystroke encryption hardware, protects user inputs on the part of the channel between itself and the trusted device. The device then re-encrypts the inputs with application-specific keys to protect the user inputs on the way from the device to the application of interest. The trusted device's display tells the user which application is currently receiving keystrokes to avoid having the user being tricked into providing sensitive data to the wrong application. In [Oprea et al. 2004], sensitive user I/O traffic to a trusted home computer via an untrusted public terminal is secured by an SSH tunnel established between the user's trusted mobile device and the home computer. Tunnel encryption and message authentication is made possible by a set of secrets shared by the mobile device and the home computer, exchanged during an enrollment phase.

Content protection technologies aimed at enforcing Digital Rights Management (DRM) are good examples for securing the user output channel between a piece of software and a display device. High-bandwidth Digital Content Protection (HDCP) is one such technology developed by Intel, which can establish a secure channel between a *source device* and *display device*. Both devices must be equipped with HDCP hardware, firmware or software enforcing the HDCP standard, including a matrix of HDCP keys. To secure user output, one could thus treat the computer's graphics card as a source device and the computer's monitor as a display device. The HDCP keys and protocols allow the source and display devices to mutually authenticate before any data is exchanged. If both devices determine their counterpart is a genuine HDCP device that has not been revoked yet (e.g., following a detected compromise), they establish a shared secret allowing them to encrypt user output traffic with a stream cipher. The Output Content Protection feature of the Microsoft Windows Vista OS leverages this HDCP technology [Lyle 2002] to create protected video paths and protected audio paths. Note that HDCP has been show to be vulnerable to cryptanalysis [Crosby et al. 2002].

Bastion: Bastion adopts the general approach of using cryptography to secure user I/O traffic. However, as opposed to the majority of approaches presented above, Bastion provides generic secure user I/O capabilities to protected software, that are not tied to a specific threat model (e.g., untrusted kiosk computing hosts) or business model (e.g., DRM in HDCP). The SP architecture [Lee et al. 2005] assumed generic secure user I/O logic, but it did not define the hardware and software mechanisms it required. Bastion defines primitives that can enable secure user I/O when trustworthy HID devices and informed users are in place. Bastion focuses on providing each protected software module with means to establish secure HID-to-CPU channels. The two endpoints can authenticate one another and maintain channel privacy and integrity despite an untrusted operating system and hardware adversaries. Bastion does not define how HID devices should be designed and protected, nor does it prescribe a method for machine-to-user and user-to-machine authentication. Bastion assumes that HID devices 1) are tamper-resistant, 2) can authenticated as genuine by the user, 3) can report to the user the identity of the software controlling the device.

6.B Basic I/O Mechanisms

Memory-Mapped I/O: On most modern computer systems, software programs can read from and write to I/O devices via a mechanism called *memory mapped I/O*, where the command, status and data exchange interfaces of an I/O device are mapped to a range of the processor's physical memory space. This allows programs to communicate with I/O devices using simple load and store instructions. Programs write data to a device interface by storing the contents of a processor register at the memory address corresponding to the device interface. They read data from a device interface by performing a memory load into a processor register.

In addition to these register-to-memory operations, some ISAs may allow for memory-mapped I/O operations via memory-to-memory moves. In both cases, an I/O transaction is equivalent to reading or writing a memory location. As a result, the memory compartments created by Bastion's Shadow Access Control mechanism can be used to give a specific software module exclusive access to an I/O device, e.g., give a trusted OS network driver module full control over the hardware Network Interface Card (NIC). This is done by mapping the range of addresses corresponding to the I/O device interfaces into the module's protected compartment. The software module can then communicate securely with the I/O device by sending and receiving encrypted and MAC'ed data and commands via these memory maps.

Direct Memory Access (DMA): Modern computer systems often support I/O transactions performed via Direct Memory Access (DMA), where data is transferred in bulk between an I/O device and main memory, without processor intervention. To carry out a transaction with an I/O device via DMA, software first needs to set up the transaction with a separate I/O device called a *DMA controller*. The DMA controller must be provided with a *DMA descriptor*, which identifies the device that is the source or destination of the I/O transaction. The DMA descriptor also specifies how much data is to be transferred between the device and main memory, and it defines a memory region where the I/O data is to be written to or read from. Finally, the descriptor specifies whether the transaction is to read data from main memory to write it in bulk to the I/O device (DMA output), or if the transaction is to read data from the I/O device and write it in bulk to main memory (DMA input).

In Bastion, the DMA controller does not need to be a trusted I/O endpoint. During setup of the transaction, there is no need for a secure channel to the controller since in effect, this setup is only a configuration of the untrusted buses and devices on the path to the trusted I/O endpoint. During the DMA transaction itself, the DMA controller simply relays secure channel data, which is encrypted and MAC'ed. It can thus be treated as an untrusted component. Following its setup, the DMA transaction is carried out, and the DMA controller relays, as per its configuration, the I/O data between main memory and the target I/O device, without having the processor intervene. Once the transfer is complete, the DMA controller triggers an interrupt to notify the processor.

Dedicated I/O Instructions: Some ISAs also offer dedicated I/O instructions, such as the `in` and `out` of Intel's x86 [Intel 2009]. These instructions typically take as operands an I/O device identifier and a processor register where the I/O data is to be read from or written to. Bastion treats these special register operations like the register-to-memory operations of memory-mapped I/O: the I/O data is read from or written to this dedicated I/O space (addressed with I/O device identifiers) directly to and from processor registers, just like for a memory-mapped I/O space. The difference between an explicit I/O space and a memory-mapped one is that Bastion's Shadow Access Control mechanism does not apply to this I/O space. As a result, I/O devices accessible via the dedicated I/O instructions cannot be bound exclusively to a specific Bastion compartment. It is still possible to establish an authenticated, confidential and tamper-evident secure channel between a specific Bastion compartment and an I/O device, but the processor can no longer guarantee that the software in this compartment is the only software that might send data to the device. This problem is similar to that of hardware attackers feeding forged data to I/O devices, which is already present in both the memory-mapped I/O or DMA I/O scenarios. To handle this problem, each module should use its own MAC key so the trusted I/O endpoint can determine the source of an I/O transaction and avoid impersonation. Extending Shadow Access Control to a dedicated I/O space is left for future work.

Chapter 7

Implementation

This chapter presents our implementation of the Bastion architecture on the OpenSPARC project. It involves modifying a SPARC processor core and its cache controller hardware to add Shadow Access Control and machine memory protection mechanisms to the microprocessor hardware. We also modified the Sun Hypervisor to implement Bastion hypercalls and the software components of shadow access control and machine memory protection. We ran our modified processor and hypervisor both within a simulator and on actual FPGA hardware. We first describe the OpenSPARC project and then present our implementation strategy for the SPARC hardware and the Sun Hypervisor. We conclude with a description of changes we applied to applications that request Bastion protection.

7.1 The OpenSPARC Project

The OpenSPARC project is an effort by Sun Microsystems to encourage research and development of new software and hardware ideas based on Sun's UltraSPARC processor architecture. The project makes publicly available the RTL (Register Transfer Level) source code of the main hardware components forming a microprocessor chip as well as the source code of the Sun Hypervisor, which runs on the so-called *hyperprivileged version* of UltraSPARC processors. The source code of the Legion functional simulator and the SPARC Architectural Model (SAM) architectural simulator are also made publicly available, along with the source code of several tools that can be used to develop and test new hardware or software functionality. Finally, the project publishes the source code and scripts necessary to produce a full-system Field-Programmable Gate Array (FPGA) implementation of an UltraSPARC microprocessor in hardware. The FPGA system can be fully synthesized, placed and routed on a variety of FPGA chips. Securing updates to FPGA logic is a challenge, especially when the FPGA implements security-critical functionality. While the issue is outside the scope of this thesis, we note here that we have tackled the problem in prior work, particularly with respect to remote updates [Champagne et al. 2008b, Badrignans et al. 2009].

OpenSPARC publishes the source code of two different implementations of the UltraSPARC architecture: the T1 microarchitecture code-named Niagara [Kongetira et al. 2005], and the T2 microarchitecture code-named Niagara-2 [Grohoski 2006]. The main differences between the two are that T2 supports more hardware threads and has a larger

L2 cache. The T2 microarchitecture also differs from T1 in its hardware debug hooks: some debug hooks are new in T2; some debug primitives are offered in software in T2 that were handled in hardware in T1, and vice versa. We chose to implement the Bastion architecture on the T1 microarchitecture for two reasons: 1) our proof-of-concept did not require the extra threads and cache capacity offered by T2, and more importantly 2) the OpenSPARC project only offers a full-system FPGA implementation for the T1 microprocessor. The processor core for FPGA implementation is the hyperprivileged version of T1, so it requires running the Sun hypervisor prior to loading an operating system and applications on the FPGA.

7.1.1 Software Tools

There are three main software tools offered by the OpenSPARC project: the Legion (functional) and SAM (architectural) simulators, and the *diags* regression test framework. The diags framework is a suite of small diagnostics test applications that allows developers to test whether the changes they make to processor hardware still maintain the definition of the UltraSPARC architecture. The framework runs diagnostic tests on either a Verilog simulator modeling the modified processor or on an actual FPGA implementation of the modified processor. The architectural state prevailing throughout the test on the simulator or FPGA is compared with the state observed on a run of the same test on a "golden", unmodified model of the processor, simulated by the SAM architectural simulator. The diags framework automates the compilation, linkage and loading of diagnostics tests onto the simulator or FPGA hardware. It also automates the running of tests and the comparison of their outputs with the golden run on SAM.

The Legion functional simulator distributed with OpenSPARC can run any commodity operating system that has been compiled for a Niagara processor, hyperprivileged edition. With the OpenSPARC release available at the time of this writing, the compatible operating systems were OpenSolaris and Linux Ubuntu. Legion needs to be invoked with a configuration file describing the hardware resources that should be made available to the simulated platform, e.g., physical memory, UART, console, etc. The configuration file must also define a virtual disk device mapped to an image file containing the root file system. This image file is read from the file system of the computer running the Legion simulator. It can be an exact copy of a file system taken from an actual computer based on a real Niagara processor.

Although the OS and applications can run unmodified on Legion, the Sun hypervisor code must be compiled with a special Legion flag to take into account the small differences in the hardware interface with Legion. During runtime, Legion can keep track of general-purpose register state and TLB contents instruction by instruction, but it does not simulate the cache hierarchy and other microarchitectural structures. As a result, it can only provide results in terms of instructions counts, not cycle counts. SAM is a more detailed instruction set simulator that can simulate hardware devices with great detail, making it a useful tool for device driver developers. Like Legion, however, it does not include a timing model so it can only provide performance results in terms of executed instruction counts. We used Legion during the implementation process. For our purpose, Legion provides the same level of functionality and precision as SAM, but with a faster simulation speed.

7.1.2 FPGA Hardware Evaluation Platform

The T1 processor core distributed with the OpenSPARC project can be synthesized for the Virtex 5 FPGA chip that is part of the ML-505 evaluation platform. The ML-505 platform is an FPGA board designed especially for the OpenSPARC project, and can be purchased from FPGA vendors. It integrates the FPGA chip with other peripheral hardware such as a DRAM chip, a Network Interface Card (NIC), a Flash memory card reader and several input and output connectors such as a Video Graphic Array (VGA), Universal Serial Bus (USB) and analog audio ports. OpenSPARC provides developers with a ready-made project for Xilinx Platform Studio (XPS), where the Niagara CPU core is integrated with a cache crossbar, an L2 cache and a memory controller to form a full system representing a CPU chip. All the components of the XPS project can be synthesized into a single bitstream configuring the Virtex 5 FPGA chip on the ML-505 platform.

Presumably due to time and FPGA resource constraints, the designers of the OpenSPARC project have implemented certain components of the full-system project in firmware rather than hardware. A Xilinx MicroBlaze processor was added to the project to run the firmware that emulates the missing hardware. The emulated components are the L2 cache controller and the I/O bridge, the hardware unit that connects the crossbar with on-chip peripherals other than the memory controller. L2 cache storage, defined to be 3MB in capacity in the T1 microarchitecture specifications, is absent from the FPGA full-system. All T1 memory is thus fetched and stored directly in main memory by the L2 cache controller, via the on-chip memory controller. An L2 controller is still needed despite the absence of L2 storage, to decode and form crossbar packets and to maintain L1 cache coherency when multiple T1 cores share the L2 cache. A multicore setup can be achieved using multiple FPGA chips, each with its own T1 core. Multicore architecture is outside the scope of this thesis.

Figure 7.1 depicts the block diagram of the full system FPGA implementation and the location of our new hardware additions: Shadow Access Control logic, new registers and instructions in the processor core; and a hardware crypto engine for memory protection (encryption and integrity tree hasing) between the MicroBlaze processor (emulating the L2 Cache) and the memory controller.

7.2 Processor Hardware

In this Section, we describe the modifications we applied to the source code of the T1 processor in order to implement Bastion functionality. The source code of the processor is in Verilog format and was synthesized using the Xilinx Synthesis Technology (XST) compiler. We first give an overview of the T1's microarchitecture and then proceed to explain our implementation strategy for adding new Bastion registers, Shadow Access Control Logic and Bastion instructions.

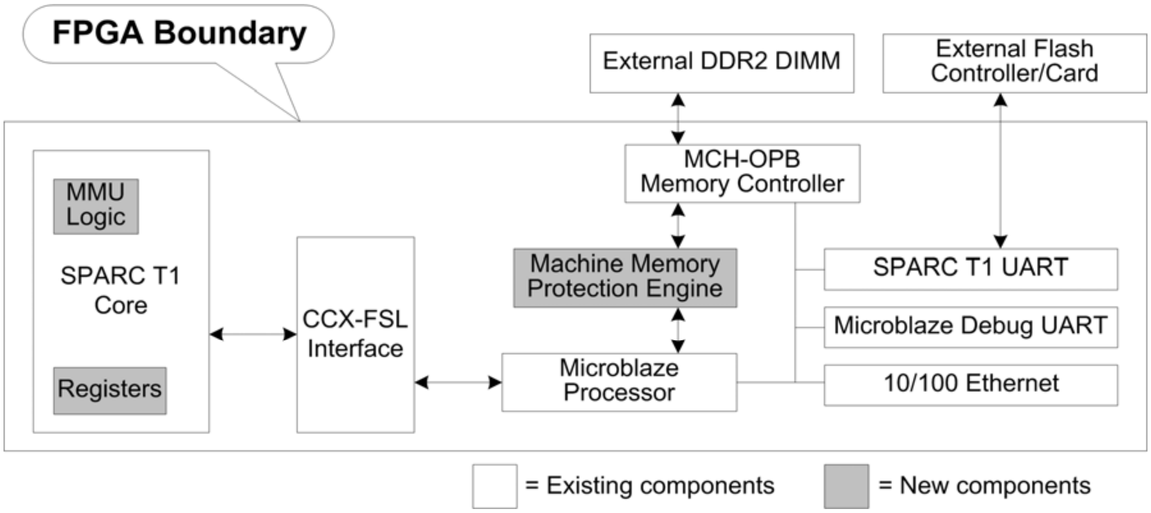


Figure 7.1. The OpenSPARC full-system FPGA implementation (white) with new Bastion components (gray). CCX stands for Cache Crossbar. FSL and OPB are the Fast Simplex Link and On-chip Peripheral Bus interfaces to the MicroBlaze processor. MCH stands for Memory Controller Hub.

7.2.1 UltraSPARC T1 Substrate

The T1 processor contains eight 64-bit UltraSPARC processor cores, each with four hardware threads of execution and a six-stage pipeline: Fetch, Thread Selection, Decode, Execute, Memory and Write Back. Figure 7.2 shows the T1 processor pipeline. Each core has its own L1 instruction and data caches, coupled with fully-associative instruction and data Translation Lookaside Buffers (TLBs). The instruction cache has a 16KB capacity while the data cache has an 8KB capacity. Both TLBs can accommodate 64 entries and support multiple page sizes. To reduce requirements for FPGA resources, the full-system project provided by OpenSPARC can be synthesized as a single-threaded, single-core version of the processor with reduced TLB capacities (16 entries). Security for multicore and multithreaded systems is outside the scope of this thesis. We thus use the single-thread, single-core optimization for the implementation described in this chapter.

Each processor core is formed of the following seven units: the Instruction Fetch Unit (IFU), the EXecution Unit (EXU), the Load/Store Unit (LSU), the Trap Logic Unit (TLU), the Stream Processing Unit (SPU), the Floating point Frontend Unit (FFU) and the Memory Management Unit (MMU). The IFU is responsible for fetching, decoding and scheduling instructions and traps. The L1 instruction cache and the instruction TLB are contained within the IFU. The EXU contains four functional units: an Arithmetic and Logic Unit (ALU), a shifter, an integer multiplier and an integer divider. These units are fed with data values from the register file as well as from the immediate operands decoded in the IFU. The EXU contains a variety of bypass paths allowing for quick forwarding of data results across pipeline stages. The LSU is responsible for processing most load and store instructions. It includes the L1 data cache, the data TLB, a load miss queue and a store buffer. The LSU is the interface between the core's functional units and the memory subsystem. It sends and receives data packets from the L2 cache via the crossbar. The TLU is responsible for receiving and processing all trap conditions. It prioritizes traps and keeps track of relevant processor state at each trap level using a trap stack.

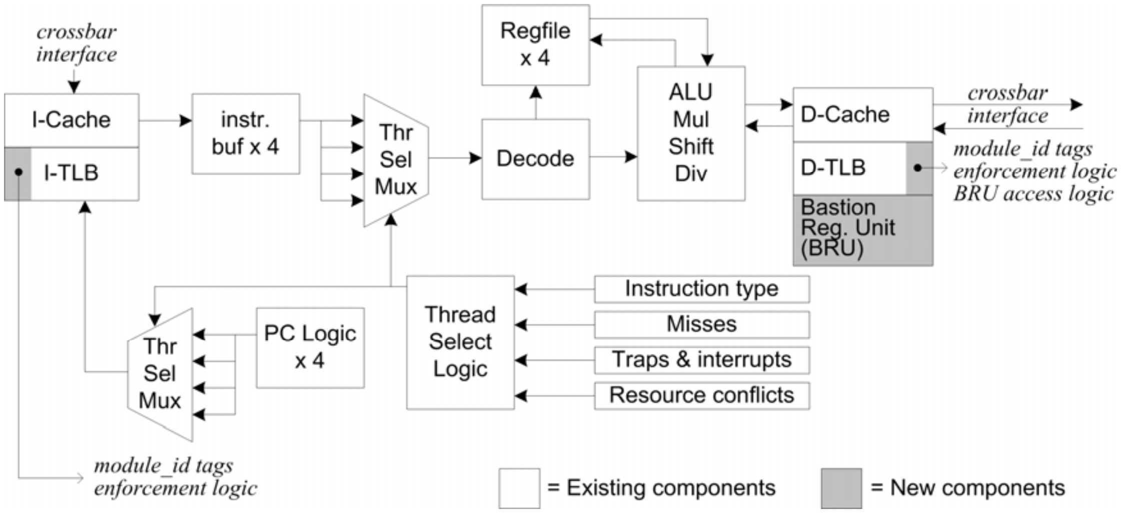


Figure 7.2. SPARC CPU Core with Bastion Additions (based on Figure from [Sun 2006])

The SPU is used to accelerate RSA asymmetric cryptography operations in hardware, for keys up to 2048 bits in width. It supports full modular exponentiation. To reduce FPGA resource utilization, our implementation synthesizes a SPARC core without the SPU. Asymmetric cryptography operations are rare in Bastion so they can be implemented in software without a significant impact on performance. The code of Bastion routines performing asymmetric crypto operations would need to be changed to benefit from the SPU's hardware acceleration in an eventual implementation on an Application-Specific Integrated Circuit (ASIC) or on an FPGA chip with more resources. The FFU unit interfaces with the Floating Point Unit (FPU) implemented outside the SPARC core. Once again, to reduce FPGA resource utilization without sacrificing functionality, the OpenSPARC project synthesizes a full-system without an FPU. All requests to the FPU are intercepted on the crossbar by the MicroBlaze firmware, which emulates floating point operations in software and returns the result on the crossbar.

The MMU is responsible for maintaining the contents of the data and instruction TLBs found in the LSU and IFU, respectively. System management software (i.e., the operating system or the hypervisor) must interact with the MMU when it wants to map or demap pages in the TLB (i.e., add or remove translations). The MMU is also responsible for generating fault status information on a TLB miss. The *partitionID* register identifying the current running virtual machine is stored by the hypervisor in an MMU register. This partitionID, also tagging each TLB entry, is what enables memory virtualization on UltraSPARC processors. There are thus several management functions that require having the OS or hypervisor read or write internal MMU registers. To make this possible, the UltraSPARC architecture offers special load and store instructions to *alternate address spaces*.

These instructions are like normal memory operations, except that they have an extra operand, the Address Space Identifier (ASI), which gets decoded within the MMU—e.g., Load Alternate (`lda` *mem_addr, gpr, asi*) or Store Alternate (`sta` *gpr, mem_addr, asi*), where *gpr* is a general-purpose register and *mem_addr* is the memory address where the data in the *gpr* is to be loaded from or stored to. ASIs are often used to direct the MMU to carry out a memory operation with or without virtual address translation or with a specific endianness. They can also be used to route load and store operations to internal CPU registers. In this case, the ASI specifies a certain set of internal registers and the load or store address specifies a register within that set. The MMU decodes the ASI and the address to identify the internal register where data is to be loaded from or stored to. Access to certain ASIs—e.g. the ASI where the partitionID register is mapped—is restricted to the hypervisor to ensure that OS and application software cannot break their virtual machine isolation.

7.2.2 Bastion Registers

The Bastion architecture defines several new processor registers that are to be used by the hypervisor and by secure on-chip routines. While some of these registers are defined to be non-volatile, the lack of non-volatile memory in the Virtex 5's user logic forces us to implement all Bastion registers as volatile storage. This still allows us to provide a proof of concept for Bastion's ability to work with an unmodified OS and gather data on its performance impact. Exploration of the different manufacturing technologies for implementing non-volatile storage on ASIC (e.g., fuses, EPROM, EEPROM) is outside the scope of this thesis. We note that while technology may make the non-volatile Bastion registers slower than volatile registers, the impact on performance should be insignificant since Bastion registers are used very rarely in comparison to general-purpose registers.

The infrequent use of the Bastion registers allows for an implementation away from the complex, tightly timed general-purpose register file. Following the advice provided by SPARC designers [Weaver 2008], we implemented the Bastion registers as a new unit, the Bastion Registers Unit (BRU), separate from the existing seven core units described in Section 7.2.1. The exact amount of register storage required in Bastion depends on the cryptographic primitives used to implement secure storage areas and attestation. For example, the width of the hypervisor secure storage hash in *storage_hash* could vary from 128 bits (for a hash value computed with AES in CBC-MAC mode) to 512 bits for a SHA-512 hash. Similarly, the width of the register storing the processor's private key (*cpu_key*) could vary between 512 bits and 4096 bits depending on the digital signature scheme employed and the desired performance-security trade-off. To accommodate for exploration of different schemes, we implemented the Bastion Register Unit as an array of thirty-two 64-bit registers, for a total of 2048 bits of register storage. Hypervisor and secure on-chip routine software can then choose a specific set of cryptographic schemes and allocate the Bastion registers within the available storage. If the need arose for a 4096-bit private key, the Bastion Register Unit can easily be scaled up to provide more register storage.

7.2.3 Shadow Access Control Logic

Four components of Bastion's Shadow Access Control logic must be implemented in the processor core: the *current_module_id* register, the instruction TLB tag extensions for the `module_id`, the data TLB tag extensions for the `module_id` and the logic for `module_id` comparison during TLB lookups. To implement these components with a minimum of changes to the existing SPARC core, we leverage the existing partitionID data path. The partitionID is very similar to the `module_id`: the former maps a page to a virtual machine compartment, the latter maps a page to a Bastion-protected OS or application compartment. In both cases, the mappings are expressed as extensions of the data and instruction TLB tags, containing the `module_id` or partitionID. At all times, the hypervisor is required to specify the partitionID of the virtual machine in control of the CPU in a "current partitionID" register, in the MMU. This mirrors the hypervisor's responsibility to specify in hardware the `module_id` of the currently executing software module in the *current_module_id* register. Furthermore, the partitionID is checked on every TLB lookup to ensure the data or instruction accessed is tagged with the partitionID in the "current partitionID" register. If there is a mismatch, the lookup triggers a TLB miss operation to be handled by the hypervisor. This is exactly the behavior defined in Bastion for TLB lookups on the `module_id`.

The similarities described above allowed us to implement the four hardware components of Shadow Access Control as extensions of the partitionID data path. The current T1 microarchitecture defines the partitionID to be 3 bits wide. We modified the hardware of the processor core to extend the partitionID to eight bits, in the "current partitionID" register, the instruction and data TLB tags and the buses leading for these storage elements. The T1 partitionID thus remains the same in the three least significant bits of these buses and registers, while the Bastion `module_id` is stored in the five most significant bits. This means our current implementation only supports thirty two concurrent modules: the unprotected module zero and thirty one other Bastion-protected modules. This is far more restrictive than what we envision in a full-blown ASIC implementation of Bastion, which should be able to support thousands of concurrent modules, with a `module_id` width of sixteen or more bits. We limited `module_id` width to exploit an unused but architecturally defined set of five bits in the 64-bit UltraSparc TLB tag. Anything larger than five bits would have required us to change the architectural width of TLB tags, which in turn would have required significant changes to hypervisor and operating system software. In a real-world implementation of Bastion architecture, the TLB tags would have to be defined to accommodate a larger `module_id`. On UltraSPARC, storing into hardware a TLB tag that is more than 64 bits (the processor word size) could still be done with a single instruction. Indeed, the ISA defines a special store operation that can take multiple registers and store them consecutively at a specified address, including within an alternate address space.

7.2.4 Bastion Instructions

Bastion introduces three new classes of instructions: 1) read or write Bastion registers (`bastion_read`, `bastion_write` instructions), 2) invoke secure on-chip routines (attest and secure_launch instructions), 3) hypercalls from application software. In UltraSPARC and possibly in other architectures with hardware support for virtualization, the ISA does not allow for application software running in user space to invoke hypervisor functionality via hypercalls. The logic behind this restriction is presumably based on the assumption that hypervisors would only offer system management services to their VMs via hypercalls. Therefore, the operating system was expected to require exclusive control over the invocation of hypercalls. In Bastion, however, we use the hypervisor hypercall mechanism to offer security services directly to applications, without the operating system as an intermediary. This requires us to redefine the hypercall mechanism so it allows at least some hypercalls to be invoked from application space, without having to go through the operating system. As a result, we consider the Bastion hypercalls as a new class of instructions.

Our goal was to implement the Bastion instructions without defining new mnemonics and operation codes (*opcodes*). This significantly simplified our implementation effort by avoiding modifications to the compilers and the processor's instruction decode logic. A full-blown ASIC implementation of Bastion should define new mnemonics to make Bastion functionality clearer to the user of the modified ISA. In our FPGA implementation, we circumvented the need for new opcodes by leveraging existing architectural mechanisms to extend the UltraSPARC ISA with the new Bastion instructions.

Bastion Register Access: To allow access to the registers in the Bastion Register Unit—i.e., to implement the bastion_read and bastion_write instructions—we mapped the BRU to an unused Address Space Identifier (ASI). This required changes in the ASI decoding logic of the hardware Memory Management Unit (MMU), but did not require any modification to the instruction decode logic in the Instruction Fetch Unit (IFU). All loads and stores are decoded in the IFU, but operations to alternate address spaces get forwarded to the MMU for decoding of the ASI operand. We thus added an ASI decoder in the MMU to intercept loads and stores to the BRU. This decoder generates the read or write enable signal for the BRU and activates the appropriate data path, either from the output of the BRU to the register file or from the register file to the input of the BRU. To invoke bastion_read or bastion_write, software thus executes an alternate load or store operation, with the ASI reserved to the BRU as an operand.

Most registers in the BRU should only be accessible to the hypervisor, and some should only be accessible to the Secure Launch and Attestation routines invoked by the secure_launch and attest instruction respectively. The ASI decoder described above thus needs to check the privilege level of executing software before allowing access to the BRU. Only software running in hyperprivileged mode—this includes the Secure Launch and Attestation routines—should be allowed access to the BRU. To distinguish between the hypervisor and on-chip routines, the decoder must also take into consideration the value of the current program counter. If it falls within the range of machine memory addresses that has been architecturally reserved for the on-chip routines, the decoder allows access to the Bastion registers reserved to these routines. Otherwise, access is denied. As detailed in Table 3.3 in Chapter 3, some Bastion registers are read-only, others are read-write; some registers can also be locked or unlocked by secure_launch. These access restrictions should be implemented in the BRU itself, where individual read- and write-enable signals can be generated for each register. To allow for a timely completion of our proof-of-concept, our current implementation does not generate individual enable signals nor checks the program counter when decoding the ASI. This reflects a working assumption that hypervisor and on-chip routine software always behaves correctly in the context of our proof-of-concept implementation.

On-Chip Routine Invocation: Invocation of the secure_launch and attest instructions needs to result in the execution of the corresponding Secure Launch and Attestation routines stored in on-chip memory. To implement these instructions without adding opcodes to the ISA, we fixed the entry points to the routine code within the reserved memory range and had hypervisor software invoke the instructions by executing a normal jump instruction with the appropriate entry point as a target. As for Bastion register access, we simplified implementation by assuming the hypervisor would always behave correctly and jump only to designated entry points, not directly to the middle of an on-chip routine. In an ASIC implementation of Bastion, the hardware needs to enforce that on-chip routines are only entered via designated entry points. This can easily be done by adding an actual secure_launch or attest opcode that gets decoded into a jump to a hardwired address corresponding to the routine entry point. In an ASIC implementation as well as in the current implementation, the return from an on-chip routine does not require a new instruction. The return address can either be passed as an operand to the instruction or recorded by the hardware during its execution. We choose to pass the address to the routines as an operand to simplify the required hardware. On-chip routines thus return into the code of their callers by jumping to the return address they received as an invocation operand. The routines need to perform some bound checking on the return address to ensure it points outside of secure on-chip routine code.

Bastion Hypercalls: The architecturally defined hypercall mechanism in UltraSPARC is the invocation of a specific software trap, with the hypercall identifier and its arguments in general-purpose registers. As stated above, the trap number reserved for hypercalls is made available only to privileged (i.e., operating system) software. As a result, invocation of a hypercall by application software, as required in Bastion, leads to a privilege violation fault. Because such violations of application/OS space privilege separation have no bearing on the hypervisor's ability to virtualize the software stack, UltraSPARC defines these traps as privileged, rather than hyperprivileged traps. The processor detecting such a condition jumps to a trap vector in the OS trap table rather than the hypervisor trap table. This means the hypervisor is unable to intercept these traps, hence we could not implement Bastion hypercalls with a simple change to the hypervisor to let user space hypercalls go through. Rather than change the hardware to reroute delivery of privilege violation faults to the hypervisor, we chose to define Bastion hypercalls as a different type of trap, which by default gets delivered to the hypervisor.

Instead of a software interrupt instruction, the Bastion hypercall is defined as an illtrap instruction (Illegal Trap), which the architecture defines as a hyperprivileged trap. To distinguish between the (unlikely) use of illtrap in regular software and its use as a Bastion hypercall, we require the Bastion hypercall be invoked with a special value in one of the general-purpose registers. This value not only distinguishes the invocation from a regular `illtrap`, but it also identifies the specific hypercall requested. Arguments to the hypercalls are stored in other general-purpose registers. We identify the origin of Bastion hypercalls by checking the value of the SPARC address space identifier in the processor register. If this strategy was to be adopted in a full-blown ASIC implementation, the definition of the ISA should be modified to specify that non-Bastion invocations of `illtrap` must leave all registers set to zero. A more elegant alternative is to implement the Bastion hypercalls as a dedicated software interrupt accessible to both OS and application software.

7.3 Cache Controller and I/O Bridge

Bastion memory protection mechanisms need to be located where every memory transaction can be intercepted, somewhere between L2 cache storage and external memory. In Niagara, we can thus implement the mechanisms in either the L2 cache controller or in the (on-chip) memory controller. We avoid modifying the memory controller circuit due to its complexity and the tight timing restrictions under which it must operate to interact with DDR-II DRAM chips. In the context of OpenSPARC, a major benefit of implementing memory protection mechanisms in the L2 cache controller is that this circuit is emulated in MicroBlaze firmware rather than in Verilog code. This enables faster prototyping for the current implementation and a lot more flexibility in future implementations, where different memory protection schemes may be compared for performance and complexity. We also modify the L2 cache controller to keep track of a virtual L2 cache storage, in order to better estimate the impact of our mechanisms if an on-chip L2 was actually implemented.

On the OpenSPARC FPGA implementation, the MicroBlaze firmware is also responsible for emulating the I/O bridge, which maps certain chunks of the machine address space to I/O peripherals and to internal units such as the Clock and Test Unit (CTU). In Bastion, the I/O bridge must be modified to enable mapping Secure Launch and Attestation routines to their reserved ranges of machine memory located in secure on-chip memory. The following sections describe our modifications to the MicroBlaze firmware to implement Bastion memory protection, secure on-chip routines and L2 cache storage emulation. We first start with a description of the current MicroBlaze firmware roles within the OpenSPARC full-system FPGA project.

7.3.1 MicroBlaze Firmware Roles

The main role of MicroBlaze firmware is to decode and form crossbar packets. In an ASIC implementation of Niagara, the I/O Bridge, the FPU and each L2 bank would have their own crossbar packet handling logic. In the OpenSPARC implementation, crossbar packet handling is centralized into one MicroBlaze firmware loop that constantly listens to the crossbar buses for incoming packets, which it then decodes and dispatches to the appropriate units. The crossbar itself is implemented in actual hardware, derived from a Verilog description. Not all I/O bridge, FPU and cache controller functionality is emulated by the MicroBlaze firmware, but enough is to enable execution of an unmodified commodity software stack.

The main I/O Bridge functionality implemented is the mapping of addresses accessed by the T1 core to internal units or peripherals. For example, crossbar packets for the Ethernet controller are sent to a firmware routine that decodes them, creates commands and sends them to the actual hardware Ethernet controller, implemented as an IP core on the FPGA chip. Crossbar packets destined for the FPU are decoded by a firmware routine and the floating point operations they request are emulated by another firmware routine. Crossbar packets for memory transactions are not directed to internal structures. They are sent to L2 cache controller emulation code, which further decodes them to determine the type of memory transaction requested, e.g., data load, data store, atomic swap, instruction cache fill from memory, instruction cache fill from PROM, etc. The memory transactions are carried out by forming and dereferencing pointers in the MicroBlaze address space to read or write T1 core data from the external DRAM chip. Since there is no L2 storage allocated on the FPGA chip, all memory transactions result in an access to external memory[26]. The firmware does, however, maintain the L1 Reverse Directory, an architecturally defined data structure that keeps track of L1 cache contents for the chip's processor cores, to preserve cache coherence. Based on this data structure, the firmware sends cache line invalidation packets defined in the UltraSPARC T1 cache coherence protocols.

7.3.2 Bastion Memory Protection

Bastion memory encryption and integrity verification mechanisms are implemented as a combination of MicroBlaze firmware routines and a hardware peripheral to the MicroBlaze processor. Due to limited FPGA resources, it was not possible to implement the entire Bastion memory protection scheme in hardware. We chose to have a single 128-bit AES ECB engine, described in VHDL, implemented in hardware as a MicroBlaze peripheral, connected via the Fast Simplex Link (FSL) bus interface. New MicroBlaze firmware routines act as device drivers for the AES engine. For each 64-byte L2 cache line read from an encrypted page, the routines request four decryptions from the AES engine, one for each 16-byte cache line sub-block. The routines then form the decrypted plaintext into a crossbar packet to forward the data to the requesting core. For writes to an encrypted page, the routines encrypt each 16-byte cache line sub-block with an invocation of the FSL peripheral. The routines then write the resulting ciphertext cache line to external memory.

A similar approach is taken for cache lines read from or written to integrity-protected pages. On a read, the firmware re-computes the tree branch covering the data read from memory. To complete the memory authentication procedure, it checks that the root of the recomputed branch is equal to the trusted reference tree root. On a write, the firmware re-computes the branch of interest and updates the reference tree root. The integrity tree we implemented has an arity of four and a node size of 16 bytes. Computing a tree node thus

[26] Except for data cached in the small Microblaze internal cache, which is allocated within the FPGA chip.

consists in computing a cryptographic hash function over the node's four children, either four hash nodes or the four sub-blocks forming a cache line. To compute a hash with a single ECB-mode AES engine, we have the firmware chain four invocations of the AES engine to implement the AES Cipher Block Chaining (CBC) mode. The computed AES-CBC-MAC hash is taken to be the ciphertext output of the last AES invocation. Following good security practices, we use different keys for ECB and CBC-MAC mode encryption. We have not yet implemented a routine for generating these keys anew on processor reset; the value of the keys is currently hard-coded in firmware code. Our implementation approach for Bastion memory protection engines is depicted in Figure 7.3.

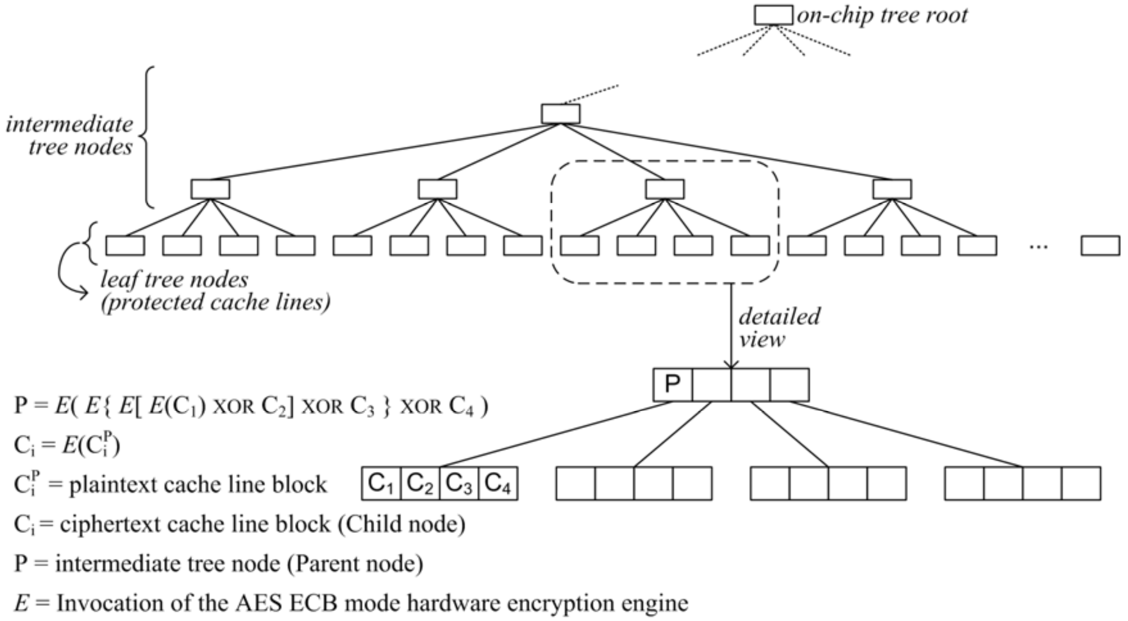

on-chip tree root
intermediate tree nodes
leaf tree nodes (protected cache lines)
...
detailed view
P
C_1 C_2 C_3 C_4
$P = E(\ E\{\ E[\ E(C_1)$ XOR $C_2]$ XOR $C_3\ \}$ XOR $C_4\ )$
$C_i = E(C_i^P)$
C_i^P = plaintext cache line block
C_i = ciphertext cache line block (Child node)
P = intermediate tree node (Parent node)
E = Invocation of the AES ECB mode hardware encryption engine

Figure 7.3. Our integrity tree, implemented with a single AES ECB mode engine for both encryption and hashing. The parent node P is effectively an AES CBC-MAC fingerprint of its four children C_1 to C_4.

The Bastion architecture defines a command interface between the software running on the T1 core and the memory protection engines. The interface is to be used by the `secure_launch` instruction and by the hypervisor software to instruct the engines that a given page is to be encrypted and added to the integrity tree's coverage. In an ASIC implementation of Bastion, this interface would likely consist of a new type of crossbar packet for I/O to the engines. New logic in the processor core would form these packets only when authorized software uses a reserved memory-mapped I/O range or dedicated I/O instructions. Each packet then consists of a command specifying a page requiring encryption and hashing.

To avoid defining a new type of crossbar packet and further modifications to the processor core, we implemented the CPU-to-crypto-engine interface as a shared memory interface between the T1 software and the Microblaze firmware. The interface consists of two control bits and two data bitmaps. Each bitmap has one bit per machine memory page that could be covered by the confidentiality and integrity protection schemes. To request encryption of a page, the hypervisor (or the secure_launch instruction) sets the page's bit in the first bitmap, called *to_encrypt*. To request the inclusion of a page under the integrity tree's coverage, the hypervisor (or the secure_launch instruction) sets the page's bit in the second bitmap, called *to_hash*. Software notifies the MicroBlaze firmware of a new request for encryption or hashing by setting the first control bit (called *update_protection*) to 1. The firmware notifies T1 software of the fulfillment of the request by using the second control bit (called *protection_updated*). The interface we defined thus allows software to set multiple page bits to request encryption or hashing of multiple pages in a single request.

We note that this implementation does not fully satisfy the definition of the mechanisms presented in chapters 4 and 5 of this thesis. The implemented interface requires each page to be read on-chip twice: once by the firmware for adding it to the coverage of memory protection mechanisms, and once by the T1 software to include the page's contents in an identity computation. This approach was taken to simplify implementation by avoiding the definition and implementation of an additional interface to synchronize T1 software and MicroBlaze firmware during Secure Launch. Our approach differs from the architectural definition, which requires that both operations be completed using a single read of each page. Our implementation thus adds a non-negligible performance overhead to every invocation of the Secure Launch procedure (the instruction or the hypercall). However, this is not critical since Secure Launch only happens during software launch, not during runtime. At the cost of about a doubling in the Secure Launch latency, our approach to the CPU-to-crypto-engine interface greatly reduced the complexity of the implementation effort without affecting security. Indeed, the security property central to Secure Launch—i.e., the data and code used to compute the software identity is the data and code that gets covered by the integrity tree—is achieved by first adding program pages to the integrity tree. The pages are then fetched back through the integrity verification mechanism to compute the identity of the piece of software being securely launched.

In addition to the interface described above, Bastion requires a mechanism to transfer i_bit and c_bit from a TLB entry to L2 cache lines on an L2 line fill. Again, to avoid further modifications to the processor core and to crossbar packet protocols, we implemented a simplified version of this mechanism. We extended the CPU-to-crypto-engine interface described above to include two more bitmaps, the c_bit table and the i_bit table. As for the previous interface, these tables each contain one bit per machine memory page potentially covered by the Bastion memory protection mechanisms. The difference between this and the previous interface is as follows: the *to_encrypt* and *to_hash* bitmaps contain *requests* to the MicroBlaze firmware for including *new pages* to the coverage of the memory protection mechanisms, whereas the c_bit and i_bit tables hold the *current state* of the i_bit and c_bit for *all existing pages*. In our current implementation, where the FPGA can only host 256MB of machine memory, and we use 8KB pages, each one of these tables is 32 kilobits in size.

If a bit is set in the c_bit table, the corresponding page is encrypted in memory, hence its cache lines have to be decrypted upon fetch. Conversely, cache lines written back to the page have to be encrypted. Similarly, a set bit in the i_bit table indicates that access to cache lines from the corresponding page must trigger integrity tree verifications and updates. Both tables are looked up upon every memory access carried out by the MicroBlaze on behalf of the T1 core. Due to time constraints, we did not implement Bastion features related to the swapping of protected virtual pages to disk. This means we do not currently have an interface to the memory protection engines for requesting the relocation of a page within the tree.

7.3.3 Secure On-Chip Routines

In a full-blown ASIC implementation of Bastion, the code and initialized data space of the secure on-chip routines must be stored in an on-chip non-volatile memory. However, the Virtex 5 FPGA chip does not offer non-volatile memory resources to the user logic. (The user logic is the reconfigurable part of the FPGA chip, where developers can implement their hardware designs. This is in contrast to the FPGA's static logic which decodes the bitstream and configures the user logic accordingly.) To implement Bastion on-chip routines on the Virtex 5, we thus assumed that part of the MicroBlaze address space was mapped to a non-volatile memory on-chip. We reserved a range of the T1's machine memory space for the secure on-chip routines and modified the I/O bridge firmware to map this T1 range to the "non-volatile" part of the MicroBlaze address space. Therefore, when the secure_launch or attest instructions are invoked on the T1 core, the instructions that follow are fetched from the "non-volatile" on-chip memory.

These Bastion routines not only need an on-chip storage for their binaries, they also need a secure on-chip execution environment, where they can store and retrieve the data they generate dynamically. The hardware must guarantee that none of the code or data in this execution environment will leave the chip. To achieve this, we simply made the "non-volatile" memory large enough to accommodate dynamic data. The amount of data needed can be measured by profiling the routines and ensuring that they can operate within a fixed amount of memory. In our case, the routines were implemented in C and compiled to SPARC64 assembly. The most complex operations they perform are cryptographic functions like hashing (secure_launch) and digital signature computation (attest). This code was extracted from the OpenSSL library [Cox et al. 2009] and linked statically to our secure routine binaries. As described in detail in Chapter 8, implementing these routines required less than 5,000 lines of code.

It is essential that our Bastion routines be self-contained executables. They cannot rely on operating system or hypervisor mechanisms such as traps, hypercalls or system calls. This requires static linkage of all libraries and inlining of any hardware interface code. Before carrying out their function, the routines must ensure they have full control over the CPU—e.g., system timer interrupts must be deactivated to avoid having unmeasured software preempt secure routine execution. We implemented our routines as a standalone, minimalist hypervisor, more or less the smallest amount of standalone code that can execute in hypervisor mode on UltraSPARC. This allowed us to have routine code that is fully privileged and thus can access all memory and assert full control over critical processor settings.

7.3.4 L2 Cache Storage Emulation

Memory protection schemes operate only on data that transit between the L2 cache and memory. Data that are cached in L2 do not need to be decrypted when read or encrypted when written since L2 storage is within the security perimeter, i.e., the processor chip. Similarly, reading cached L2 data does not require an integrity tree verification since the data was verified upon being loaded into the cache, from external memory. An L2 cache line being written by the processor core does not need to trigger an integrity tree update, since the tree only needs to fingerprint the state of external memory. Updates carried out in the cache get reflected in the tree upon eviction of the dirty L2 cache line. In security processors, caching of a line in L2 thus not only makes access to the data faster, it also saves the *O*(log(N)) cryptographic operations required by the memory protection mechanisms, where N is the number of protected memory blocks. To improve the performance of our implementation, we must thus ensure that access to cached L2 data does not trigger memory protection operations.

Unfortunately, we cannot provide L2 cache storage in our implementation due to limited FPGA resources. However, we can implement L2 cache storage emulation to keep track of which data would be in the L2 cache, if we actually had an L2 cache. This allows us to avoid integrity tree and encryption operations for data that is resident in the L2 cache. This approach still does not make our implementation fully realistic: accessing “cached” data takes as much time as accessing data in external memory. This is because although the MicroBlaze firmware can determine the data is “cached”, it still has to fetch it from external memory to satisfy a read request from the T1 processor core. Nevertheless, we can now make this “cached” read more realistic by avoiding cryptographic operations that would not be carried out in an ASIC implementation.

To implement L2 storage emulation, we defined an array of cache lines in the data space of the Microblaze firmware. The array follows the UltraSPARC T1’s microarchitectural definition of the L2 cache: 64-byte lines, four-way set associative, three megabyte total capacity, each line with an accessed bit and a dirty bit. Each array element holds a tag wide enough to identify a line within such a cache, but it does not allocate memory for storing the actual data content of the line. This is because fetching a cache line from this “cache array” in MicroBlaze firmware or fetching its machine memory location usually leads to the exact same operation: a fetch from external memory. Indeed, MicroBlaze firmware executes from external memory so except for when a piece of data is in the MicroBlaze’s small internal cache (on the FPGA chip), accessing the MicroBlaze data space means accessing external memory.

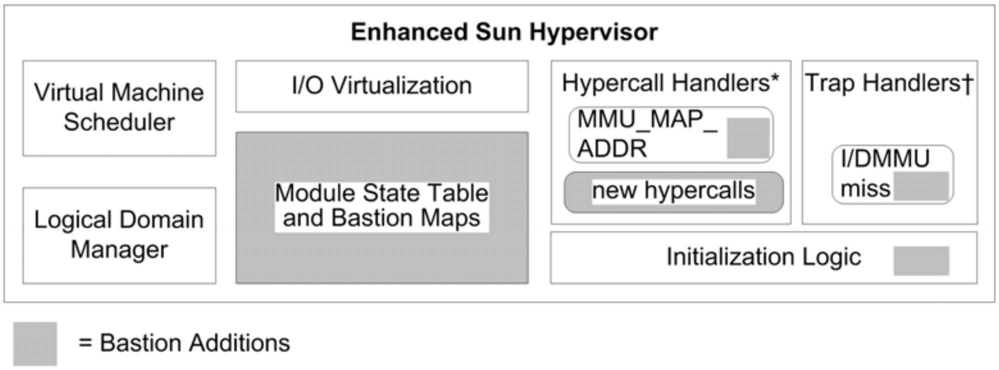
Accessing the “cached” line from external memory avoids the variability due to unpredictable MicroBlaze caching patterns. We implemented an L2 access function that is invoked by MicroBlaze firmware whenever a memory access is performed. This function checks whether a line is “cached” by comparing the address of the accessed datum with the tags in the cache line array, and it updates the accessed and dirty bits according to the type of access performed. When a line is not already in the array, the function looks for an available array element in the set where the line maps. If none is available, it selects a line for eviction according to the UltraSPARC cache replacement algorithm. It replaces the line being evicted with the line being accessed. When the evicted line is dirty, it gets written back to main memory encrypted and the integrity tree is updated to reflect the new state of the line, depending on the `c_bit` and `i_bit` values in the `c_bit` and `i_bit` tables.

7.4 Hypervisor

In this section, we first give an overview of the Sun hypervisor's structure and then describe the changes to its code base that were required by Bastion. Except for a few initialization functions written in C, the vast majority of Sun hypervisor code base is written directly in SPARC64 assembly code. Development in assembly language is error-prone and difficult to debug. Due to the time constraints on the Bastion project and a lack of skilled SPARC64 assembly programmers, we chose to write complex Bastion functionality in C code. This fast prototyping strategy runs contrary to the UltraSPARC architecture's intended hypervisor behavior—e.g., no stack use during runtime, no use of SPARC register windows—so it required developing some special C function wrappers (in assembly) to ensure C code execution would not corrupt the state of software running in the virtual machines. We describe these wrappers in Section 7.4.2.

7.4.1 Sun Hypervisor Overview

Figure 7.4 depicts the Sun Hypervisor and our Bastion additions, described in detail below. The Sun hypervisor creates one or more Virtual Machines and allocates each of them its own share of platform resources—e.g., memory and I/O devices. As opposed to other hypervisors that are an optional component in a regular platform boot process, the Sun hypervisor is distributed with the platform, as firmware stored in a PROM. The processor's reset vector points to a small piece of reset code which then systematically jumps to the hypervisor's initialization routines in PROM. One of the first actions taken by these routines is to relocate the hypervisor to platform RAM, where it can allocate memory for a dynamic data space. The hypervisor then allocates and initializes data



* The UltraSPARC Hypervisor supports 147 hypercalls. We modified one and created 9

† The UltraSPARC Hypervisor supports 66 trap types. We modified 2 of them.

Figure 7.4. The Bastion-Enhanced Sun Hypervisor

structures describing the virtual machines it is to run. We extended these initialization routines to initialize the Bastion data structures, e.g., the Module State Table. The hypervisor then copies the OpenBoot "firmware" to each virtual machine's address space and triggers virtual power-on reset interrupts to initiate the bootstrap of each VM.

During runtime, the hypervisor's main roles are to virtualize the machine memory space and provide each VM with its own slice of CPU time. The hardware supports the hypervisor's virtualization of memory by routing every MMU-related trap (e.g., instruction and data TLB miss traps) to the hypervisor's trap handlers. In addition, it allows the hypervisor to define memory partitions and map pages to partitions using the partitionID register and TLB tags described in Section 7.4.4. The hypervisor thus retains full control over how guest physical addresses are mapped to the machine memory space. To enable sharing of CPU time across virtual machines, the UltraSPARC core offers a hyperprivileged system timer that allows the hypervisor to set periodic interrupts to take back CPU control and schedule a different VM.

The Sun Hypervisor does not virtualize I/O devices, i.e., it does not host device drivers that allow several virtual machines to unwittingly share the same hardware I/O device. This architectural decision was taken to minimize the amount of code in the hypervisor and to avoid having to update hypervisor code base every time a new or updated device is to be supported [Saulsbury 2008]. The Sun Hypervisor assigns I/O functionality exclusively to a specific Virtual Machine, either at the bus level (to map a group of devices to a VM), at the device level (to map a device to a VM) or at the function level (to map a particular function of a given device to a VM). The Sun Hypervisor assumes the platform has an I/O MMU to prevent virtual machines from breaking out of their memory compartments using DMA I/O transactions. The I/O MMU is configured by the hypervisor to ensure that DMA transactions by a device mapped to a given VM can only access memory allocated to that VM.

When many VMs want to share an I/O device, they need to designate a specialized VM that handles the device. This VM is responsible for receiving and servicing I/O requests from the other(s) VM(s). To allow for communication between VMs, the Sun Hypervisor allows for the establishment of *Logical Domain Channels*, essentially a mailbox mechanism allowing two VMs to exchange data and commands. We do not use or discuss this mechanism in the rest of this thesis.

Another important function of the Sun Hypervisor is to handle hypercalls from operating systems. The UltraSPARC hyperprivileged edition requires operating systems to be aware that they are being virtualized (i.e., UltraSPARC implements para-virtualization, as described in Section 4.1.2 of Chapter 4). Indeed, an OS running on the Sun hypervisor needs to invoke hypercalls in order to be able to manage the hardware resources under its control. For example, it needs to obtain I/O device mapping information from the hypervisor before it can set up I/O transactions, and all console operations have to be carried out via dedicated hypercalls. In addition, an OS must use hypercalls in order to inform the hypervisor of current valid virtual-to-guest-physical memory mappings. This can be done directly, with one hypercall specifying one mapping, or indirectly, via the use of a special Translation Storage Buffer (TSB) structure. A TSB is essentially a cache for the most frequently used entries from a guest page table. There is one TSB per guest page table, and the OS describes each one to the hypervisor using a hypercall. When TSBs are used, the hypervisor first looks into the relevant TSB for a translation on a TLB miss, rather than automatically revectoring the trap to the operating system. In all cases, hypercalls are invoked via a dedicated software interrupt number, with up to five parameters passed in general-purpose registers. All hypercalls are handled by SPARC64 assembly code and first go through a dispatcher that identifies the requested hypercall by looking at the dedicated register operand.

7.4.2 Fast Prototyping Strategy

In the baseline hypervisor, the only code written in C is the guest initialization code that runs at the very beginning of hypervisor execution. Using C code makes traversing complex guest configuration data structures much simpler than in SPARC64 assembly. On UltraSPARC, executing C code requires using a stack and register windows since both are expected to be available by the C compiler. During initialization, using C code is not a problem since a small stack can be allocated within hypervisor memory space and no software other than the hypervisor needs register windows.

However, the Sun hypervisor was not designed to use a stack and register windows during runtime, when VM-based software has its own stack and is also using the register windows. As a result, entry points into hypervisor code do not set up a stack or save the current state of register windows. An unmodified hypervisor executing C code at runtime would thus end up using the guest’s stack pointer (invalid since it is either a virtual or guest physical address, whereas the hypervisor uses machine addresses) and corrupting guest register windows irreversibly. This is why runtime hypervisor code is written in SPARC64 assembly, using only a special set of global registers that are allocated afresh upon trapping into hypervisor code.

To allow for fast development of Bastion hypervisor functionality in C code, we thus implemented SPARC64 assembly wrappers that execute prior to invocation of a Bastion function written in C. These wrappers set up a private stack for the hypervisor and free up at least one register window, to allow C code to use any register it may require. While these wrappers and the use of C code negatively affect the performance of our implementation, our approach ensured we could produce a Bastion prototype within a reasonable time frame.

7.4.3 Bastion Hypercalls

The Bastion hypercall mechanism we implemented allows application software to invoke hypercalls directly by overriding the definition of the `illtrap` instruction on UltraSPARC. Prior to invoking `illtrap`, applications must load specific general-purpose registers so they contain a hypercall identifier and the arguments to the identified hypercall. We rewrote the Sun Hypervisor’s handler of the illegal trap event to make it into a Bastion hypercall dispatcher. This dispatcher (less than one hundred lines of

SPARC-64 assembly code) reads the register identifying the hypercall and invokes the Bastion routine handling that specific hypercall. The routine can then use the parameter registers as part of its servicing of the hypercall.

TRANSLATE_VA: Application software executes out of a virtual address space and the hypervisor executes out of the machine address space. Hence memory pointers generated by the application are not directly usable by the hypervisor: they must be translated from a virtual address pointer to a machine memory pointer. Bastion hypercalls such as SECURE_LAUNCH have an application provide a memory pointer (e.g., to a security segment data structure) referencing application memory. To be able to read or write the referenced memory, the hypervisor can translate the pointer via guest and nested page tables (as described in Section 4.1.3 of Chapter 4) and dereference the translated pointer. In UltraSPARC, the hypervisor can also use the virtual address pointer directly in a special load or store instruction targeted at an alternate address space. The hardware executes this instruction by translating the referenced virtual address as though it was the application, not the hypervisor, that was performing the operation.

We adopt the translate-then-dereference approach since the hypervisor needs the translated pointers anyway to populate the vMap (see Section 5.1.1 of Chapter 5). Rather than provide the hypervisor with a virtual pointer that it must translate, we translate pointers from application space and provide machine pointers to the hypervisor. Applications carry out translations by providing a virtual pointer as an argument to a new TRANSLATE_VA hypercall, which returns the corresponding machine address. The hypervisor handles this hypercall with a routine that is practically identical to the TLB miss handlers. The routine looks up TSBs to find the appropriate virtual-to-guest-physical address translation and generates the requested machine address by looking up its own mappings of guest physical memory spaces. The application can use the machine pointer thus obtained to reference application memory in hypercalls. This implementation approach based on the TRANSLATE_VA hypercall was chosen for convenience and is not a requirement. The TRANSLATE_VA handler should be invoked by the hypervisor directly, from within hypervisor space, whenever it needs to translate a virtual address[27]. This is why the TRANSLATE_VA hypercall is not part of the definition of the Bastion architecture.

SECURE_LAUNCH: The SECURE_LAUNCH hypercall receives as an argument a pointer to a module's security segment. A C-language routine in the hypervisor handles the hypercall by parsing the security segment and by allocating and populating a Module State Table entry for the module being launched. In the process, the hypervisor checks that the private and shared pages requested do not violate the memory compartments defined by modules launched previously. This is done by looking up the mMap for each page described in the security segment, as described in Chapter 5. The Module State Table is a global data structure allocated in the hypervisor's BSS (Block Started by Symbol) section of its data space. This allows reserving the large amount of memory required to store state for multiple modules during runtime, without taking up any space in the binary file of the hypervisor. This is essential in UltraSPARC because the hypervisor's binary image

[27] We recommend this approach, since exposing machine addresses to VM-based software is not only inelegant, but might also introduce covert channels across VMs, which share the same machine memory space.

is stored in a PROM with a limited capacity. We extended hypervisor initialization routines to initialize upon startup the BSS region of memory containing the Module State Table.

CALL_MODULE: Our implementation of the CALL_MODULE hypercall does not carry out the actual jump into callee code. Instead it just validates the requested transition and registers it in an implementation-specific *Transition Registry* field of the Module State Table and returns to caller code. The caller code then carries out the actual jump to the callee code, which triggers a TLB miss due to a `module_id` mismatch. As explained in detail, in Section 7.4.4, the Bastion-enhanced TLB miss handlers then carry out the actual module transition by changing the `module_id` in the *current_module_id* register in hardware. Once again, this approach was taken for convenience and is not necessary. The hypervisor handler for CALL_MODULE could effect the transition by changing the `module_id` in hardware and returning from the hypercall by jumping to the requested entry point in callee code.

RETURN_MODULE: We implemented the RETURN_MODULE hypercall using the same approach as for CALL_MODULE. The handler for RETURN_MODULE registers the requested transition back into caller code in a Transition Registry and returns to the code that invoked the hypercall. This code then carries out the actual return into caller code, typically using a `return` instruction that reads the return address from the stack. This jump into the caller's compartment triggers a TLB miss, which the hypervisor services by changing the `module_id` in the hardware. Again, a more direct way to implement RETURN_MODULE is to have the handler of the hypercall carry out the actual jump back into caller code and change the `module_id` in hardware itself.

ATTEST: In our implementation of the ATTEST hypercall, the application or OS module can bind extra data to the attestation report, in addition to its certificate. The hypercall also uses a 32-bit bitmask to identify modules in a trust domain, rather than have the hypervisor keep track of trust domains in the tdMap (not implemented). This approach was taken at a time when the definition of the architecture was still evolving. With the ATTEST hypercall defined in Chapter 6, modules can bind any arbitrary amount of data to an attestation message by hashing everything into the certificate; there is no need for a separate data argument to the ATTEST hypercall. As implemented, the hypercall takes five arguments: 1) a `module_id` mask specifying the set of modules to be attested to, 2) a pointer to memory containing data to bind to the attestation report, 3) an integer specifying the size of these data, 4) a pointer to the certificate to bind to the attestation report, and 5) a pointer to module memory where the final attestation report signed by the CPU is to be written back.

After doing some validation on the arguments, the hypercall handler allocates a memory buffer large enough to store the identities of all modules specified, the data referenced by the second and third arguments and the certificate. These data are copied in the buffer so they are in a contiguous memory region. The handler then computes a cryptographic hash over the contents of the buffer. This hash becomes the hypervisor's certificate to use as an operand to the `attest` instruction. The hypervisor then invokes the attest instruction with the address of the certificate and the fifth hypercall argument as operands.

The on-chip processor routine concatenates the hypervisor certificate with the hypervisor identity read from the *hv_identity* register, and digitally signs the concatenated data. The resulting signature is copied at the memory address specified by the second operand to the attest instruction, which is the fifth hypercall argument: the module-allocated attestation report buffer. The on-chip routine then returns to the hypercall handler, which returns back to the caller module.

Although it was not done in our implementation, the ATTEST hypercall handler should copy into the attestation report buffer the full-length version of the data that was condensed into the signed attestation report—i.e., the identities of the hypervisor and all modules reported. The hypervisor should also provide the module with the public key of the processor, which can be embedded in the hypervisor's data space. In turn, the module should use these extra data in its communication with a remote attestation party, to simplify the party's trust decision. To make this trust decision efficiently, the remote party needs to know which processor it is communicating with (identified by the public key) and which software identities were hashed together to form the signed attestation report.

Secure Storage Hypercalls: We implemented the four architecturally defined secure storage hypercalls: WRITE_STORAGE_KEY, WRITE_STORAGE_HASH, READ_STORAGE_KEY and READ_STORAGE_HASH. The handlers servicing these hypercalls are very simple since Bastion offloads to the modules themselves all the hashing and encryption involved in secure storage. As their mnemonics imply, the READ_STORAGE_KEY and READ_STORAGE_HASH hypercalls only have to read the module's secure storage key and hash from the Module State Table, while WRITE_STORAGE_KEY and WRITE_STORAGE_HASH store new key and hash values in the Module State Table. To commit the state of newly updated module storage areas to its hardware-anchored secure storage, the hypervisor can then compute a hash over the new module key and hash values, and commit the computed hash to the *storage_hash* register in hardware, using the "store to alternate address space" instruction. Multiple invocations of the instruction are necessary to store the hash value, whose storage requires several 64-bit registers in the Bastion Register Unit. To compute hashes in software, we linked the Bastion hypervisor with object files obtained from the OpenSSL [Cox et al. 2009] library, to get access to the SHA cryptographic hashing primitives.

7.4.4 TLB Miss Handler Changes

The Sun Hypervisor virtualizes memory using a form of nested paging, handled mostly by handlers for instruction and data TLB miss events. The role of these handlers is to provide the hardware with the virtual-to-machine address translations that are needed by the processor but absent from the TLB. The handlers can either generate the missing translation themselves by looking up the guest TSBs, or they can revector the TLB miss trap to the concerned guest OS. In the former case, the hypervisor inserts the missing virtual-to-machine translation itself in the TLB hardware using a store instruction to an alternate address space mapped to internal TLB registers. In the latter case, the OS looks up the appropriate guest page table, updates the related TSB (if any) and then invokes the MMU_MAP_ADDR hypercall to commit the virtual-to-guest-physical translation to the hypervisor directly.

The MMU_MAP_ADDR hypercall is offered to operating systems by the Sun Hypervisor to allow them to request the mapping of a page into the TLB. This hypercall must be invoked for every page when the TSB interface to the hypervisor is not used. When the TSB interface is used, the operating system still invokes the hypercall to accelerate TLB insertions following a TSB miss, as described above. The handler of this hypercall takes the virtual-to-guest-physical translation provided by the invoking OS and—after confirming the requested memory should be made accessible to the guest—transforms it into a virtual-to-machine translation, which it inserts in the hardware TLB. We needed to modify this handler to ensure that TLB entries were tagged with the right `module_id`.

As described below, the TLB miss handlers already perform the Shadow Access Control checks that determine `module_id` tags. Rather than adding full-blown Shadow Access Control checks to this hypercall handler as well, we only process the simplest Shadow Access Control case in this handler and leave the more complicated cases to the TLB miss handlers. Our goal was to centralize all Shadow Access Control decisions into a single point of control: the TLB miss handlers. Our extended MMU_MAP_ADDR handler thus only inserts TLB entries if they belong to the module zero compartment. Otherwise, the handler simply returns to the invoking OS. Not knowing that the requested translation has not been inserted in the TLB, the OS will reschedule the software that caused the TLB miss. The TLB miss occurs again, but this time, the TLB miss handlers can handle the miss themselves, including the Shadow Access Control checks that we now describe.

We extended TLB miss handler logic with code enforcing Bastion memory compartments. A Bastion routine looks up the mMap described in Chapter 5 to check that a translation about to be inserted in TLB hardware respects Shadow Access Control rules. The routine first identifies the executing software module by reading the *current_module_id* register and then checks that this module is allowed to access the page about to be mapped into the TLB. If so, the translation is inserted in the TLB hardware, tagged with the `module_id` of the executing module. If not, an exception is raised to signify a memory compartment violation. Our implementation does not try to recover from these exceptions, it simply halts all executing software. We leave it to future work to implement Bastion's resilient execution capabilities, where only infringing modules are halted and surrounding software is allowed to make forward progress, use its secure storage and perform attestation.

We also adapted instruction TLB miss handlers to detect and process module transitions. The first action taken by our extended miss handler, before any translation is created and validated as above, is to check whether the current instruction TLB miss was caused by a legitimate module transition, or by the OS preempting module execution. These checks are carried out by a C function analyzing the context of the miss event. This function looks at the current Trap Program Counter (TPC), the TPC of the previous trap level (if applicable), the current *contextID*, the current `module_id`, and the contents of the UltraSPARC *pstate* register.

The TPC is always available. It is read from an UltraSPARC register and identifies the instruction that caused the current instruction TLB miss. The previous TPC exists only if the current TLB miss was caused during handling of a previous trap. Such a double trap event occurs whenever a Bastion-protected module is preempted by a guest OS interrupt. In this case, the timer interrupt that preempts the module causes a first level of trap, a timer interrupt trap. But since the OS handler for the timer interrupt trap is in a different module (typically in module zero), the CPU's jump to the OS interrupt handler causes a second trap; this time, it is an instruction TLB miss caused by a `module_id` mismatch. The contents of the TPC register thus reflect the address of the timer interrupt handler's first instruction, while the value of the previous TPC is the address of the module instruction that got preempted by the timer interrupt.

The *pstate* register reflects the privilege mode of the processor when the TLB miss occurred. It is either in privileged mode or in non-privileged mode. The former indicates the OS was executing when the trap occurred, while the latter indicates an application was executing. The final piece of information examined by our C routine is the *contextID*. This is a value in an UltraSPARC register identifying the application that was

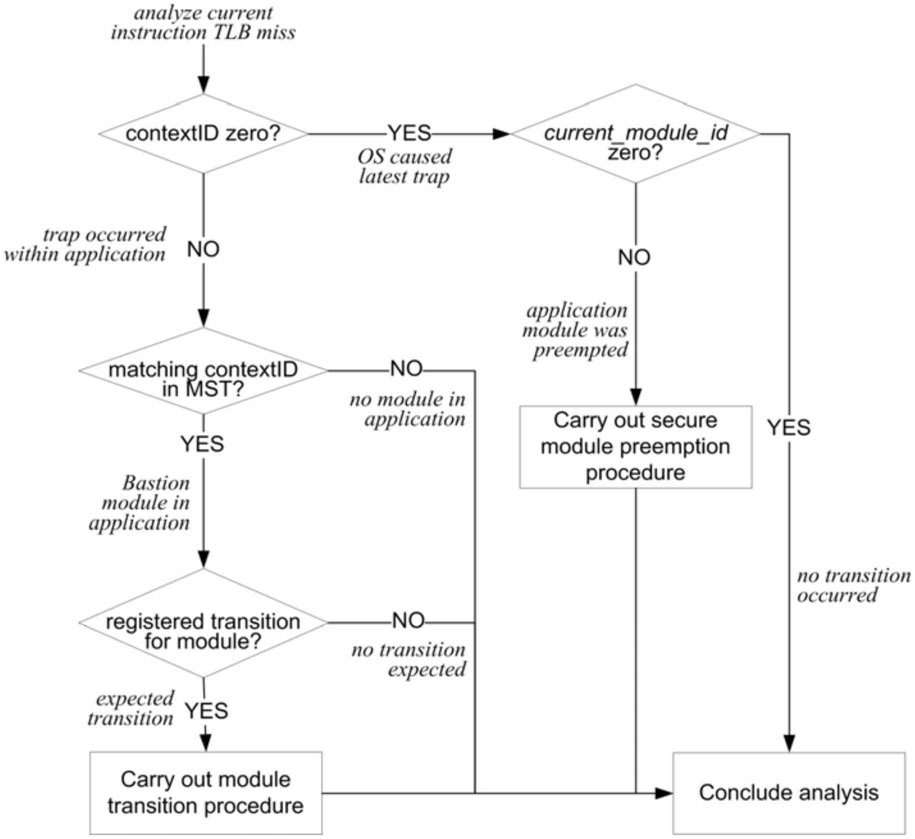

analyze current instruction TLB miss
contextID zero?
YES
OS caused latest trap
current_module_id zero?
trap occurred within application
NO
NO
application module was preempted
matching contextID in MST?
NO
no module in application
Carry out secure module preemption procedure
YES
YES
Bastion module in application
no transition occurred
registered transition for module?
NO
no transition expected
expected transition
YES
Carry out module transition procedure
Conclude analysis

Figure 7.5. Analysis of instruction TLB miss events to detect module transisitons

executing when the trap occurred: when the OS executes, the *contextID* is zero; when an application executes, the contextID is set to a value unique to the application, assigned by the OS upon application launch. We use this value as our *context_id* address space identifier defined in Chapter 5.

Figure 7.5 depicts the analysis performed to detect module transitions on an instruction TLB miss. If the *contextID* is zero, the OS was executing when the miss occurred. This means that either 1) the OS has had a regular instruction TLB miss or 2) a protected application module was interrupted by the OS, which caused an instruction TLB miss due to a `module_id` mismatch. The latter form of TLB miss event cannot occur within the OS since our implementation does not currently support OS-based protected modules. (To support OS-based modules, we need to extend our analysis routine to detect inter-module transitions within privileged mode, where the contextID is always zero. This case should not require any major changes, as it is similar to a case we already support: multiple modules within a given application, with a given contextID.) Therefore, to distinguish between scenario 1 and 2, we look at the *current_module_id* in hardware. If it is zero, we were not in a protected module when the miss occurred, so the OS simply had a regular TLB miss (scenario 1). There are no modules in the OS so this miss even could not have been the result of a module transition. We conclude the analysis since we do not have to carry out any module transition and no protected module was preempted by the OS code that caused the trap. A non-zero value for *current_module_id* tells us that a protected application module was executing prior to the OS taking control of the CPU. Therefore, our C routine carries out the secure preemption procedure. This involves saving the state of the preempted module in its Module State Table entry, along with the value of its Program Counter (PC) upon preemption. As explained above, this value is the previous TPC passed as an argument to the C routine.

A non-zero *contextID* indicates that an application was executing when the instruction TLB miss trap occurred. In UltraSPARC, application code executes in “trap level zero”. This means that a trap taken by an application is always a first level trap. There are no trap handlers in application space so an application trap cannot occur during the processing of a prior trap. Therefore, there is no previous TPC argument to the C analysis routine, only a TPC reflecting the address of the application instruction that was preempted by the trap. The first action taken by our trap analysis routine is to determine whether the trap occurred within an application that contained a Bastion-protected module. This is done by comparing the prevailing *contextID* with the *contextID* associated with each active module in the Module State Table. When none matches, the routine concludes its analysis, as the currently running application does not contain any protected module. When a match is found, the routine checks whether any of the modules within the current application are expecting a CALL_MODULE or RETURN_MODULE transition. It performs that check by looking up the transition registry of each module’s Module State Table entry.

When a pre-authorized CALL_MODULE transition is detected, the module’s private stack pointer (or the saved module zero stack pointer, when calling module zero) is read from the Module State Table and copied to the stack pointer register in hardware. The C routine then returns, the *current_module_id* is changed in hardware to the `module_id` of the callee, and the TLB miss trap handler retries the faulting instruction. Since the `module_id` has been changed to officially enter the callee's compartment, the retried instruction will not fault again due to a `module_id` mismatch. However, the retried instruction may then cause a regular instruction TLB miss, due to the absence of the appropriate translation in TLB hardware—i.e., the related page table entry is not cached in the TLB. Such a TLB miss, unrelated to compartment transitions, is handled differently by our enhanced TLB miss handler. In this case, the C analysis function in the handler determines that no pending transition exists for the currently executing module and so it returns without taking action. The rest of the TLB handler executes to generate the missing translation and insert it in hardware. As explained above, the handler code makes sure that the translation generated is legal for the currently executing module.

The C analysis routine in the instruction TLB miss handler also detects and processes RETURN_MODULE transitions. By comparing the TPC with the return address of registered RETURN_MODULE transitions, the routine determines that the current instruction TLB miss event is due to a pending return transition. The C routine then switches the prevailing stack to the stack of caller—i.e. the target of the transition. Finally, the TLB miss handler sets the *current_module_id* in hardware to the caller's `module_id`, and retries the faulting instruction to complete the transition.

Note on SPARC Register Windows: Register windows are a programming paradigm unique to SPARC. It differs significantly from the way other architectures, e.g., Intel x86, handle general-purpose register allocation. For application writers and for compilers, register windows are a useful, simplifying feature: whenever a function call is made, all caller registers can be saved in one fell swoop by invoking the `save` instruction. This saves the set of general-purpose registers that were used by the caller (a register window) and gives the callee a fresh set of general-purpose registers (another register window). To allow for passing function arguments from the caller to the callee, the two register windows just described overlap by a few registers. These registers are accessible by both the caller and callee so the former can write arguments readable by the latter. Upon a function return, the `restore` instruction can be invoked to restore the register window of the caller. The overlap that exists between the caller and the callee windows can now be used to pass return values from the callee back to the caller.

SPARC register windows provide a simple way to allocate application registers at the cost of more complex register allocation code in the operating system. For the system programmer, the SPARC register window mechanisms require saving and cleaning several sets of hardware registers upon a context switch rather than a single set, as in Intel x86. More importantly, it requires several new trap handlers to manage the spilling and filling of the finite register window stack in hardware, to maintain the impression of an infinite register window stack for software. In Bastion, spilling, filling and cleaning the register windows of a protected module cannot be done by untrusted OS trap handlers, it must be done in the hypervisor. Aside from the module itself, the hypervisor is the only entity which can be trusted with access to trusted module registers; it is also the only entity allowed access to the module's private stack, where register windows are to be spilled to and filled from.

We implemented special spill, fill and clean trap handlers in the hypervisor code base, to manage the register windows of trusted software modules within the Bastion TCB. All register windows are spilled to the module's private stack upon preemption of the module, to ensure that the intervening operating system is denied access not only to the module's current register window, but also to all the windows it has saved in the hardware-based register window stack. The register windows of a caller are also spilled to the caller's private stack upon invocation of CALL_MODULE, to ensure the callee cannot invoke `restore` to spy on its caller's register state. The only registers available upon calling of a module are the architecturally defined registers in the window overlap, where the caller and callee can exchange invocation parameters and return results.

When a trusted module runs out of register windows in hardware, the invocation of the `save` instruction triggers a *spill_window* trap. The hypervisor can intercept this trap since the OS handler to which it is naturally routed is in module zero, causing a TLB miss due to `module_id` mismatch. The same is true of *fill_window* traps that occur when a module wants to restore a register window that has previously been spilled. The *clean_window* trap, which occurs when software is allocated a hardware register window that has not been explicitly zeroed by system software (e.g., for security purposes), must also be intercepted by the hypervisor. Our instruction TLB miss handler thus contains code to detect whether a TLB miss is due to *spill_window*, *fill_window* and *clean_window* traps. This can be done by comparing the current Trap Program Counter (TPC) with the architecturally defined addresses (i.e., the OS cannot change these addresses) of the corresponding spill, fill and clean window trap handlers in the operating system. The addresses are relative to the OS trap table base address, which is available to the hypervisor in a dedicated hardware register.

7.5 Application

Bastion can run arbitrary software in module zero, without any modification to its code base. There are three aspects to running a software module inside a Bastion-protected compartment:

1) defining the software module,

2) wrapping the software module with Bastion SECURE_LAUNCH, CALL_MODULE and RETURN_MODULE hypercalls and

3) wrapping the library calls and system calls it makes to software outside the compartment.

Defining a trusted software module within an untrusted application or operating system is dependent on the security policy one wishes to enforce on the platform of interest. One needs to look at the sensitive code and data that need to be protected to reach a given security objective, and group it into a given module that is to receive Bastion protection. Interfaces to this module must be designed to enable validation—

using cryptographic means or others—of data coming from untrusted space. They must also enable protection of data written to untrusted space, when required by the security policy. The development of a formal methodology for the partitioning of software into trusted and untrusted components is an open research problem, considered outside the scope of this thesis. We refer the interested reader to the literature on the topic, e.g., [Chen and Lee 2009]. In this thesis, we assume the software developer or some trusted entity examines the code by hand to define modules and their interfaces. The sections below describe our modifications to application software enabling the execution of trusted Bastion-protected modules within untrusted applications.

Bastion supports all legacy software in untrusted space, and most legacy software wrapped in Bastion-protected modules. The main limitation of Bastion is that it cannot support code reentrancy for trusted modules. This means that once invoked with CALL_MODULE, a Bastion-protected module cannot be invoked again by another piece of software and it cannot invoke itself recursively, until the module returns using the RETURN_MODULE hypercall. This limitation could be overcome by redefining the Module State Table entries so that they can support multiple contexts for each securely launched module, e.g., multiple pending returns for the different iterations of a recursive module invocation. We leave the extension of Bastion for trusted module reentrancy to future work.

7.5.1 Security Segment

Protected modules must be defined in security segments to allow for their launch via the Bastion SECURE_LAUNCH hypercall. In future iterations of our Bastion implementation, we envision the construction of security segments to be carried out automatically during the linking phase of software compilation. Automation would require the software developer to use new type annotations mapping specific code routines and data structures to a given module. Type annotations could also be used to identify authorized module entry points as well as data structures being shared between modules. A modified linker could then map the annotated data and code to private and shared modules pages. Finally, the linker would populate the fields of the security segment with the virtual addresses of the private and shared pages for the module.

In our current implementation, we avoid having to define new type annotations and modifying the linker by constructing security segments in a less automated way. The construction of a security segment is done in the untrusted part of the application, using calls to a library that is aware of the correct format of security segment internals. This library contains functions like `init_segment`, `add_entry_point`, `add_private_range`, `add_shared_range` and `set_stack`. These calls are used to manually populate individual fields of the security segment. Symbols are passed as parameters to these functions, which get translated into a virtual address during linking of the program (trusted software modules are always statically linked). When the security segment construction code executes at runtime, these virtual addresses are added to the security segment data structure in memory, which can then be referenced during invocation of SECURE_LAUNCH. To ensure that module code and data get mapped to their own set of virtual pages, we write linker scripts (called *map files* in SunStudio parlance [Sun 2010]) that force the linker to assign separate pages to the routines and data structures forming trusted modules.

To support heap and stack allocation, we define data arrays in software that are sized to be large enough to accommodate the expected peak stack and heap needs of applications modules during runtime. These data structures are mapped to private module pages. The highest address within the data array defined to be the stack is referenced in the `set_stack` function call, to identify the top of the private module stack within the security segment. Also, the base address of the data array defined to be the heap is passed to the module's private instance of `malloc`. This ensures dynamically allocated module memory is taken from pages defined as private module pages in the security segment.

7.5.2 Hypercalls

Once the module is defined, the code invoking the software within the module must be modified to use the SECURE_LAUNCH hypercall once to launch the module, and then use the CALL_MODULE hypercalls every time it wants to invoke module code, rather than branch instructions in the core ISA. Modules must be written to return to their callers using the RETURN_MODULE hypercall, and can use the storage and attestation hypercalls to use Bastion secure storage and remote attestation services. We implemented Bastion hypercalls as a library of inline assembly code macros, following GNU inline assembly guidelines in GCC for Linux, and SunStudio's .il inline assembly file format. Using inline assembly not only allows us to invoke the `illtrap` instruction directly, it also allows us to place hypercall arguments into the registers where they are expected by the hypervisor handlers.

7.5.3 Library and System Calls

System calls are typically invoked via a wrapper library such as LibC. Such a library provides high-level functionality to software, and invokes low-level OS functionality as needed, via system calls. System call parameters are processed and formatted versions of the parameters passed to the high-level function. As described in Chapter 5, the Bastion architecture supports system calls to both trusted and untrusted system call handlers in the OS. In our implementation, we only consider untrusted system call handlers. This means we consider system call parameters as either protected, or not security-sensitive. For example, parameters to console output system calls can be considered not security-sensitive when the trusted module only displays status information that can be made public (and does not require integrity protection) according to the security policy. As another example, parameters to file I/O system calls can be considered protected when the file data sent and fetched via these calls is protected by the Bastion secure storage mechanism. Within such a security policy, the parameters to the system calls are as sensitive as the parameters to the high-level LibC function from which they were derived. Since trusted software modules in our implementation must be designed to exclude the system call handlers from their trust domain, this allows us to also exclude the related

LibC code from trusted module code base in our implementation. For a LibC call in which no system call is required (e.g., the function `isalphanum`, checking for whether a character is an alphanumeric value), we require the trusted software module to have its own copy, within its private code pages of the LibC function.

To invoke system calls via LibC functions, we wrote wrappers that ensure invocation parameters can be transferred from private module memory to memory pages shared with the untrusted module zero. LibC and system calls handlers execute in module zero so they will be able to access those invocation parameters. Any memory data allocated for return parameters is also copied into shared memory by our wrappers. Each LibC function has its own wrapper, split into two components: a trusted routine and an untrusted routine (this follows the logic presented in Figure 5.7 of Chapter 5, where a trusted software module goes through an untrusted syscall stub in module zero). For example, the `puts` system call depicted in Figure 7.6, which is invoked by the *puts* LibC function, is wrapped by a trusted *bastion_puts* routine, which calls the untrusted *module0_puts* routine. The trusted routine allocates memory within the Bastion shared page P, where it copies all invocation parameters. It then calls the untrusted routine *module0_puts*, with references to the copied parameters as arguments. Since the untrusted routine sits in the module zero compartment, this call must be carried out through an invocation of the CALL_MODULE hypercall. The untrusted routine then carries out the actual LibC call with the parameters it received. The call can be carried out without any hypervisor intervention since it does not involve a module transition: it originates from module zero and gets handled by OS code in the module zero compartment. Once the LibC call returns, the untrusted wrapper routine returns to the trusted wrapper routine by invoking the RETURN_MODULE hypercall. This triggers a transition back into trusted module code, where the wrapper routine returns. Module code can then process the results returned by the LibC call by reading them from the shared page.

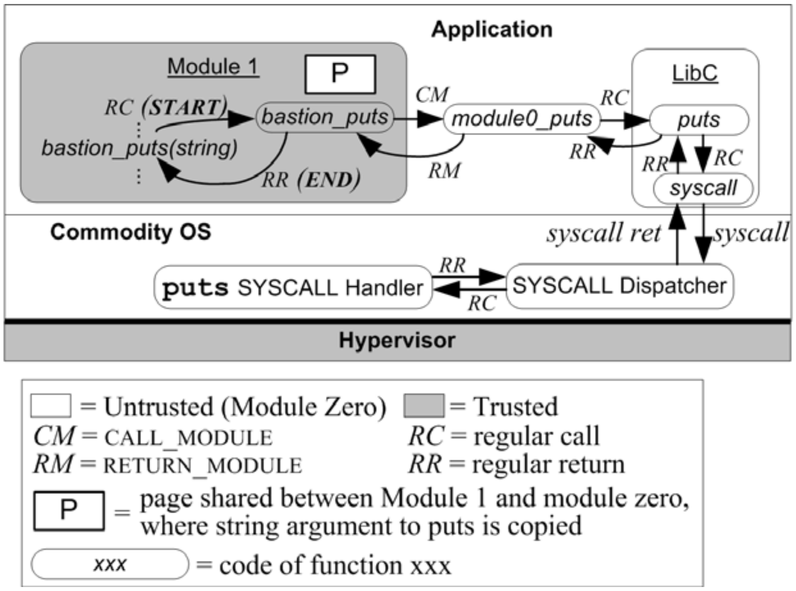


Figure 7.6. Bastion LibC Wrapper Example with *puts*

The approach described above allows for dynamic linkage of LibC to the application of interest. Because LibC code does not need to be integrated into the trusted module, its location within the virtual address space does not need to be known at compilation time—i.e., to populate the fields of the security segment. The dynamic linker can load the LibC DLL (Dynamically Linked Library) anywhere it wants within the virtual address space, without violating any Bastion compartment rules. In fact, any untrusted DLL can be linked to an application containing trusted modules without any changes to the DLL itself. If DLL functions are to be invoked from within a trusted module, a wrapper as described above must be written for each function.

One important limitation of our current implementation is that it does not support Position Independent Code (PIC) for trusted module data and code. This is due to the fact that there is currently only one Global Offset Table (GOT) and one Procedure Linkage Table (PLT) per application. Trusted modules compiled as PIC must thus share these tables, which contain very sensitive information, with untrusted code. By corrupting a GOT or PLT entry used by trusted code, the untrusted code can control where the trusted code fetches data or instructions, leading to a major security compromise. Until the compiler is modified to partition the PLT and GOT into trusted and untrusted parts, we cannot compile trusted module code as PIC. To deal with this current limitation, which is to be addressed in future work, we preclude the use of position-independent code in for trusted modules therefore ensuring that the trusted modules do not use the PLT or GOT.

7.6 Chapter Summary

In this chapter, we detailed our implementation of a Bastion prototype on the OpenSPARC platform. We modified the RTL source code of a Niagara T1 processor core and added hardware components to an FPGA-based microprocessor system to implement Bastion's hardware mechanisms. We also modified the SPARC-64 assembly source code of the UltraSPARC hypervisor running on the T1 processor to implement our Bastion software mechanisms. Finally, we describe the application-level libraries we created to implement our Trusted Programming Interface. The next chapter presents security, functionality, complexity and performance evaluations for the Bastion architecture.

Chapter 8

Evaluation

This chapter evaluates the security, functionality, complexity and performance of the Bastion architecture. We do not aim to provide a formal proof of security for Bastion. Rather, we present a security analysis that provides an informal, yet rigorous argument for why Bastion can uphold the security properties described in Chapter 2: confidentiality and integrity of protected information. We then provide an example of Bastion functionality by describing in detail an application of Bastion for three concurrently running security-critical tasks, in three separate trust domains. Our software complexity evaluation looks at the additional lines of code required in the Bastion hypervisor and in trusted application modules, as well as the increase in memory space requirements at runtime, and the increase in binary file size. To provide an estimate of Bastion's hardware complexity, we measure the amount of FPGA resources required to add Bastion hardware to the OpenSPARC full-system FPGA project. Finally, we analyze the performance impact of Bastion protection on the hypervisor and on the software in virtual machines by running a vi text editor enhanced with two trusted software modules.

While extensive security evaluations for trusted computing platforms have been done in the past—e.g., [Smith and Austel 1998]—a formal security proof requires using hardware and software design methods that lend themselves to machine-checked proofs, from gate-level up to chip-level and from assembly code up to high-level source code. Formal platform-level security analysis has been done in the past for the seL4 microkernel [Elkaduwe et al. 2008]. While these analysis methods provided formal proofs of simple security properties like pointer safety, they did not look at more advanced properties like application-level security, attestation and secure storage. Formal security proofs are outside the scope of this thesis. Their application to Bastion would require a redesign and re-implementation from scratch, to allow automated theorem proving software to provide a machine-checked proof. As demonstrated by the seL4 project, such an endeavor is a long-term project requiring many man-years of highly-skilled specialists in theorem proving software. Hence, this chapter concentrates on providing a security analysis that can serve as a framework for penetration testing and other forms of empirical security validation.

8.1 Security

We begin our security analysis with a look at the Bastion Trusted Computing Base (TCB): the processor and the hypervisor. We then reason about the security of Bastion-protected Operating System (OS) and application compartments. We divide thisdiscussion into the three main phases of trusted software execution: launch, transition and execution. Finally, we analyze the security of the Bastion attestation and secure storage mechanisms, which are used to provide secure I/O capabilities to protected software modules. The structure of our analysis follows the chain of trust established in Bastion. Upon a platform reset, the Bastion processor is the only trusted component. Its security mechanisms allow for the establishment of the software part of the Bastion TCB, the hypervisor. In turn, the hypervisor creates secure execution compartments in virtual machines, for trusted software. From within these compartments protected by the TCB, trusted software can invoke the Bastion attestation and secure storage mechanisms to perform secure I/O transactions.

8.1.1 Processor

The Bastion-enhanced processor chip is assumed to be manufactured and distributed by trusted entities, hence is considered trustworthy upon delivery to its owner. The chip is the only trusted component on system reset, hence it contains the roots of trust for all Bastion security mechanisms.

Hypervisor Launch: The root of trust for hypervisor identification is the on-chip routine invoked by the `secure_launch` instruction. It is written and verified by the trusted processor manufacturer, and stored to a non-volatile, write-once memory upon manufacture[28]. Its invocation is protected by the `secure_launch` instruction. The hardware executing the instruction ensures the routine is always invoked through its entry point, and only by software running in hypervisor mode. The hardware also checks that the routine is executed only once per power cycle. Finally, the processor hardware provides a locked down, read-write memory environment where the secure hypervisor launch routine can execute without having its code and data evicted off-chip. Trusted Bastion processor hardware thus protects the integrity of the secure launch procedure's storage and execution, and guarantees the procedure is invoked correctly, i.e., by hyperprivileged software, once per power cycle.

Memory Protection: The Bastion memory protection mechanisms on the processor chip are also protected by chip manufacturing complexity and by hardware restrictions put on their internal registers and interfaces. There are two main security-critical registers within the memory encryption and hashing engines: the *memory_key* and *tree_root* registers

[28] This is like the firmware, on certain processors, that executes instructions consisting of a sequence of micro-operations, often called micro-code. If an update capability were to be needed for the `secure_launch` or `attest` routines, one could define a Bastion version of the micro-code update mechanisms, where a new Bastion instruction would allow loading on-chip signed updates of the routine code.

containing the memory encryption key and the memory integrity tree root respectively. Both registers are volatile. The protected machine memory space is reconstructed anew on every platform reset so there is no need to keep memory keys and root hashes across reboots. The registers can only be read and written by the hardware memory protection engine themselves. The *memory_key* register is read directly by the hardware as part of memory encryption and decryption and written indirectly by the secure hypervisor launch routine, as the memory protection mechanisms are initialized. This indirect write, ultimately carried out by the engine itself, is requested by the secure launch routine via the dedicated interfaces to the memory protection engines. The *tree_root* register is read and written every time a tree hash verification or update must be carried out on a memory block fetched from or written back to external memory. The *tree_root* register can only be written by the hashing engine itself, either during the tree initialization procedure or as part of a tree update, both described in Chapter 4. Hardware restricts the interfaces to the memory protection engines to authorized software, either to the on-chip secure launch routine or to a securely launched hypervisor.

The symmetric key for memory encryption in the *memory_key* register can only be written once per power cycle. The key is generated anew on each processor reset from the entropy provided by the on-chip Random Number Generator. The root hash of the memory integrity tree in *tree_root* is also recomputed on every power cycle by the secure launch routine. Because they are stored in dedicated on-chip registers, these two hardware roots of trust for memory protection are inaccessible to software and physical attackers. The regeneration of the memory encryption key upon processor reset ensures that knowledge of (plaintext, ciphertext) pairs in one power cycle does not give an attacker information about encrypted memory contents in another power cycle. The regeneration of the root hash of the memory integrity tree for the hypervisor's memory space ensures that memory contents cannot be replayed across power cycles.

Attestation: For the purpose of this thesis, we assume the CPU private key—one of the roots of trust for attestation—is generated by the (trusted) manufacturer and written to a dedicated write-once CPU register during chip fabrication; this read-only register, named *cpu_key*, is only accessible to the on-chip attestation routine. The manufacturer is trusted to have properly certified the key using a Public key Infrastructure (PKI) accessible to all parties receiving an attestation report from the CPU. The attestation routine is trusted not to use the private key for any purpose other than signing attestation reports, so attacks stemming from dual use of asymmetric key pairs do not apply. In certain trust models, the owner of the platform may trust the manufacturer only to produce a correct chip, not to generate, insert and certify a CPU private key. In this case, the manufacturer should deliver the processor without a burnt-in private key. The owner needs to generate his own private key, insert it in the chip himself by writing the write-once *cpu_key* register and set up his own PKI certifying the public part of the key. Future work on Bastion should address this need for trust model flexibility, by offering other private key initialization schemes such as those supported by the TPM [TCG 2006]. Within the trust model presented in this thesis, the security of processor identification, a core component of attestation, is entirely dependent on the processor manufacturer, who is trusted to securely generate, store and certify the CPU private key.

To ensure the security of the attestation process, the hypervisor identity measured during secure hypervisor launch must be protected from tampering while the hypervisor is running. The hardware protects this identity by enforcing access control on the register where it is stored, the volatile *hv_identity* register. This register can only be written by the on-chip secure hypervisor launch routine and read by the on-chip attestation routine. The processor detects execution of the routine by comparing the current program counter with the range of memory addresses architecturally reserved for the routine. The *hv_identity* register is volatile so it is zeroed automatically upon a processor reset. This reflects the fact that hypervisors lose control of the software stack upon a processor reset. Therefore, the *hv_identity* register contains a non-zero value only when the Bastion platform is controlled by hypervisor software that has been launched by the `secure_launch` instruction. This guarantees that the Bastion attestation routine will only report the identity of securely launched hypervisors, notifying the remote party when no such hypervisor is in control.

The on-chip attestation routine itself must also be trustworthy to ensure the security of Bastion attestation. As for the secure hypervisor launch routine, the attestation routine is written by the trusted chip manufacturer and burnt into a read-only memory upon manufacture. The code and data of the routine are mapped to a reserved part of the machine memory space, which the hardware only makes accessible to the `attest` instruction and the attestation routine itself. Using the new `attest` instruction is thus the only way to enter the code of the on-chip routine. Once invoked, the routine is provided with a hardware-protected execution environment ensuring its runtime integrity. As for the secure hypervisor launch routine, security of the storage, invocation and execution of the attestation routine are thus guaranteed by hardware mechanisms. Correctness of the return into the hypervisor code that invoked `attest` is ensured since the CPU itself saves the return address (the instruction following the call site of `attest`) into a general-purpose register upon invocation (see Section 6.3.2, in Chapter 6).

Secure Storage: The processor also holds the roots of trust for secure storage in dedicated non-volatile registers. Hardware-enforced constraints on access to these registers ensure that they can be read and written only by authorized software. The on-chip secure hypervisor launch routine is the only software allowed to access the *storage_owner* register, while *storage_key* and *storage_hash* can only be accessed by software running in hypervisor mode. The processor detects execution of the routine by comparing the current program counter with the range of memory addresses architecturally reserved for the routine. As described in Chapter 6, the hardware enforces access control on these registers such that a given secure storage hash and key pair is only accessible to the hypervisor that created the related secure storage area. This protects the integrity and confidentiality of the registers while the platform is powered on. Offline, the integrity and confidentiality of these non-volatile registers is protected by the complexity of the chip's manufacturing process. As stated in Chapter 2, we assume this is sufficient to make these registers safe from hardware attacks.

8.1.2 Hypervisor

The security of the Bastion hypervisor relies on the processor primitives discussed in the previous section. We described how the hardware protected these primitives from hardware and software attackers. We now examine how the architecture uses these primitives to secure hypervisor launch, invocation, execution, storage and attestation.

Launch: As stated in Chapter 4, Bastion adopts the flexible measured boot approach of the TPM rather than the rigid secure boot approach taken by certain platforms, e.g., [Arbaugh et al. 1997]. Therefore, secure hypervisor launch does not mean booting a hypervisor known to be good, as identified by a whitelist. It means booting a hypervisor that is precisely identified (i.e., measured) by the hardware, such that it can be reported to any entity that needs to make a trust decision about the platform. This measurement of the hypervisor must be computed by a trusted component, and stored in a secure area. In Bastion, measurement of the hypervisor is carried out by the trusted on-chip secure launch routine, in a protected on-chip environment. The resulting measurement is stored in the hardware-protected *hv_identity* register. In addition to this TPM-like boot-time security, the Bastion hypervisor launch must establish a secure memory space to enable detection of runtime corruption. Initially, this secure memory space—hashed and encrypted by on-chip engines—must be identical, bit for bit, to the hypervisor state measured into *hv_identity* by the secure launch routine. This is necessary to avoid Time Of Check to Time Of Use (TOCTOU) attacks, where the hypervisor reported by the attestation mechanism differs from the hypervisor whose runtime integrity was enforced by hardware.

In Bastion, the correspondence between the contents of *hv_identity* and the initial state of the secure memory space is made possible by the collaboration of the secure launch routine and the on-chip memory protection engines. The secure launch routine notifies the engines, via their internal interface, that it is about to start computing the hypervisor identity. The engines then establish the initial secure memory space using the data that is being fetched by the on-chip routine following this notification. Therefore, the exact same data, fetched at the exact same time, is used by both the on-chip routine and the on-chip engines to compute, respectively, the hypervisor identity and the initial state of the secure memory space. The Bastion hypervisor launch thus achieves the required security properties:

1) the hypervisor is measured by a trusted entity,

2) the measurement is stored in a secure area and

3) the measurement corresponds to the initial hypervisor state upon which memory integrity verification is to be carried out at runtime.

Upon reset, the Bastion processor jumps to an untrusted hypervisor loading routine; hence `secure_launch` is not the first instruction to execute. In fact, it may not get executed at all. However, the semantics of Bastion secure launch, secure storage and attestation are such that security is maintained even if the untrusted loader skips `secure_launch` or loads a corrupted hypervisor. In such scenarios, the *hv_identity* value either does not get computed at all or it is computed over the corrupted hypervisor. A good hypervisor's secure storage area thus remains locked, since it is tied to the good hypervisor's identity in *storage_owner*. This ensures that no other software can read or modify the information in the hypervisor's secure storage. In turn, this ensures a bad hypervisor is unable to corrupt without detection the secure storage areas of trusted software modules.

Invocation: Securing hypervisor invocation means controlling the initial entry into hypervisor code as well as re-entry into the hypervisor at runtime, either via interrupts, exception traps or hypercalls. The initial entry into hypervisor code is performed by the secure launch routine itself, as it completes its execution. The entry point used then is architecturally defined—and thus burnt in the routine code directly—to be the address of the hypervisor's processor power-on reset vector. This is the only way to carry out an initial entry into the code of a securely launched hypervisor. During hypervisor runtime, as virtual machines take control of the CPU, some events may trigger re-entry into hypervisor code. Regardless of the event (interrupt, exception trap or hypercall), the hardware itself initiates jumps into hypervisor code, at an entry point determined by a hypervisor trap table vector. In no way can software in virtual machines choose where to enter hypervisor code: the architecture makes trap events the only mechanism that can cause a privilege switch to hyperprivileged mode. Since the hypervisor entry point addresses contained in trap vectors are covered by the memory integrity tree, runtime invocation of the hypervisor is protected against entry point corruption.

Execution: Hypervisor execution is protected by the Bastion machine memory protection mechanisms, which guarantee the integrity and confidentiality of hypervisor code and data. Every cache line of hypervisor state is encrypted by the on-chip encryption engine and hashed by the hardware-rooted integrity tree mechanism. Secrets manipulated by the hypervisor during runtime are thus protected from snooping and corruption by hardware adversaries. The hypervisor memory space can only be modified by the hypervisor itself; any modification by a hardware adversary will be detected as soon as an affected hypervisor cache line is fetched on-chip. To protect hypervisor execution against software attacks, Bastion relies on the existing privilege level mechanism in the hardware, which gives the hypervisor full control over hardware resources. The hypervisor can thus make sure its memory space is unreachable by VM software, and control the way interrupts, traps and other exceptional events are processed. This thesis does not consider protection of the Bastion hypervisor (or modules) across the different power-saving sleep states supported by modern microprocessors. An example of architectural support for secure sleep state transitions can be found in [Caceres et al. 2005, Kumar et al. 2008]. The main idea in this work is to save an encrypted and hashed blob of the memory contents upon entering a sleep state, which is decrypted and verified upon exiting the sleep state.

Storage: Hypervisor secure storage is a direct application of the hardware secure storage primitive analyzed in the previous section. The hypervisor allocates a data structure within its protected memory space and populates it with sensitive information. To make the data structure into a secure storage area, it encrypts and hashes the data and stores the key and hash in the hardware registers. This effectively roots the confidentiality and integrity of the area in the trusted hardware, and binds the area to the hypervisor's identity. Regardless of where the encrypted and hashed area is stored, both software and hardware adversaries are unable to decipher it or tamper with it without being detected. Any hypervisor with an identity differing from that of the area's creator will also be unable to read or modify the area. The hardware-rooted storage mechanism thus provides tamper-evidence, confidentiality and authentication for the hypervisor secure storage area.

Attestation: The Bastion hypervisor can attest to its identity by invoking the new `attest` instruction. The instruction takes two operands: a pointer to an attestation certificate and a pointer to a memory area where the signed attestation report is to be written back by the processor's on-chip attestation routine. The authenticity of attestation reports is ensured by the routine's use of the CPU private key in signing the reports. The integrity of attestation reports is ensured since they are computed from protected data—i.e., the certificate from protected hypervisor memory and the hypervisor identity and CPU private key from hardware-protected registers—by a trusted attestation routine running in a hardware-protected execution environment. The reports are written back to protected hypervisor memory referenced with the second pointer operand to the `attest` instruction.

As in most security protocols, it is crucial that the Bastion attestation protocol includes provisions to defend against replay attacks. In a replay attack, an attestation report produced for remote party A at time t is replayed to A at time $t+\Delta$ to trick A into thinking the platform runs the same software at $t+\Delta$ as it did at time t. To defend against replay attacks, freshness must be introduced into the attestation protocol to ensure the uniqueness of each attestation report. As described in Chapter 6, A generates an unpredictable nonce and asks the hypervisor to include it in the certificate passed to `attest`. By including the certificate data into its digital signature, the on-chip attestation routine guarantees the uniqueness of the attestation report and thus prevents replay attacks. In addition to preventing replay attacks, a secure attestation protocol must also allow for binding of a computation result to an attesting entity. In other words, the attestation protocol must enable authentication of data produced by attesting software. In Bastion, this can be done upon production of each data result, by invoking the `attest` instruction with the result in the certificate. As explained in Chapter 6, the certificate can also be used to establish a secure communication channel between the attesting software and the remote party, allowing for authentication of many data results with a single invocation of `attest`.

Retirement: In this thesis, we assume that once launched, the Bastion hypervisor remains in full control of the CPU until the next platform reset. Therefore, there is no need for an explicit secure hypervisor retirement procedure. Hypervisor retirement occurs upon processor reset, when all volatile registers are zeroed. This includes the root of the memory integrity tree and the memory encryption key. Zeroing these registers ensures that:

1) persisting encrypted hypervisor memory in one power cycle cannot be decrypted by the platform in the next power cycle;

2) hypervisor memory state cannot be replayed across power cycles.

8.1.3 Module Compartments

This section analyzes the security of Bastion-protected OS and application compartments. We divide our analysis into the different phases of module execution: launch, invocation, execution, secure storage, attestation and retirement. All phases rely on the security of the hypervisor and processor forming the Bastion TCB. We use the term critical resources to refer to both non-volatile resources (the secure storage areas) and volatile resources (Bastion-protected private and shared module pages, and the data they contain)

We note that due to its measured boot approach, the Bastion platform may launch and protect a malicious hypervisor. In this case, the security properties of module execution may not be properly enforced by the hypervisor. However, this does not violate the security objectives of the Bastion architecture. In the Bastion trust model, security-critical resources like cryptographic keys and personal data are bound to good software module running on good hypervisors, as defined by the parties providing the security-critical resources to the platform. This binding is done with both persistent resources in secure storage and volatile resources exchanged during a run of the Bastion attestation protocol. In both cases, a software stack containing a corrupted hypervisor will not be given access to critical resources. For persistent resources, the on-chip secure launch routine detects a discrepancy between *hv_identity* and *storage_owner*. For volatile resources, the remote party inspects the attestation report signed by the trusted CPU to determine, before sending any critical data, that a compromised or unknown hypervisor is in charge. The same argument applies to modules trying to use the Trusted Programming Interface (i.e., Bastion hypercalls) on a platform where no hypervisor is running.

Launch: As for the hypervisor, software modules are launched with a measured boot approach rather than secure boot. Any module can thus request Bastion protection via the SECURE_LAUNCH hypercall. The hypervisor precisely measures the identity of the module being launched, but it does not check this identity against a whitelist of modules known to be good. Modules are provided with the protection they request via their security segment, which is included in the computation of the module's identity. The hypervisor ensures protection to any securely launched module, but it does not unlock critical resources unless these resources—e.g., a secure storage area, a shared memory interface—are explicitly bound to the module's identity. This ensures that only a properly protected module free from corruption can access the critical resources it is associated with. Compromised, unknown or improperly protected modules can create a new secure storage area or open up a new shared memory interface, but they cannot access existing critical resources, which have not been explicitly associated to their identities. Computation of the module identity and unlocking of critical resources is performed by trusted hypervisor code, to ensure the correctness of the process. The computed module identity, to be used in future module attestation requests, is securely stored in protected hypervisor memory to ensure it cannot be tampered with. The hypervisor's secure module launch procedure guarantees that this identity was computed with the same data space that was used to initialize the module's share of the memory integrity tree.

Invocation: Modules can only be invoked via the CALL_MODULE hypercall, with entry points specified in the security segment provided upon launch. The hardware is the ultimate enforcer of these controlled transitions: it rejects any jump into module code that is not sanctioned by the hypervisor (with a *current_module_id* switch). The point of re-entry into module code following preemption by the OS is also validated by the hypervisor and enforced by hardware. Allowing the use of unauthorized entry points on module invocation or re-entry would open the door to a wide range of attacks. For example, an attacker could change an entry point so module code is entered past a critical security check. This would allow the attacker to violate the security policy that the module is expected to enforce. This is why in Bastion, authorized entry points are part of module identity, like everything else in the security segment. This means that critical resources are unlocked only if the module's entry points are properly set. It also means that remote parties can make their trust decisions based not only on the module's initial state and its memory protections, but also based on how the platform controls entry into the module during runtime.

Execution: The confidentiality and integrity of critical module execution are protected against hardware adversaries by our machine memory protection mechanisms, and against software adversaries by our Shadow Access Control mechanism. Both schemes are set up and maintained by the hypervisor, and enforced by the hardware. Software in virtual machines and hardware adversaries are denied access to the metadata—i.e. contents of the Module State Table, vMap and mMap—used by the hypervisor to configure the hardware part of the schemes. They are also unable to interfere with the mechanisms directly in hardware. VM software is prevented from any interference by the privilege ring mechanism, which prevents access to hypervisor memory space as well as Bastion registers and crypto engine interfaces in hardware. Machine memory protection prevents hardware adversaries from accessing hypervisor memory space. On-chip components are also safe from hardware attacks as per our threat model. As a result, secure module launch sets up a protected execution environment whose integrity and confidentiality cannot be breached by hardware or software attackers.

Secure Storage: Securing a module-specific storage area consists in guaranteeing that its contents cannot be revealed or modified by a module other than the one that created the area, i.e., the owner. Since module identification in Bastion depends on the hypervisor, whose security depends on the correctness of the processor, secure storage areas are bound to both a module identity and a hypervisor identity, and the binding is ultimately protected by the processor-anchored storage. If any hypervisor or module bit changes, the secure storage area remains locked. If a genuine hypervisor running a genuine module tries to unlock a secure storage area on a processor other than the one where it was created, it fails. As described in Chapter 6, storage areas can only be migrated via a trusted third party, which allows a source processor to migrate secure area content to another processor's area. Having two valid copies of a given secure storage area in two different locations may violate the security policy of certain modules. In these cases, storage migration should be precluded altogether. Definition of an elaborate migration and backup scheme that allows specifying certain data as non-migratable (e.g., as in TPM) is outside the scope of this thesis.

Attestation: As with all module security services, module attestation relies on hypervisor security services. The module itself is only responsible for generating the certificate passed as an argument to the ATTEST hypercall. Depending on the requirements of the remote attestation party, this may include nonces for protocol freshness, data results to be bound to the module's identity or cryptographic keys for establishment of a secure channel with the module. The hypervisor creates its own certificate, which binds the module's certificate with the identities of modules to be attested to. Every one of these identities was measured during an invocation of SECURE_LAUNCH, and corresponds to an active (non-retired) module that is provided with machine memory protection and Shadow Access Control. These identities specify the initial state of critical modules, authorized memory interfaces and memory protection. By this reporting, the platform gives the remote party a guarantee that the data dynamically generated by the module, as well as the interactions it initiates with surrounding software, respect the module programmer's intention. In other words, when the program is properly protected, it behaves as can be expected from inspection of its source code.

This thesis focuses on attacks originating from outside a critical module: it does not consider software vulnerabilities located inside a critical module's code and that could thwart the remote party's security objectives. The remote party is responsible for ascertaining that a critical software module is free of vulnerabilities. This should be a feasible task for small modules with a few thousand lines of code. An interesting avenue for future research is to define a form of attestation that allows a remote party to determine whether a large piece of code (that is trusted but potentially contains exploitable software vulnerabilities) has been compromised in a way that affects the remote party's security objectives.

In addition to a public encryption key for secure communications, the certificate generated by a critical module during a typical attestation run binds the module's identity with any data required for the remote party to make a trust decision (e.g., the contents of a configuration data file). The CPU signs an attestation report containing this certificate, critical module identities, the hypervisor identity and a nonce provided by the remote party. This report provides the remote party with the (authenticated) information needed to make a trust decision. The nonce ensures that runs of the protocol cannot be replayed. The hypervisor identity and CPU signature identify the TCB (hypervisor and CPU hardware) attesting to the requester.

Retirement: The SECURE_RETIRE hypercall iterates over the module's virtual pages in the hypervisor's vMap, dMap and mMap data structures to remove references to the module's `module_id`. In doing so, the hypervisor also asks the on-chip memory integrity engine to replace, in the integrity tree, the page roots of integrity-protected pages with a null page root. This effectively prevents other software from accessing the plaintext of the module's private encrypted pages, even if the ciphertext is still in machine memory. As detailed below, because Bastion requires that every encrypted page also be integrity checked (i.e., `c_bit` = 1 requires `i_bit` = 1), removing a page P's page root from the integrity tree means the page can no longer be decrypted, even if it is later mapped to some other module M's memory space[29].

There are only three ways for module M to map the encrypted P in its memory space: 1) upon launch, by requesting P as one of its encrypted pages (c_bit = 1), 2) during runtime, by substituting one of its already encrypted pages (c_bit = 1) with the already encrypted P, or 3) at any time, using P as an unencrypted page (c_bit = 0). The third option clearly precludes M from reading the plaintext P, as accesses to the encrypted P are not decrypted by the processor engines. The first option also precludes plaintext accesses to P since adding a page to M's memory space on SECURE_LAUNCH causes the encrypted P to be encrypted a second time. Therefore, read accesses to the doubly encrypted P, which cause the processor to carry out a single decryption, yield encrypted and hence unintelligible data for M. The second option, runtime substitution of an already encrypted page for P, is only concerned with substitutions occurring via a software or physical attack. Any substitution via legitimate software means requires the page to already be mapped in the M's memory space, a scenario covered by either option 1 or 3. An attacker substituting an existing page for P cannot reflect the substitution in the Bastion integrity tree, since updates to the tree are only triggered for legitimate page accesses (i.e. page accesses requested by via legitimate software). Therefore, any access to P following the substitution causes an integrity verification error. The secure retirement procedure thus allows for reuse of the retired module's module_id and makes all confidential module pages undecipherable.

8.1.4 Defense against Concrete Attacks

In this section, we argue that Bastion can defend against the concrete attacks presented in Section 2.3 of Chapter 2. Hence Bastion provides greater security than the real-world commercial security platforms that are vulnerable to these attacks.

Cold Boot Attack: This attack is based on the fact that TPM-based systems send secret key materials in plaintext for storage in hardware memory chips, whose state can be made to persist across reboots. Bastion defends against this class of attacks by providing security-critical software with an encrypted memory space where they can perform sensitive operations and store secrets. This memory space is inaccessible across reboots, since a processor reset causes the prevailing memory encryption key to change, and therefore makes any persistent memory state undecipherable.

TPM Reset Attack: This attack exploits the decoupling between the TPM and the processor it protects. Software measurements stored on the TPM can be wiped out with a spoofed reset even though the measured software is still in control of the CPU. This cannot happen in Bastion since the integrity measurement and storage logic is tightly coupled with the runtime execution protection mechanisms. The hypervisor identity can only be wiped out of its dedicated processor register on a processor reset. When this

[29] Although we do not discuss covert channels in this thesis, we note that overwriting P with zeroes could prevent certain covert channels when machine pages are recycled across VMs.

happens, the hypervisor loses CPU control (due to the reset), it is locked out of its secure storage registers (because the hypervisor identity register is wiped out so it no longer corresponds to the storage owner) and its protected memory space becomes inaccessible (because the memory encryption key changes on a reset).

Blue Pill Attack: The Blue Pill attack relies on a claim that virtualization cannot be detected and therefore stealth hypervisor-based rootkits can be deployed. The goal of this rootkit is to install malware on the platform, to corrupt and snoop on critical application and OS state. When a Bastion hypervisor is already deployed, the Blue Pill attack fails because a malicious Blue Pill hypervisor cannot be installed (`secure_launch` can only be invoked once per power cycle). When no Bastion hypervisor is running, a malicious Blue Pill hypervisor can be deployed, but it cannot access the secure storage areas of Bastion-protected OS and application modules, which are bound to the identity of a good hypervisor. In addition, any remote party about to provide the platform with more sensitive data can detect the presence of the malicious Blue Pill hypervisor by requesting a Bastion attestation report, which automatically reports the presence of the Blue Pill hypervisor (if it was deployed using `secure_launch`) or the absence of a securely launched hypervisor. In both cases, the remote party refrains from sending sensitive data to the platform, since it does not run a full Bastion TCB. We note that if a Blue Pill hypervisor was to include randomness into its initial memory space to thwart signature detection mechanisms, a remote party might not be able to infer with certainty, from the attestation report, that the running hypervisor is indeed a Blue Pill hypervisor. Nevertheless, the identity of this obfuscated Blue Pill will not be in a list of “known to be trusted” hypervisors so the remote party will also refrain from sending sensitive data to the platform in this case.

SMM Attack: By definition, the SMM attack is not possible on Bastion since we define the hyperprivileged execution mode as the most privileged CPU mode. However, we note that if there was a System Management Mode on Bastion, we could install an SMM monitor using an adapted version of our secure launch instruction routine, which is stored in a non-malleable memory on the processor chip. This would make the SMM monitor installation procedure less vulnerable to attacks than the BIOS code—stored in an easily accessible PROM chip—responsible for this installation on traditional platforms.

XBOX Attack: This attack relies on two security weaknesses of the XBOX security architecture. The first is that an encryption key is stored in plaintext outside the processor, who needs it to decrypt the boot sector. The second is that the integrity of the boot sector, which dictates what security policy to apply on software loaded next, is not checked by the processor. In Bastion, the integrity and confidentiality of the initialization procedure (i.e. `secure_launch`) are protected by virtue of being stored and executed directly on the processor chip, outside the reach of attackers.

Cipher Instruction Search Attack: The main goal of this attack is to create an encrypted program that is accepted as genuine by the processor. The attack is feasible because, as for the XBOX security architecture, this processor’s encryption mechanism is not accompanied by an integrity-checking mechanism. Therefore, an attacker who eventually learns about some plaintext-ciphertext pairs can use his knowledge to build his own encrypted program and feed it to the processor. In Bastion, memory encryption can only be activated for pages that are also protected by the hardware-rooted integrity tree. Therefore, this attack cannot be carried out successfully on a Bastion processor.

8.2. Functionality

In this section, we demonstrate the flexible security functionality of Bastion by describing a hypothetical software stack where three different security-critical tasks must coexist: Digital Rights Management (DRM), personal banking and distributed computing. We build the software stack with a single OS hosting all three tasks. There are several principals in this scenario: the user, his bank, media distribution companies, and an internet server managing the distributed computation. Each principal has its own security policy, which may differ from that of other authorities. Bastion is not tied to a particular security policy, e.g., the Biba integrity model or the Bell-LaPadula Multi-Level Security (MLS) model; it simply protects the trusted software modules defined by the authorities. Within the Bastion-protected environments, and using the Bastion secure I/O primitives, these modules can then enforce the security policy defined by their principal. This policy can be hard-wired as part of each module's code base, or it can be provided dynamically to the module, as a configuration file.

Sections 8.2.1 to 8.2.3 describe the three secure computing tasks in our hypothetical software stack. In Section 8.2.4, we describe a real-world security-critical task split into two modules, enforcing an ORiginator-CONtrol (ORCON) security policy within a real-world application, the vi text editor. The two security-critical software modules involved were written for the purpose of illustrating the secure application partitioning methodology proposed in [Chen and Lee 2009]. We ported these modules to the Bastion Trusted Programming Interface, and we ran them within the Legion simulator. The related complexity and performance analyses are presented in Sections 8.3 and 8.4.

8.2.1 DRM

To illustrate the application of Bastion to digital rights management, we choose a media player application managing a library of copyrighted media files (e.g., MP3 songs, AVI movies). Media files can come from a variety of online media content providers, each with its own distribution server. For simplicity, we assume that our user only deals with Sony and Disney. Both companies have the same security objective: protect the integrity and confidentiality of the media files they distribute so that only paying users are able to view them. However, the two companies are mutually suspicious so they do not want to entrust the management of their DRM-protected files to a common piece of software. They each have written their own DRM module, which they entrust with their DRM keys and enforcement of their file viewing policy (e.g., unlimited viewing of purchased songs, three-day viewing window for "rented" movie files). Both modules can be loaded as plug-in extensions in the user's media player, a large application with many functions that are not security-critical, e.g., Graphical User Interface (GUI) creation and maintenance, management of non-copyrighted materials, retrieval of metadata such as disk covers and movie ratings. The Sony and Disney DRM modules thus run in the same virtual address space, along with lots of non-security-critical, untrusted media player code.

Each DRM module has three security objectives: (1) receive protected media files from the Sony or Disney server, via the network interface, (2) protect the media files stored on the local platform and (3) enforce the viewing policy on files requested by the user. Reception and storage of media files are carried out using an untrusted network and an untrusted disk respectively. However, secure media output to the user is only possible with trusted I/O endpoints for the display and the sound card. Without these, media contents can be copied by an attacker at the point of rendering. We thus assume that the local platform is connected to audio and video rendering hardware that is trusted by both Sony and Disney to respect their copyrights, i.e., the output peripheral will not record media playback or allow physical attackers to snoop on its internal buses. This is an assumption similar to that made by technologies such as High-Definition Content Protection (HDCP) [Lyle 2002] or BluRay's broadcast encryption scheme [Intel et al. 2006], where a certain set of media output hardware devices are trusted by the content distributors.

To get from a raw media file to the output hardware, we need decoder routines to create audio and video frames, and output device drivers to convert these frames into a signal that can be understood by the output hardware. Media codec libraries required to decode the media files are loaded as part of the media player's address space. The operating system hosts the device drivers for this trusted media output hardware. Each company has to define a set of trusted codec libraries and output device drivers. This can be done with the help of some trusted third party responsible for inspecting the codec and driver code. This party, e.g., Verisign [Verisign 2009], could then define and certify a set of codec and device driver modules expected on trustworthy user machines. The Bastion architecture presented in this thesis does not support sharing of modules across trust domains. This means that even if Sony and Disney happen to trust the exact same codec library or device driver, there still needs to be separate copies of the modules for each company. By definition of a module identity in Bastion, two copies of a given module have a different identity when they are part of different trust domains. This allows for distinguishing between the two copies at launch, runtime, and in attestation reports.

For simplicity, we assume the user has already completed all commercial transactions related to buying or renting the media files he wishes to access. As part of these transactions, the user was identified in the Sony and Disney databases by the public key of his Bastion processor. This public key will be used by the two companies to identify the user as a paying user. The user must then download the Sony and Disney DRM modules before any protected media can be retrieved and played. This has to be done via an unauthenticated internet connection, since initially, there is no root of trust on the user's platform that would allow authentication of the servers. There is thus a possibility that the downloaded modules are corrupted or unauthorized. This is not a problem, however, since the company servers will check the identity of these modules before sending any media files to the platform. The same applies to codec and output device driver modules in application and OS space.

The Sony trust domain is thus composed of the following modules: DRM_Module$_{\text{Sony}}$, Audio-Video_Codec$_{\text{Sony}}$, Video_Driver$_{\text{Sony}}$ and Audio_Driver$_{\text{Sony}}$. Similarly, the Disney trust domain is composed of the following modules: DRM_Module$_{\text{Disney}}$, Audio-Video_Codec$_{\text{Disney}}$, Video_Driver$_{\text{Disney}}$ and Audio_Driver$_{\text{Disney}}$. (Sony also needs to trust at least one Bastion hypervisor and an entity certifying the public key of the Bastion processor.) In both cases, the DRM module has a shared memory interface with the audio-video codec, which in turn has memory interfaces with audio and video device drivers in the operating system. The DRM module has an interface to module zero in order to receive playback requests from the GUI in the untrusted part of the application, and to send and receive file data via the untrusted file system and network stacks. The device driver modules also have interfaces to module zero in order to access file system data. The existing system call mechanism (see Chapter 5) is used by the codec to invoke the drivers, and drivers can communicate with the trusted output hardware using DMA (see Chapter 6).

We assume the modules in both trust domains follow the exact same procedures for establishing the initial key material. For simplicity, we will only discuss Sony as an example. There are three sets of keys that are needed by the modules in the Sony trust domain:

1) keys to decrypt and authenticate Sony media,

2) keys to establish secure I/O channels with the audio hardware and

3) keys to establish secure I/O channels with the video hardware.

To obtain these keys the first time the modules are installed and loaded on the platform, the DRM module requests from the hypervisor an attestation report identifying all the modules in the Sony trust domain. This report, bound to a public key generated on-the-fly by the DRM module, is sent to the Sony server, over an untrusted connection. Sony identifies the report as coming from the Bastion processor of a paying customer and determines that the modules identified are trustworthy[30]. As a result, Sony accepts to send the required keys to the platform. These keys are sent over a secure communication channel, encrypted with the public key generated by the DRM module. The DRM module decrypts the keys using the private key it generated as a counterpart to that public key. This happens within the module's protected memory space so snooping by the untrusted part of the software stack is prevented by Bastion Shadow Access Control.

Using its secure shared memory interface, the DRM module sends the device driver keys to the drivers, via the codec. The DRM and device driver modules can then use the Bastion secure storage mechanism to store the keys they just received in confidential, tamper-evident persistent storage. As described in Chapter 6, this requires an interface to the untrusted file system in order to send and receive encrypted and tamper-evident files. Storing keys in persistent storage ensures that the key establishment procedure described

[30] In this thesis, we do not consider privacy. Therefore, we do not consider platform anonymity as a security requirement.

above only needs to occur once. Once the key materials are in place, the DRM module can request media files from Sony servers via the untrusted network stack running within the untrusted operating system. The files obtained from untrusted space are authenticated and decrypted by the DRM module, within its protected memory space, using the keys initially obtained from the Sony server. The media files can also be stored encrypted on the untrusted disk, where they can subsequently be retrieved for decryption and playback.

Decrypted file contents are sent to the trusted codec library via the Bastion protected shared memory interface (encrypted and integrity verified by the Bastion processor). The codec then decodes the files and sends their audio and video components to the respective device driver in the operating system, via a system call. The data is passed via a buffer located in Bastion-protected memory shared between the codec module and a trusted device driver module. The drivers format the data to be compatible with the hardware, and wrap them with encryption and Message Authentication Codes (MACs), using the keys obtained from Sony. This encrypted and MACed data is then sent over unprotected buses to the trusted hardware, which decrypts them, authenticates them and renders them to the user. With respect to the new security applications mentioned in Chapter 3, the example above can be seen as an incarnation of a hardened extension, where the media player plug-in enforcing a DRM security policy is protected against surrounding software, including the OS, and hardware attackers.

8.2.2 Personal Banking

The user of the platform has a much bigger stake in protecting the confidentiality of his personal banking information, and the integrity of his transactions than he had in protecting copyrighted media files. In this application of Bastion, we need a trusted keyboard input mechanism to ensure the authenticity of user inputs, which can direct a personal banking application to display information or carry out a transaction with the bank server. There are two application space modules in this scenario: a main module in a banking application and a secondary module loaded as an internet browser plug-in. The main module is part of a larger application such as QuickBooks [Ivens 2008], containing functions that are not security-critical, e.g., financial tips. The user interacts with this main module whenever he wants to input new financial information or display existing information. In the operating system, trusted keyboard input and display drivers allow for communication with the trusted I/O endpoints, i.e., an enhanced keyboard such as in [McCune et al. 2006] and the trusted display endpoint used for the DRM application. Peripheral hardware can be accessed by multiple display drivers using different function interfaces (each driver uses a separate device function, see Chapter 6).

As for the DRM application, the remote server—the bank's server in this case—is responsible for distributing secret key material. After determining via attestation that the banking modules and the I/O drivers are trustworthy, the bank server sends the keys necessary for the device drivers to securely communicate with the trusted I/O endpoints. As part of attestation, the banking modules also exchange keys with the server, to be used in establishing a secure communication tunnel between the server and the browser's banking plug-in. We note that in the DRM and banking applications, the keys used for secure I/O do not necessarily have to come from the remote server. As explained in Chapter 6, with a public key embedded in its data space and using the attestation mechanism, the device driver and the peripheral hardware it communicates with could exchange session keys as part of a cross-authentication protocol.

In this application, the user interacts with his bank via the web browser. Requests for transactions are initiated by the user on the keyboard, handled by the corresponding device driver and sent up to the plug-in module via a shared memory interface between the device driver and the banking module. Banking data exchanged between the server and the banking plug-in are sent via the encrypted and authenticated channel established during attestation. The user may also interact offline with the banking application rather than directly with the bank. In this case, the trusted input driver sends keyboard strokes to the trusted banking application's module rather than the plug-in, again via a Bastion-protected shared memory interface. The banking application module collaborates with the web browser plug-in via shared memory: data retrieved from the bank server by the latter can be archived by the former, and data provided offline to the former can be committed online by the latter. As in the DRM application, both modules can securely commit their sensitive data to local persistent storage using the Bastion secure storage mechanisms. Again, in terms of the new security applications presented in Chapter 3, the banking application and browser plug-in modules in this example can be seen as hardened extensions. This time, however, they are focused on protecting the user's data rather than enforcing a remote authority's security policy as in the DRM example.

8.2.3 Distributed Computing

For our third application example, we consider an internet telephony service, P2Pcall, where client devices participate as nodes in a peer-to-peer network forming a distributed routing infrastructure. Users can place calls and chat with contacts using the service's Voice-over-IP (VoIP) application. The application runs a Secure Routing Module (SRM) which accesses the P2Pcall network to inject the user's outgoing VoIP packets, extract the packets directed to the user, and route packets of other P2Pcall users. P2Pcall allows trustworthy SRMs to access the peer-to-peer network by giving them a session key K with which they can open secure communication channels with nearby nodes. The P2Pcall security policy defines a trustworthy SRM as a module routing calls according to the P2Pcall routing algorithm, without modifying any call packets. Availability of a given SRM is not a concern for the server, as the P2Pcall routing algorithm can circumvent unresponsive nodes without dropping call packets.

The software stack contains the VoIP application, with a voice acquisition module, an SRM, a chat module and a main module—all but the SRM are untrusted. The SRM uses an OS-based library of cryptographic functions to establish and maintain secure channels. On receiving an attestation request, the platform reports to the P2Pcall server the identity of the hypervisor, the SRM and the OS-based crypto module. The report could also include a configuration data file (e.g., specifying the protocol parameters or packet format to be used on the P2Pcall network) in case the server needed this information to decide whether to trust the node. The operating system's network stack, a rather large and complex software system, is not included in the attestation report, as it only performs a benign task: relay network packets, which have an encrypted and authenticated payload, to and from the SRM. In the worst case, a compromised network stack merely drops the packets or corrupts them in a tamper-evident way. This results in a denial of service for the affected node. As stated in our threat model, the architecture we propose does not try to address denial of service attacks.

Upon receiving the attestation report, the P2Pcall server determines whether it trusts the reported software to execute the routing protocol according to its security policy. If so, it sends the key K to the SRM, wrapped with an SRM-generated public key. K can then be used by the SRM to open secure communication channels with nearby P2Pcall nodes. The SRM in this example can be seen as a protected monitor, as introduced in Section 3.4.2 of Chapter 3. Reading packets from or writing valid packets to the network can only be done through the SRM, i.e., the SRM monitors network accesses, despite the surrounding untrusted software stack.

8.2.4 ORCON vi[31]

The implementation presented in this thesis is an experimental proof-of-concept. The current Bastion hypervisor code base is not stable enough to allow for development of complex trust domains in large applications within a reasonable amount of time. Therefore, we did not attempt to implement the complex usage example described above. For the purpose of evaluating Bastion with a real-world application, we chose to use the simple vi text editor, enhanced with two trusted software modules—one enforcing an ORCON policy on protected files, and the other interfacing with a PKI for the purpose of user authentication. Application of Bastion to more complex trust domains is left to future work, when a stable second-generation Bastion hypervisor should enable streamlined development and debugging.

The work in [Chen and Lee 2009] augmented vi with two trusted software modules, an *ORCON module* and a *PKI module*, allowing for enforcement of an ORCON access control policy on text files created with the enhanced vi software. The ORiginator CONtrol (ORCON) security policy consists of giving the creator of a document control on who can access a document and with which rights, even after the document has been distributed and potentially copied to several untrusted systems. Recipients of a document have either no rights to the document, read-only rights or read and append rights. Restricting write-access to an append-only policy ensures that the original content can never be deleted or falsified.

The trusted ORCON module interfaces with the rest of the (untrusted) vi application to allow a user to create a new ORCON file from data entered into the regular vi buffer, using regular vi commands. In our Bastion implementation, the vi buffer is thus located in memory shared between module zero (containing the untrusted vi code) and the trusted

[31] We are very grateful to Yu-Yuan Chen for providing us with the source code of his implementation of the security-enhanced vi text editor enforcing the ORCON access control policy.

ORCON module. Each ORCON file is associated with a policy file containing the list of recipients allowed to access the file (identified with their respective public keys), and the access rights granted by the originator to each recipient. Prior to requesting access to a file, a recipient must authenticate himself to the PKI module, which is in charge of retrieving the recipient's private key (from Bastion-protected secure storage in our implementation), or generating a new public/private key pair for a recipient who is not yet associated with a pair. In a network-connected platform, this PKI module would be in charge of interfacing with an online PKI to authenticate the user. In this implementation, the user is authenticated locally by providing a passphrase.

The ORCON module collaborates with the PKI module to check whether a recipient requesting access to an ORCON file has been properly authenticated. To do so, the ORCON module generates a random challenge and encrypts it with the recipient's public key, obtained from the policy file. The encrypted challenge is then provided to the PKI module via a shared memory interface. The PKI module decrypts the challenge with the recipient's private key and sends the decrypted challenge to the ORCON module, to prove that the user has been properly authenticated. Hosting the two modules in separate Bastion compartments allows the PKI module to collaborate securely with the ORCON module without having to trust its code base.

Once the ORCON module confirms it is interacting with an authenticated user[32], it decrypts the file and provides the user with read-only or read and append access, based on the policy file. Any append modifications are committed to the ORCON file when the user exits vi, at which point the file is encrypted and fingerprinted by the ORCON module. The key and hash protecting the file are stored in the module's secure storage area. In terms of the new security applications described in Chapter 3, this vi example can be seen as an incarnation of policy-protected objects, where each ORCON file comes with its own policy, enforced through its lifecycle by the vi modules, the only entities able to read or modify the ORCON files.

8.3 Complexity

Based on our implementation experience, the vast majority of the complexity for the Bastion architecture lies in the hypervisor software. The processor hardware requires extra registers, cache tags, crypto engines and extended TLB lookup logic. All of these hardware components are generic enough to accommodate a wide range of variation in the definition of the Trusted Programming Interface. This means that changes or extensions to the definition of Bastion can most likely be carried out by updating the Bastion hypervisor (using the procedure described in Chapter 6), without having to change the processor hardware, which is usually impossible to update once deployed. Modifying the hypervisor allows extending the definition of the Module State Table or adding hypercalls. As long as no new hardware registers are needed and the architectural mechanisms related to memory compartmentalization are not changed, there is no need to modify the processor hardware.

[32] N.B.: This implementation assumes secure I/O paths to the user are in place.

Tables 8.1 and 8.2 show, respectively, the hardware and software complexity of the Bastion TCB mechanisms. Table 8.1 shows our modifications to the microprocessor core increase FPGA resource usage by less than 10%. The new AES core causes the biggest increase in resource consumption. Table 8.2 shows that increases in the size of the hypervisor and firmware code base are between 6 and 8%. We note that the hypervisor is two orders of magnitude smaller in complexity than the guest operating system, which supports one of the core assumptions of this thesis: it is easier to trust a hypervisor than it is to trust a commodity operating system. Inspecting software to discover all security vulnerabilities is much more likely to succeed with systems formed of fewer than 50K lines of code, than with systems formed of millions of lines of code.

Table 8.1. 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 8.2. Software Complexity

	Original System	Our TCB Additions *absolute (change)*
OpenSolaris OS	9.06M	0 (+0%)
Ubuntu Linux OS	5.45M	0 (+0%)
Hypervisor	38.1K	2.9K (+7.6%)
Cache Firmware	12.1K	0.8K (+6.6%)
CPU Routines	0	4.5K (N/A)
`secure_launch`	0	1.5K (N/A)
`attest`	0	3.0K (N/A)

In Table 8.3, we present the complexity of the application-level libraries forming our Trusted Programming Interface. The Bastion hypercall library is used by trusted software modules in requesting security services from the Bastion TCB and the LibC Wrapper library is used to interface with the untrusted part of the software stack (see Figure 7.6 in Section 7.5.3 of Chapter 7). The security segment library is used to facilitate the creation of a security segment data structure respecting the Bastion architectural definition. Table

8.4 shows the complexity of our LibC wrapper library[33] in more detail, distinguishing between the trusted and untrusted part of the LibC call wrappers[34]. Note that we only implemented the LibC calls that were necessary to run the security-enhanced vi on Bastion. Functions highlighted in gray are utility functions that do not require operating system intervention; they are fully contained within each trusted software modules, as part of the trusted component of the LibC call wrapper library. All software complexity numbers presented in this chapter were generated using 'SLOCCount' by David A. Wheeler [Wheeler 2010]. We note that the complexity numbers in tables 8.1 to 8.4 could be significantly reduced if the Bastion hardware logic and hypervisor software was to be implemented and optimized by experienced engineers.

In Table 8.4, the trusted parts of the `malloc`, `printf` and `scanf` wrappers have the largest number of lines of code. This can easily be explained for `malloc`: as described in Section 3.1.1 of Chapter 3, each trusted software module must have its own copy, within its protected memory space, of heap allocation functions. This explains the high number of lines for the trusted part of the wrapper and the absence of an untrusted wrapper component. The `printf` and `scanf` are special cases: these functions allow for a variable number of function arguments, which cannot be determined before runtime. In order to copy the function arguments to module zero memory, the trusted part of the wrapper must parse the format string argument to determine the number and format of arguments. This parsing requires a significant portion of the regular `printf` and `scanf` code to be included in the trusted part of the wrapper. Most other LibC call wrappers do not have to deal with a variable argument list and therefore can be written with a small trusted component simply copying a statically determined number of arguments to module zero memory.

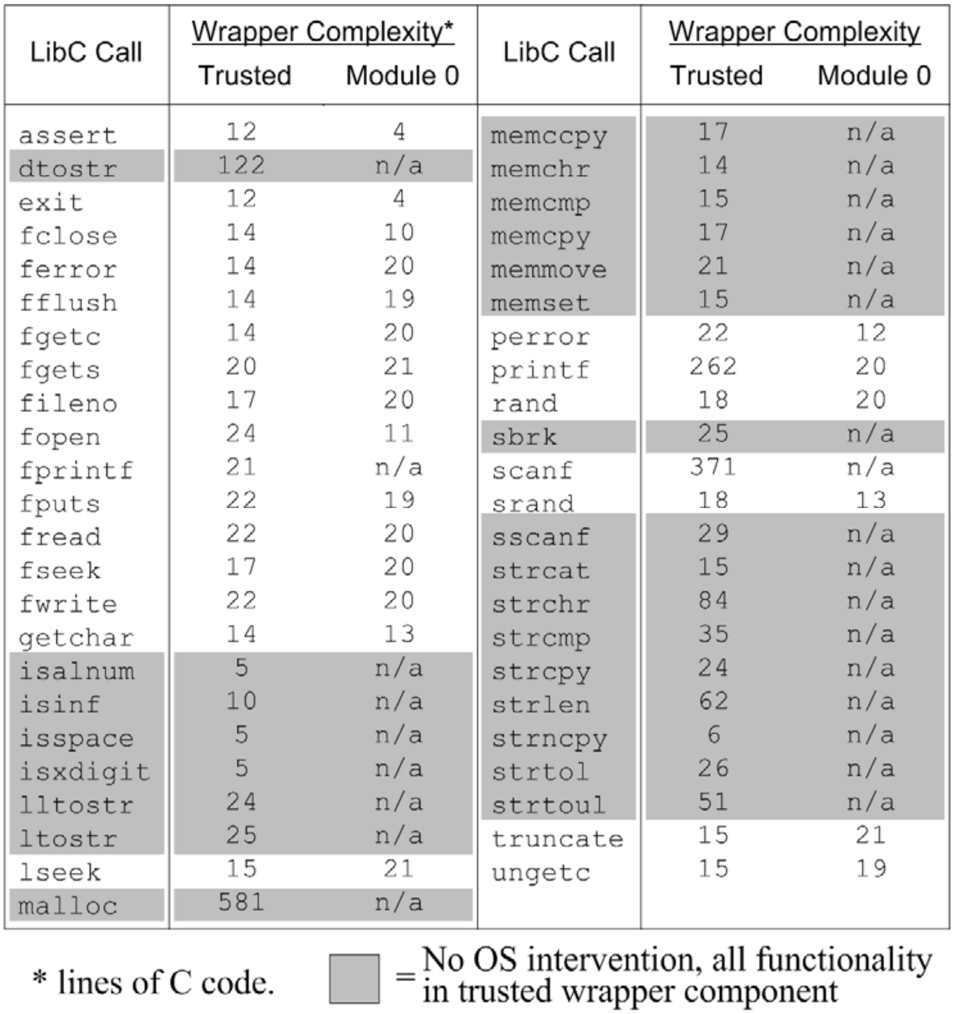
Table 8.3. Complexity of Application-level Bastion Libraries
(complexity in lines of C code, except for the hypercall library,
where it is given in lines of inline SPARC-64 assembly)

Application-level component	Complexity
Bastion Hypercall Library	133
Security Segment Library	202
LibC Call Wrappers Library	2,958

[33] Special thanks go to Yu-Yuan Chen for helping out in the coding of this library, and for lending us the code used in [Chen and Lee 2009]

[34] As a quick summary of the classes of LibC functions presented, we can say that functions starting with the letter “f” relate to file operations, the “str” prefix relates to string operations, the “is” prefix relates to checks on the value of a character and the “mem” prefix relates to memory operations such as move and copy. `malloc` and `sbrk` are heap management functions.

Table 8.4. Complexity of LibC Wrapper Library



LibC Call	Wrapper Complexity*		LibC Call	Wrapper Complexity	
	Trusted	Module 0		Trusted	Module 0
assert	12	4	▨ memccpy	17	n/a
▨ dtostr	122	n/a	▨ memchr	14	n/a
exit	12	4	▨ memcmp	15	n/a
fclose	14	10	▨ memcpy	17	n/a
ferror	14	20	▨ memmove	21	n/a
fflush	14	19	▨ memset	15	n/a
fgetc	14	20	perror	22	12
fgets	20	21	printf	262	20
fileno	17	20	rand	18	20
fopen	24	11	▨ sbrk	25	n/a
fprintf	21	n/a	scanf	371	n/a
fputs	22	19	srand	18	13
fread	22	20	▨ sscanf	29	n/a
fseek	17	20	▨ strcat	15	n/a
fwrite	22	20	▨ strchr	84	n/a
getchar	14	13	▨ strcmp	35	n/a
▨ isalnum	5	n/a	▨ strcpy	24	n/a
▨ isinf	10	n/a	▨ strlen	62	n/a
▨ isspace	5	n/a	▨ strncpy	6	n/a
▨ isxdigit	5	n/a	▨ strtol	26	n/a
▨ lltostr	24	n/a	▨ strtoul	51	n/a
▨ ltostr	25	n/a	truncate	15	21
lseek	15	21	ungetc	15	19
▨ malloc	581	n/a			

* lines of C code. ▨ = No OS intervention, all functionality in trusted wrapper component

8.4 Performance

In this section, we present the performance evaluation we carried out with our Bastion prototype. These results represent only a fraction of the full Bastion performance and sensitivity analysis which is to be carried out in future work. The current Bastion prototype requires significant efforts in optimization and debugging to improve its performance and reliability before it can be used to fully profile the performance impact of our mechanisms. The construction of architectural prototypes that can be used for precise comparison of Bastion with past architecture proposals is also left to future work.

One of the main challenges lies in synthesizing a Bastion microprocessor system with realistic clock domains for the processor core, the L2 cache logic and the memory encryption and hashing engines. The current prototype could only be synthesized with highly asymmetrical clock values (very slow core, very high speed cache and crypto logic), making cycle accurate performance results non representative. Our FPGA system implementation was thus used mainly to demonstrate that the Bastion CPU hardware and hypervisor software could work together to bypass an unmodified commodity operating system in providing protection to trusted applications software modules. We did so by running an application containing two simple modules interacting via a shared page. The two modules were successfully launched and protected as the securely launched Bastion hypervisor was running within its encrypted and hashed memory space. This experiment was repeated on both the Sun OpenSolaris operating system and the Linux Ubuntu operating system, without having to modify a single line of hypervisor code. The rest of our evaluation experiments, presented below, was carried out on the Legion architectural simulator (described in Chapter 7).

In future work, comparing Bastion with past architecture proposals should ideally be carried out by implementing all proposals on the same hardware and software baseline, on the FPGA. While this approach represents a significant amount of implementation work, it can lead to a precise characterization of the memory subsystem behavior, which the Legion simulator cannot achieve. An alternative is to develop a timing model for the different architectures and use the output of the functional simulator in conjunction with the timing model to determine the exact impact of the different architectural mechanisms—especially memory protection mechanisms—on the performance of protected software.

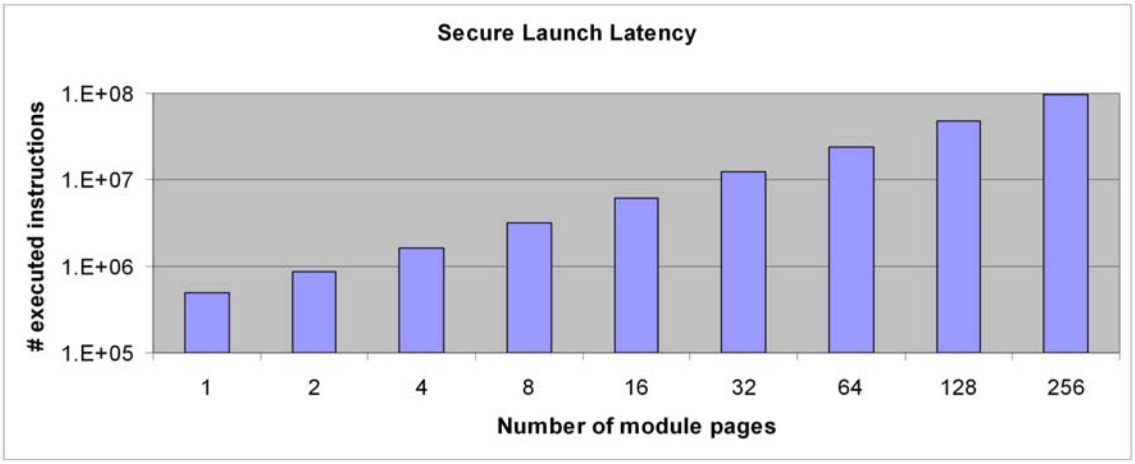
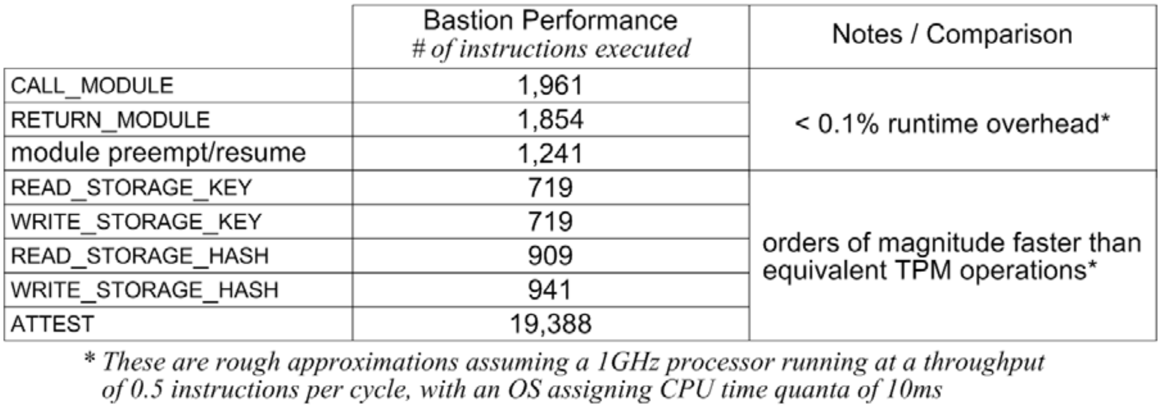


Figure 8.1. Latency of SECURE_LAUNCH hypercall for various module sizes

Figure 8.1 shows the latency associated with our SECURE_LAUNCH hypercall, for various module memory space sizes (note the logarithmic scales). It shows a linear relationship between the number of pages forming the module's memory space and the latency of the SECURE_LAUNCH hypercall. This was expected due to the large fraction of the hypercall workload dedicated to computing the module's identity, which includes a hash over its entire memory space. In other implementations of Bastion, the latency of the hypercall could be greatly reduced by modifying the on-chip integrity tree engine so that it computes the module identity as it adds the module pages to the integrity tree, without involving hypervisor software. In our current implementation, large modules with short life spans suffer from a large overhead due to the cost of the SECURE_LAUNCH hypercall. However, the SECURE_LAUNCH latencies shown in Figure 8.1 can be amortized to be less than a 1% overhead in the module's lifetime when the module executes for several billion instructions (which would amount to a few seconds of real CPU time on most modern processors).

Table 8.5. Bastion Hypercall Latencies (# of assembly instructions executed)



	Bastion Performance *# of instructions executed*	Notes / Comparison
CALL_MODULE	1,961	< 0.1% runtime overhead*
RETURN_MODULE	1,854	
module preempt/resume	1,241	
READ_STORAGE_KEY	719	orders of magnitude faster than equivalent TPM operations*
WRITE_STORAGE_KEY	719	
READ_STORAGE_HASH	909	
WRITE_STORAGE_HASH	941	
ATTEST	19,388	

* *These are rough approximations assuming a 1GHz processor running at a throughput of 0.5 instructions per cycle, with an OS assigning CPU time quanta of 10ms*

Table 8.5 shows the measured latencies for the module invocation hypercalls (CALL_MODULE and RETURN_MODULE), the secure module preemption and resumption operations, the secure storage hypercalls (READ/WRITE_STORAGE_KEY, READ/WRITE_STORAGE_HASH) and the ATTEST hypercall. To give a rough approximation of the overall overhead caused by our secure module transition mechanisms (invocation and preemption), we take a hypothetical scenario where 10ms CPU time quanta are allocated by the operating system. In the case of a long-running module that does not invoke other modules, the invocation overheads become negligible and the sole overhead is caused by module preemption monitoring from the hypervisor. Assuming a conservative throughput of 0.5 instructions per cycle, this leads to a runtime overhead of less than 0.1% (1241 instructions @ 1GHz = 1241 nanoseconds overhead inccured every 10 milliseconds. 1241ns/10ms = 0.012%)

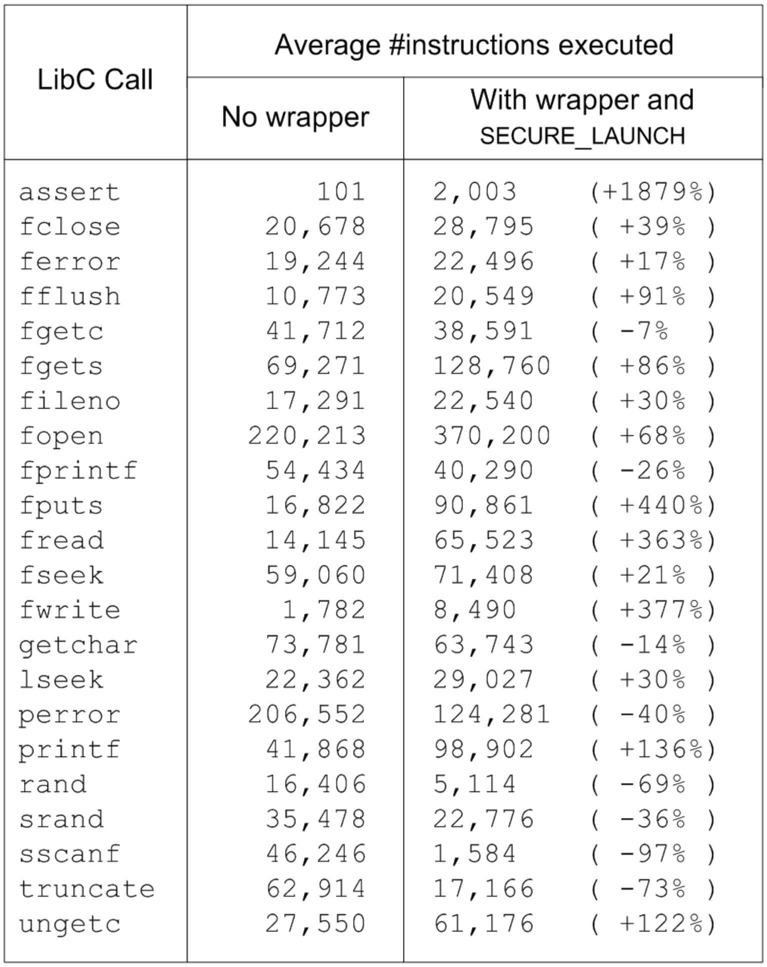
Table 8.5 also shows that integration of Bastion secure storage and attestation primitives to the hypervisor and CPU make these operations orders of magnitude faster than equivalent TPM operations, which have been reported to take several hundreds of milliseconds [McCune et al. 2008]. In general, TPM operations are significantly slower because they need to execute on a separate, low resource chip. Bastion operations take place on a high-performance general-purpose microprocessor, while TPM operations take place on a low-resource chip, with poor computational power and little memory. These limitations are not intrinsic to TPM's design; rather, they are due to the business model of TPM vendors, who want to implement TPM as a low-cost security anchor adding only a few cents to the overall price of a computing platform. In addition to having security operations take place on a low-performance chip, TPM-based security solutions must also incur the extra costs of communicating with the external TPM. In most existing computing platforms, the TPM chip is connected to the main processor via the slow LPC I/O bus. This implementation decision was presumably taken to ensure the TPM's I/O interface would be as simple and thus as cheap as possible to implement. Recent Intel platforms [Intel 2009b] offer an instantiation of the TPM logic within their chipset, which has a higher frequency than a traditional TPM chip, and is connected to the main processor via a much faster bus. A drawback of all TPM instantiations is that the TPM's design requires all sealing and unsealing operations to be carried out using a public-private key pair to encrypt a symmetric key which in turn protects a piece of secure storage. This requirement for asymmetric key cryptography makes TPM secure storage operations very slow compared to Bastion, which uses symmetric key cryptography.

Table 8.6 presents the overheads associated with our trusted LibC wrapper library, which allows trusted software modules to interface with the untrusted operating system (LibC functions that do not require OS intervention are not profiled in this table). The overhead of each wrapper varies depending on how many extra operations must be carried out in order to transfer call parameters between the trusted and untrusted domain. In the case of the `assert` function, the number of operations carried out by the LibC call is so small that the number of instructions required for our CALL_MODULE and RETURN_MODULE hypercalls lead to a very significant overhead of 1,879%. These overheads can be reduced by implementing CALL_MODULE as a direct jump to the callee module rather than using the two-step, register-then-intercept approach described in Section 7.4.3 of Chapter 7. The next three calls with the highest overheads—`fread`, `fwrite` and `fputs`—all involve the copying of a string and a transition between the trusted software module and module zero, leading to a significant performance hit.

In certain cases, the overheads are negative, meaning that invoking the LibC call via the trusted wrapper is faster than doing the LibC call directly. This is due to the fact that some of our wrappers contain LibC code to perform several operations within the trusted part of the wrapper before making a call to the untrusted OS or the untrusted LibC. The LibC code we integrated into our wrappers was taken from DietLibC [DietLibC 2010], a minimalist LibC library usually used in embedded systems. For the cases used to test the overheads of our wrapper library, some of the wrappers using DietLibC code actually performed better than the non-simplified LibC implementation used as a baseline. This explains the negative overheads in Table 8.6, where each instruction count is an average over 10 experiments.

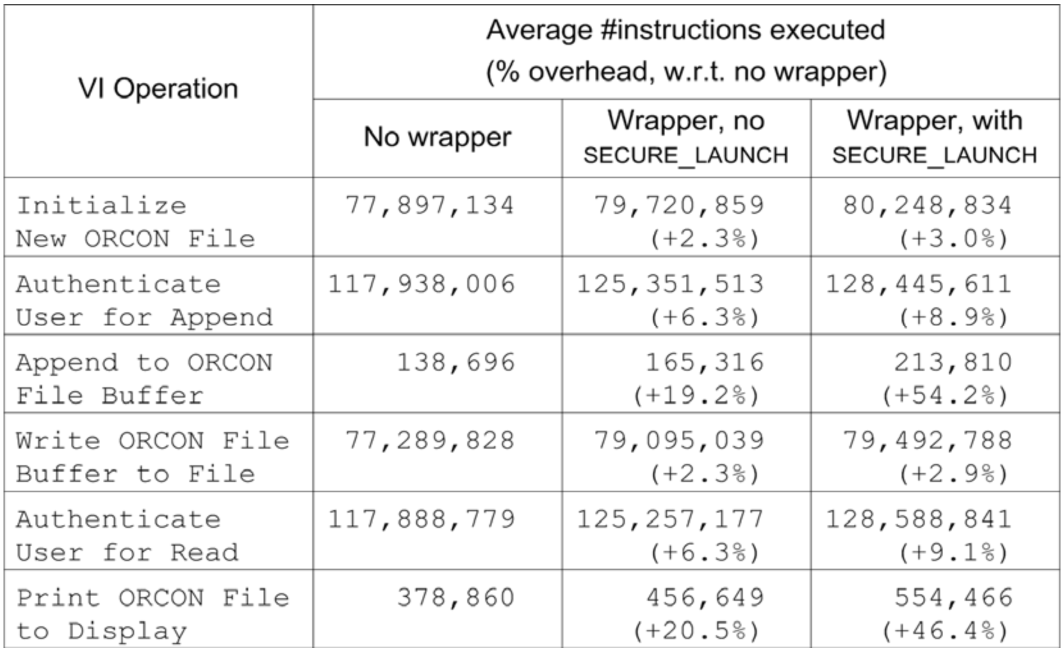
Finally, in Table 8.7, we present the performance impact of Bastion on a real-world security-critical application: the security-enhanced vi editor described in [Chen and Lee 2009]. The six different security operations performed by the editor were profiled for performance in three scenarios. “No wrapper” is the baseline scenario, where the security operations are carried out without any Bastion protection, using direct calls to the untrusted LibC library. The “Wrapper, no SECURE_LAUNCH” scenario presents the latencies for security operations in modules that are not provided with Bastion protection, but do make LibC calls through our library of LibC wrappers. This is a fictitious scenario, included to isolate the performance overhead of the wrappers themselves from the overhead of Bastion protection, included in our third scenario. The third, “Wrapper, with SECURE_LAUNCH”, scenario presents results for the security operations performed by securely launched modules running in Bastion-protected execution compartments, and using our library of LibC wrappers to interface with the untrusted part of the software stack.

Table 8.6. LibC Wrapper Overheads



LibC Call	Average #instructions executed	
	No wrapper	With wrapper and SECURE_LAUNCH
assert	101	2,003 (+1879%)
fclose	20,678	28,795 (+39%)
ferror	19,244	22,496 (+17%)
fflush	10,773	20,549 (+91%)
fgetc	41,712	38,591 (-7%)
fgets	69,271	128,760 (+86%)
fileno	17,291	22,540 (+30%)
fopen	220,213	370,200 (+68%)
fprintf	54,434	40,290 (-26%)
fputs	16,822	90,861 (+440%)
fread	14,145	65,523 (+363%)
fseek	59,060	71,408 (+21%)
fwrite	1,782	8,490 (+377%)
getchar	73,781	63,743 (-14%)
lseek	22,362	29,027 (+30%)
perror	206,552	124,281 (-40%)
printf	41,868	98,902 (+136%)
rand	16,406	5,114 (-69%)
srand	35,478	22,776 (-36%)
sscanf	46,246	1,584 (-97%)
truncate	62,914	17,166 (-73%)
ungetc	27,550	61,176 (+122%)

Table 8.7. Performance Impact of Bastion on security-enhanced vi



VI Operation	Average #instructions executed (% overhead, w.r.t. no wrapper)		
	No wrapper	Wrapper, no SECURE_LAUNCH	Wrapper, with SECURE_LAUNCH
Initialize New ORCON File	77,897,134	79,720,859 (+2.3%)	80,248,834 (+3.0%)
Authenticate User for Append	117,938,006	125,351,513 (+6.3%)	128,445,611 (+8.9%)
Append to ORCON File Buffer	138,696	165,316 (+19.2%)	213,810 (+54.2%)
Write ORCON File Buffer to File	77,289,828	79,095,039 (+2.3%)	79,492,788 (+2.9%)
Authenticate User for Read	117,888,779	125,257,177 (+6.3%)	128,588,841 (+9.1%)
Print ORCON File to Display	378,860	456,649 (+20.5%)	554,466 (+46.4%)

The results in the second column (Wrapper, with SECURE_LAUNCH) show that our LibC wrappers add an overhead of around 2% when there are few interactions with the untrusted part of the software stack (i.e., the the original vi code, LibC and the OS), as in the initialization of a new ORCON file and the writing of the ORCON file buffer to the ORCON file. More interaction with the OS is involved in the authentication operation, where several console operations are performed[35], leading to a higher impact of our wrappers on performance, and an overall overhead of around 6%.

The results in the third column (Wrapper, with SECURE_LAUNCH) show that the overall performance impact of Bastion varies depending on the nature and complexity of the operations it performs, with very simple operations (low instruction counts) being affected by a high overhead (~50%). Note that in appending to the ORCON file buffer, the example related in Table 8.7 increased the size of the initial ORCON file by 33%. For complex operations (requiring more than 50M instructions), the overhead is very low when most computations occur within the module itself. Indeed, the overhead is around 3% for initialization of the ORCON file (first row) and writing to the file (fourth row); in both cases, the main computation consists in computing cryptographic fingerprint over the file, an operation carried out entirely within the module. For complex operations that require many interactions with the second trusted software module and with the untrusted part of the software stack (i.e., the the original vi code, LibC and the OS), the overhead is around 9%. This is explained by the extra costs associated with switching between trusted modules and interfacing with the untrusted domain. As described in Section 8.2.4 of this chapter, authenticating the user requires a challenge-response protocol between the two trusted software modules in vi, where one module encrypts a challenge with a public key and the other decrypts it with the corresponding private key. This explains the high instruction counts involved in these operations.

8.5 Chapter Summary

In this chapter, we presented our security, functionality, complexity and performance evaluations of Bastion. Our security evaluation presented an informal, yet rigorous argument for why the Bastion security mechanisms achieve the security objective introduced in Chapter 2. We then presented a detailed scenario demonstrating the flexibility of Bastion functionality in supporting multiple security policies concurrently. We then described our implementation of security-critical tasks in a real-world text editor application and its protection using Bastion mechanisms. Based on this application and other micro-benchmarks, we obtained quantitative complexity and performance numbers for Bastion mechanisms.

[35] In gathering instruction counts, we removed the time spent waiting for a user console input to avoid the variability associated with user reaction times and typing speeds.

Chapter 9

Conclusion

Contemporary attackers exhibit an increasing degree of sophistication in trying to undermine the confidentiality and integrity of security-critical software and the sensitive data it handles. Corruption and snooping may be carried out by first compromising the OS on the target platform, or by launching a physical attack against the platform's hardware. Existing solutions for protecting security-critical software suffer shortcomings in either security, functionality or scalability—making them less likely to be enthusiastically deployed in mainstream computer systems.

In this thesis, we presented the Bastion architecture, a hardware-software Trusted Computing Base resistant to physical attacks and able to secure trusted software within an untrusted software stack, including a potentially compromised operating system. The Bastion hypervisor provides scalable security primitives enabling support for a variety of security policies in trusted OS or application software modules. This chapter presents a summary of the contributions of this thesis and proposes directions for future research.

9.1 The Bastion Architecture

The Bastion architecture we presented in Chapters 1 to 8 provides four distinct contributions to the field of secure computing.

The first contribution of this thesis is the design of a set of hardware mechanisms in the microprocessor to enable the secure launch and runtime protection of a virtualization layer, also called a hypervisor. A new `secure_launch` instruction traps to a trusted routine stored in an on-chip ROM and executed in a protected memory area within the microprocessor. This routine asserts full control over critical microprocessor settings and leverages two new hardware engines to set up an encrypted and integrity-protected memory space protecting the execution of the virtualization layer against physical attacks; the existing negative ring mechanism protects it against software attacks. Two dedicated non-volatile registers in the processor store the key and hash protecting a piece of persistent storage for the hypervisor. With virtualization support on an increasing number of mainstream microprocessors, design of direct hardware mechanisms protecting the so-called hyperprivileged layer of software against new threats is a significant contribution to the field of secure computing.

Our second contribution is the co-designed hardware-software Trusted Computing Base (TCB) protecting trusted OS and application modules within an untrusted, unmodified commodity software stack. A simple, but flexible Trusted Programming Interface allows modules to request TCB protection directly from the hypervisor, bypassing the guest operating system. The TCB protects the physical and virtual memory spaces of modules, and secures their execution by mediating invocation and preemption. Our TCB enables secure inter-module collaboration, a novel security service allowing for cross-authentication of caller and callee modules within a trust domain, and protected data exchange, despite untrusted surrounding software and hardware attackers.

The third contribution of this thesis is the design of two services, tailored attestation and secure persistent storage, assisting with secure I/O within an untrusted commodity software stack. Our TCB can tailor attestation reports to attest solely to the integrity of our hypervisor and the trusted software modules in a given trust domain, tied to a specific hardware CPU. This greatly simplifies the task of the attestation report recipients, who need to determine whether the reported software can be trusted. Secure storage is bound to, and accessible by only the associated trusted software modules so they can protect the integrity and confidentiality of long-lived secrets across platform reboots, despite a potentially compromised OS and (runtime or offline) physical attackers. Leveraging this module-specific secure storage and tailored attestation, Bastion-protected modules can establish secure I/O channels with local or remote trusted entities

Our fourth contribution consists in the implementation and evaluation of a Bastion prototype on the OpenSPARC platform. We modified SPARC processor hardware and the SPARC hypervisor to implement our Bastion TCB. Our implementation demonstrates the feasibility of bypassing unmodified commodity operating systems like Linux Ubuntu or Sun OpenSolaris in providing protection to trusted software modules against software and hardware attacks, at reasonable costs in complexity and performance. This prototype is a valuable platform to explore methodologies for the development and security evaluation of security-critical tasks protected by the Bastion TCB. It can also serve as a basis for implementing and evaluating extensions to the Bastion architecture developed as part of future research work.

In addition to the above, we also provide in Appendix A a survey of memory authentication techniques and engines that have been proposed in the literature. Emphasis is set on two proposals developed by the author of this thesis. One addresses the problem of memory authentication without a hypervisor, while the other proposal builds a novel tree structure out of the Added-Redundancy Explicit Authentication primitive, which provides both data integrity and confidentiality in a single invocation.

9.2 Directions for Future Research

Future research can build on the contributions of this thesis by extending the definition of the Bastion architecture to improve its security, functionality, performance and deployability. It could also leverage the prototype presented in this thesis to explore in more depth the impact of each architectural mechanism (and alternatives) on performance, complexity and compatibility with existing conventions. In the longer term, research work can look into developing new security services to improve the reliability and usability of the Bastion architecture.

The Trusted Computing Base we presented in this thesis is based on the threat model presented in Chapter 3. For the sake of bounding the problem under consideration, this threat model excludes the possibility of certain threats. Future work should look into enhancing the Bastion TCB to deal with an extended threat model. This should include improvement to the availability guarantees that can be provided by the Bastion TCB, as well as protection against information leakage on the memory address bus. As with the threat model, Bastion mechanisms ignore certain software conventions to ensure their design can be clearly presented and evaluated within the scope of this thesis. Research into extending Bastion mechanisms to support recursive virtualization, multiple page sizes, demand paging (and its lazy mapping to physical memory) and module code reentrancy could deliver great insight into the tradeoffs between security, compatibility and performance in modern software stacks.

Since it can run real applications on actual hardware, the Bastion prototype presented in this thesis can be used as a platform to explore methodologies for partitioning applications and operating systems into trusted and untrusted components. Partitioned software systems can then be run on the Bastion TCB to perform penetration testing and to determine whether their functionality was affected by the partitioning. Our prototype can be refined and further instrumented to provide a platform for more precise measurement of the performance impacts of the Bastion mechanisms. It can also be used as a basis for exploration of the different variations and alternatives for the security mechanisms in the Bastion TCB, especially with respect to physical memory protection engines.

This thesis focused on integrity and confidentiality of the data handled by trusted software. Longer term research should look into developing new secure computing services improving the reliability and usability, which could apply to Bastion as well as other security architectures. One possible avenue is to look at how memory integrity mechanisms can be extended to provide recovery from memory corruption rather than simply detecting corruption. Another important reliability issue is to provide better guarantees for the non-bypassability of trusted software modules playing the role of security monitors. It may also be possible to improve the usability of attestation mechanisms by developing a form of attestation that allows a remote party to determine whether a large piece of code (that is trusted but potentially contains exploitable software vulnerabilities) has been compromised in a way that affects the remote party's security objectives. This would improve on the current "binary" state of affairs, where a software component described in an attestation report is either fully trusted or completely untrusted.

Attackers today are motivated by the increasing amount of high value data processed by computer systems. Intrusion vectors abound, as the attack surface of the typical system is not only large, but well-exposed, thanks to ubiquitous internet connectivity. Defending against threats is thus a problem that requires a full-blown security architecture rather than an amalgam of ad hoc solutions. By protecting critical software systems, and the data they handle, from run-of-the-mill as well as sophisticated attacks, the mechanisms and concepts developed for the Bastion security architecture can make the digital world a safer place.

Appendix A

Alternative Memory Protection Schemes[36]

Memory authentication is a core security service provided by the Bastion architecture. Defined in a general sense, memory authentication is the ability to verify that the data read from memory by the processor (or by a specific application) at a given address is the data it last wrote at this address. In chapters 4 and 5, we presented the Bastion memory integrity tree, a processor memory authentication scheme based on the Merkle Tree. We chose this simplified scheme to ensure that Bastion architectural features could be clearly distinguished from the intricacies of the underlying memory authentication mechanism. In this appendix, we present a more systematic exploration of the memory authentication design space, by presenting the problem formally, and discussing the memory authentication techniques and engines that have been proposed in the literature. Our discussion includes two mechanisms developed by the author of this thesis, including a new type of memory integrity tree [Elbaz et al. 2007] and a solution for application memory authentication when the OS is untrusted on a platform without a trusted hypervisor [Champagne et al. 2008], e.g., in devices that do not support virtualization. The latter solution also enables memory integrity tree coverage of dynamically expanding memory spaces, a useful property when protecting only those pages that have been touched by a protected application.

The appendix is organized as follows. Section A.1 describes the active attacks threatening the integrity of memory contents. Then Section A.2 presents the existing techniques providing memory authentication, namely integrity trees; in particular, we show why all existing memory authentication solutions defending against the attacks in Section A.1 are based on tree structures. Section A.3 presents the architectural features proposed in the literature to efficiently integrate those integrity trees in general-purpose computing platforms. Section A.4 discusses the security of the integrity tree under operating system compromise and describes the architecture we proposed to efficiently deploy an integrity tree on a computing platform running an untrusted OS without a hypervisor. Section A.5 describes existing techniques to verify memory integrity that are not based on tree structures. While this appendix focuses on memory integrity for uniprocessor platforms, we present in Section A.6 the additional security issues to consider for data authentication in symmetric multiprocessor (shared memory) systems.

[36] This appendix is adapted from one of our prior publications [Elbaz et al. 2009]

A.1 Threat Model

This section describes the attacks challenging the integrity of data stored in the off-chip memory of computer systems. Section A.1.1 describes the model we consider for active attacks in the context of physical adversaries. Section A.1.2 widens the discussion by including the cases where these attacks are carried out through a malicious operating system; we conclude the section by defining two threat models upon which existing solutions for memory integrity are built.

A.1.1 Hardware Attacks

The common threat model considered for memory authentication assumes the protected system is exposed to a hostile environment in which physical attacks are feasible. The main assumption of this threat model is that the processor chip is resistant to all physical attacks, including invasive ones, and is thus trusted. Side-channel attacks are not considered.

The common objective of memory authentication techniques is to thwart active attackers tampering with memory contents. In an active attack, the adversary corrupts the data residing in memory or transiting over the bus; this corruption may be seen as data injection since a new value is created. Figure A.1 depicts the example of a device under attack, where an adversary conceptually connects its own (malicious) memory to the targeted platform via the off-chip bus. We distinguish between three classes of active attacks, defined with respect to how the adversary chooses the inserted data. Figure A.2 depicts the three active attacks; below, we provide a detailed description of each one by relying on the attack framework in Figure A.1:

1) *Spoofing attacks*: the adversary exchanges an existing memory block with an arbitrary fake one (Figure A.2-a, the block defined by the adversary is stored in the malicious memory, the adversary activates the switch command when he wants to force the processor chip to use the spoofed memory block).

2) *Splicing or relocation attacks*: the attacker replaces a memory block at address A with a block at address B, where A≠B. Such an attack may be viewed as a spatial permutation of memory blocks (Figure A.2-b: the adversary stores at address 5 in the malicious memory the content of the block at address 1 from the genuine memory. When the processor requests the data at address 5, the adversary activates the switch command so the processor reads the malicious memory. As a result, the processor reads the data at address 1).

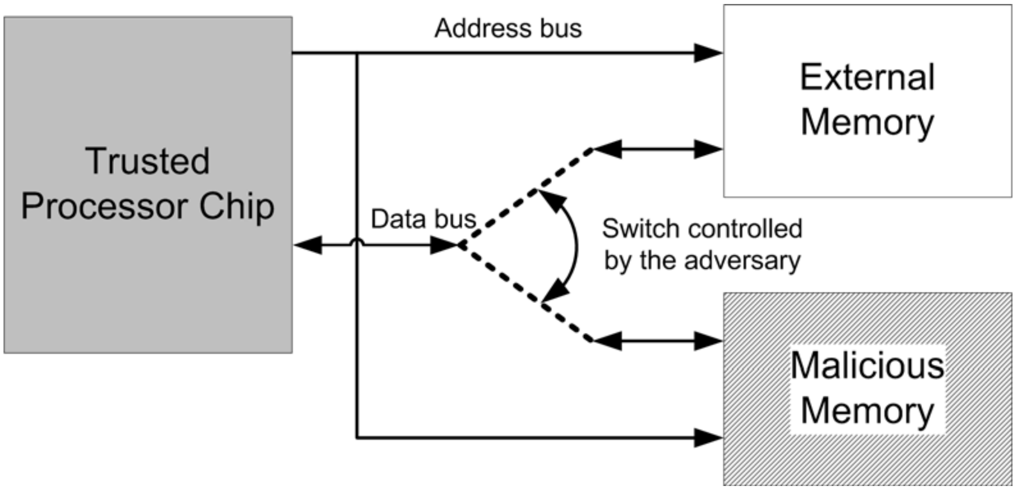


Figure A.1: An Example of Framework of Attack Targeting the External Memory of a Computing Platform.

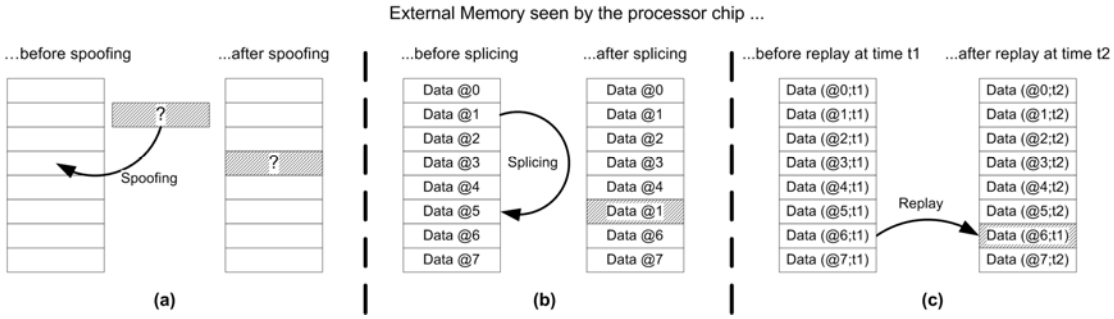


Figure A.2: Three Kinds of Active Attacks: (a) Spoofing, (b) Splicing and (c) Replay

3) *Replay attacks*: a memory block located at a given address is recorded and inserted at the same address at a later point in time; by doing so, the current block's value is replaced by an older one. Such an attack may be viewed as a temporal permutation of a memory block, for a specific memory location (Figure A.2-c: at time t1, the adversary stores at address 6 in the malicious memory the content of the block at address 6 from the genuine memory. At time t2, the memory location at address 6 has been updated in the genuine memory but the adversary does not perform this update in the malicious memory. The adversary activates the malicious memory when the processor requests the data at address 6, thus forcing it to read the old value stored at address 6).

A.1.2 Software Attacks

In a software attack, a compromised (or outright malicious) operating system or application tries to corrupt the memory space of a sensitive application. To model these attacks, we subsume all possible attack vectors into a single threat: a malicious, all-powerful operating system. Such an OS can directly read and write any memory location belonging to a sensitive application and can thus carry out any of the splicing, spoofing and replay attacks presented in the previous section.

The Bastion architecture presented in this thesis uses a trusted hypervisor to manage the integrity tree, which protects specific application or OS components against both hardware and software attacks. Past proposals on memory authentication use different threat models, where virtualization is not considered. They either exclude software attacks [Suh et al. 2003 (second proposal), Elbaz et al. 2007, Gassend et al. 2003, Suh 2005, Yan et al. 2006, Rogers et al. 2007, Elbaz et al. 2006] (referred in the following as *threat model 1*) or include software attacks [Suh et al. 2003 (first proposal), Champagne et al. 2008] (referred in the following as *threat model 2*). In threat model 2, the hardware architecture must protect sensitive applications against attacks from software, since no trusted hypervisor is present; in threat model 1, the hardware does not provide such protection. When software attacks are not considered (threat model 1), the operating system (OS) or at least the OS kernel must thus be trusted to isolate sensitive applications from malicious software. In the other case, the OS can contain untrusted code since the hardware protects sensitive applications against malicious software.

Conceptually, integrity trees are built and maintained in the same way regardless of the threat model considered. Section A.2 describes existing integrity trees without specifying the threat model. Section A.3 presents the strategies allowing efficient integration to computing platforms when threat model 1 is considered. Section A.4 shows that threat model 2 requires trees built over the virtual (rather than physical) address space or, as recently proposed in [Champagne et al. 2008], over a compact version of it. Chapters 4 and 5 of this thesis presented a strategy for implementing an integrity tree when the Bastion threat model is considered, i.e., threat model 2 with a trusted hypervisor.

A.2 Integrity Trees: Cryptographic Schemes for Memory Authentication

We consider there are three distinct strategies to thwart the active attacks described in our threat model. Each strategy is based on different authentication primitives, namely cryptographic hash function, Message Authentication Code (MAC) function and block-level Added Redundancy Explicit Authentication (AREA). In this section, we first describe how those primitives allow for memory authentication and how they must be integrated into tree structures in order to avoid excessive overheads in on-chip memory.

A.2.1 Authentication Primitives for Memory Authentication

Hash Functions. The first strategy (Figure A.3-a) allowing to perform memory authentication consists in storing on-chip a hash value for each memory block stored off-chip (*write operations*). The integrity checking is done on *read operations* by re-computing a hash over the loaded block and by then comparing the resulting hash with the on-chip hash fingerprinting the off-chip memory location. The on-chip hash is stored on the tamper-resistant area, i.e., the processor chip and is thus inaccessible to adversaries. Therefore, spoofing, splicing and replay are detected if a mismatch occurs in the hash comparison. However, this solution has an unaffordable on-chip memory cost:

by considering the common strategy [Suh et al. 2003, Lee et al. 2005, Gassend et al. 2003, Elbaz et al. 2006] of computing a fingerprint per cache line and assuming 128-bit hashes and 512-bit cache lines, the overhead is 25% of the memory space to protect.

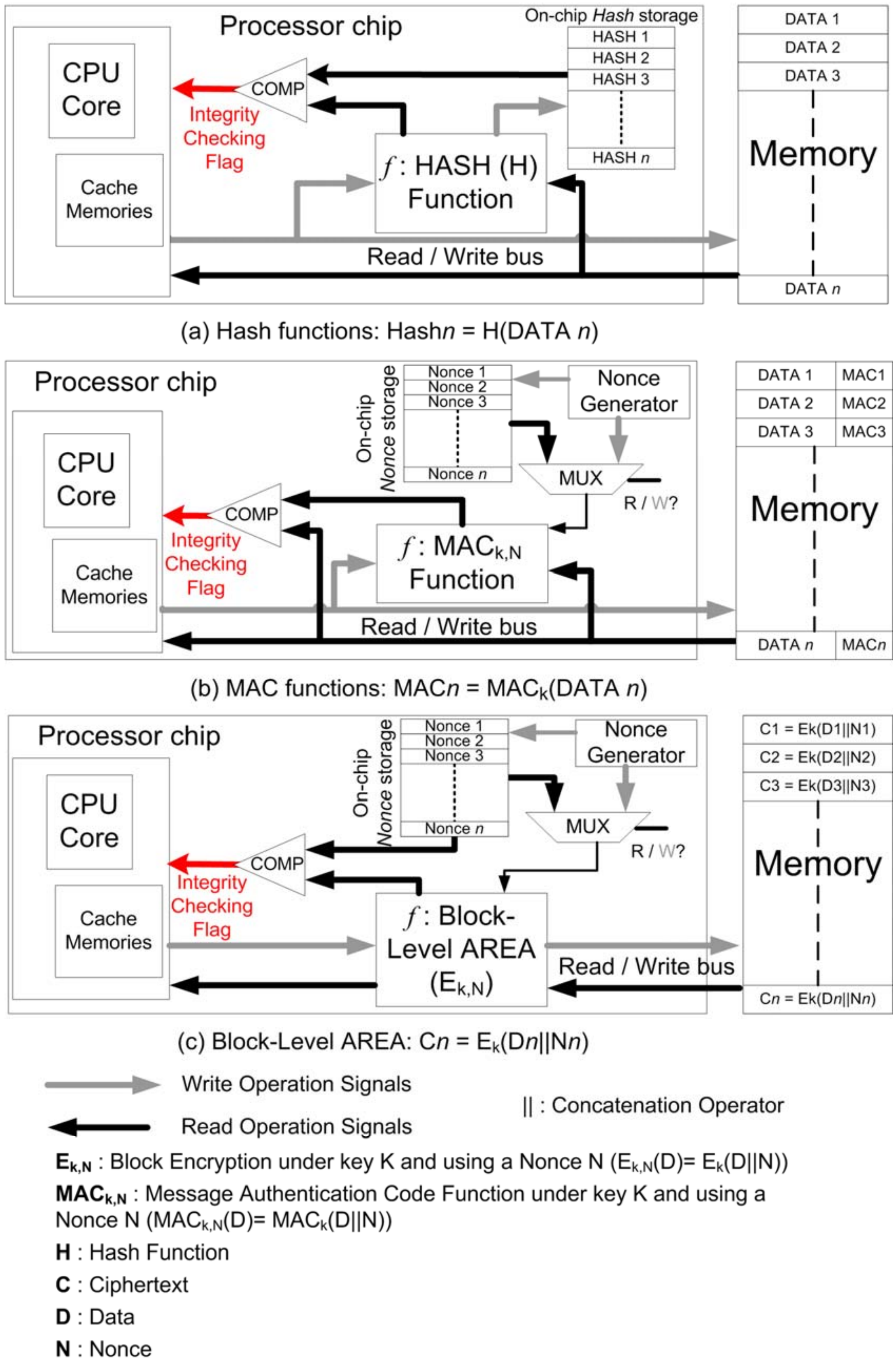
MAC Functions: In the second approach (Figure A.3-b), the authentication engine embedded on-chip computes a MAC for every data block it *writes* in the physical memory. The key used in the MAC computation is securely stored on the trusted processor chip such that only the on-chip authentication engine itself is able to compute valid MACs. As a result, the MACs can be stored in untrusted memory because the attacker is unable to compute a valid MAC over a corrupted data block. In addition to the data contained by the block, the pre-image of the MAC function contains a nonce. This allows protection against splicing and replay attacks. The nonce precludes an attacker from passing a data block at address A, along with the associated MAC, as a valid (data block, MAC) pair for address B, where A ≠ B. It also prevents the replay of a (data block, MAC) pair by distinguishing two pairs related to the same address, but written in memory at different points in time. On *read operations*, the processor loads the data to read and its corresponding MAC from physical memory. It checks the integrity of the loaded block by first re-computing a MAC over this block and a copy of the nonce used upon writing, and then it compares the result with the fetched MAC. To ensure the resistance to replay and splicing, the nonce used for MAC re-computation must be genuine. A naïve solution to meet this requirement is to store the nonces on the trusted and tamper-evident area, the processor chip. The related on-chip memory overhead is 12.5% if we consider computing a MAC per 512-bit cache line and that we use 64-bit nonces.

Block-Level AREA: The last strategy [Elbaz et al. 2006] (Figure A.3-c) leverages the diffusion property of block encryption to add the integrity-checking capability to this type of encryption algorithm. To do so, the AREA (Added Redundancy Explicit Authentication [Fruhwirth 2005]) technique is applied at the block level:

1) Redundant data (an *n*-bit nonce N) is concatenated to the data D to authenticate in order to form a plaintext block P (where P=D||N); ECB (Electronic CodeBook) encryption is performed over P to generate ciphertext C.

2) Integrity verification is done by the receiver who decrypts the ciphertext block C' to generate plaintext block P', and checks the *n*-bit redundancy in P', i.e., assuming P'=(D'||N'), verifies whether N=N'.

Thus, upon a *memory write*, the on-chip authentication engine appends an *n*-bit nonce to the data to be written to memory, encrypts the resulting plaintext block and then writes the ciphertext to memory. The encryption is performed using a key securely stored on the processor chip. On *read operations*, the authentication engine decrypts the block it fetches from memory and checks its integrity by verifying that the last *n* bits of the resulting plaintext block are equal to the nonce that was inserted upon encryption (on the write of the corresponding data). [Elbaz et al. 2006] propose a System-on-Chip (SoC) implementation of this technique for embedded systems. They show that this engine is efficient to protect the Read-Only (RO) data of an application (e.g., its code) because RO



(a) Hash functions: Hash$n$ = H(DATA n)

(b) MAC functions: MACn = MAC$_k$(DATA n)

(c) Block-Level AREA: Cn = E$_k$(Dn||Nn)

E$_{k,N}$: Block Encryption under key K and using a Nonce N (E$_{k,N}$(D)= E$_k$(D||N))

MAC$_{k,N}$: Message Authentication Code Function under key K and using a Nonce N (MAC$_{k,N}$(D)= MAC$_k$(D||N))

H : Hash Function

C : Ciphertext

D : Data

N : Nonce

Figure A.3: Authentication Primitives for Memory Integrity Checking

data are not sensitive to replay attacks; therefore the address of each memory block can be efficiently used as a nonce[37]. However, for Read/Write (RW) data (e.g., stack data), the address is not sufficient to distinguish two data writes at the same address carried out at two different points in time: the nonce must change on each write. To recover such a changing nonce on a read operation while ensuring its integrity, [Elbaz et al. 2006] propose storing the nonce on-chip. They evaluate the corresponding overhead between 25% and 50% depending on the block encryption algorithm implemented.

A.2.2 Integrity Trees

The previous section presented three authentication primitives preventing the active attacks described in our threat model. Those primitives require the storage of reference values—i.e., hashes or nonces—on-chip to thwart replay attacks. They do provide memory authentication but only at a high cost in terms of on-chip memory. If we consider a realistic case of 1GB of RAM memory, the hash, MAC (with nonce) and the block-level AREA solutions require respectively at least 256MB, 128MB and 256 MB of on-chip memory. Those on-chip memory requirements clearly are unaffordable, even for high-end processors. It is thus necessary to "securely" store these reference values off-chip. By securely, we mean that we must be able to ensure their integrity to preclude attacks on the reference values themselves.

Several research efforts suggest applying the authentication primitives recursively on the references. By doing so, a tree structure is formed and only the root of the tree—the reference value obtained in the last iteration of the recursion—needs to be stored on the processor chip, the trusted area. There are three existing tree techniques:

1) Merkle Tree [Merkle 1980] uses hash functions,

2) PAT (Parallelizable Authentication Tree) [Hall and Jutla 2005] uses MAC functions with nonces,

3) TEC-Tree (Tamper-Evident Counter Tree) [Elbaz et al. 2007] uses the block-level AREA primitive.

In this section, we first present a generic model for the integrity tree, then we describe the specific characteristics of each existing integrity tree; we finally compare their intrinsic properties.

General Model of Integrity Tree. The common philosophy behind integrity trees is to split the memory space to protect into M equal size blocks which are the leaf nodes of the balanced A-ary integrity tree (Figure A.4). The remaining tree levels are created by recursively applying the authentication primitive f over A-sized groups of memory blocks, until the procedure yields a single node called the root of the tree. The arity of the constructed tree is thus defined by the number of children A a tree node has. The root

[37] Note that the choice of the data address as nonce also prevent spoofing and splicing attacks of RO data when MAC functions are used as authentication primitives.

reflects the current state of the entire memory space; making the root tamper-resistant thus ensures tampering with the memory space can be detected. How the root is made tamper-resistant depends on the nature of *f* and is detailed next. Note that the number of checks required to verify the integrity of a leaf node depends on the number of iterations of *f* and thus on the number of blocks M in the memory space and the tree's arity. The number of check operations corresponds to the number of tree levels N_L defined by: $N_L = \log_A(M)$.

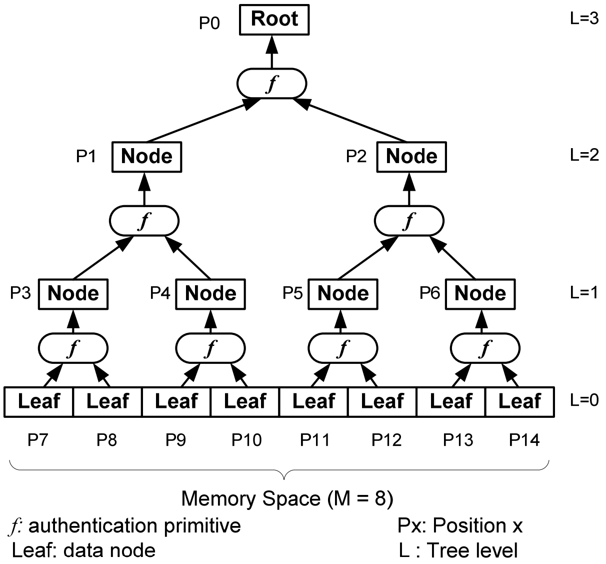


Figure A.4: General Model of 2-ary Integrity Tree

Tree Authentication Procedure. For each memory block B (i.e., leaf node), there exists a branch[38]—starting at B and ending at the root—composed of the tree nodes obtained by recursive applications of *f* on B. For instance in Figure A.4 for the leaf node at position P8, the branch is composed of the nodes at positions P3, P1 and the root P0. Thus, when B is fetched from untrusted memory, its integrity is verified by re-computing the tree root using the fetched B and the nodes—obtained from external memory—along the branch from B to the root (i.e., the branch nodes and their siblings; so for the leaf node at position P8, the nodes that need to be fetched are at position P7, P3, P4, P1 and P2). We confirm B has not been tampered with during the last step of the authentication process when the re-computed root is identical to the root (which has been made tamper-resistant). The authentication procedure is parallelizable if, when all nodes in a branch are present on-chip[39], the hardware can apply the authentication primitive in parallel to each node in a first step and make comparisons on the resulting outputs in a second step to determine whether the authentication check passes.

[38] This branch is also called in the following the *authentication branch*.

[39] This may not always be possible, especially when latencies for external memory accesses are high and on-chip buffering resources are low.

Tree Update Procedure. When a legitimate modification is carried out over a memory block B, the corresponding branch—including the tree root—is updated to reflect the new value of B. This is done by first authenticating the branch B belongs to by applying the previous tree authentication procedure, then by computing on-chip the new values for the branch nodes, and finally by storing the updated branch off-chip, except for the on-chip component of the root. The tree update procedure is parallelizable if, when all nodes in a branch are present on-chip[40], the hardware can apply the authentication primitive in parallel to each node (including the updated leaf node) in a single step.

Merkle Tree is historically the first integrity tree. It has been originally introduced by Merkle [Merkle 1980] for efficient computations in public key cryptosystems and adapted for integrity checking of memory content by Blum et al. [Blum et al. 1991]. In a Merkle Tree (Figure A.5-a), *f* is a cryptographic hash function H(); the nodes of the tree are thus simple hash values. The generic verification and update procedures described above are applied in a straightforward manner. The root of this tree reflects the current state of the memory space since the collision resistance property of the cryptographic hash function ensures that in practice, the root hashes for any two memory spaces differing by at least one bit will not be the same. With Merkle Tree, the root is made tamper-resistant by storing it entirely on the trusted processor chip. The Merkle Tree authentication procedure is fully parallelizable because all the inputs required can be made available before the start of the procedure; however, the update procedure is sequential because the computation of a new hash node in a branch must be completed before the update to the next branch node can start. By assuming that all tree nodes have the same size, the memory overhead MO_{MT} of a Merkle Tree, as stated in [Gassend et al. 2003], is of:

$$MO_{MT} = \frac{1}{(A-1)}.$$

The Parallelizable Authentication Tree (PAT) [Hall and Jutla 2005] overcomes the issue of non-parallelizability of the tree update procedure by using a MAC function (with nonces N) $M_{K,N}()$ as authentication primitive *f* where K and N are the key and nonce, respectively (Figure A.5-b). The memory space is first divided into M memory blocks (in Figure A.5-b, M=4). We begin by applying the MAC function to *A* memory blocks (in Figure A.5-b, we have *A*=2) using an on-chip key K and a freshly generated nonce N, i.e., $MAC_{K,N}(d_1||\ldots||d_A)$ where d_i is a memory block. Next, the MAC is recursively applied to A-sized groups formed with the nonces generated during the last iteration of the recursion. For every step of the recursion but the last, both the nonce and the MAC values are sent to external memory. The last iteration generates a MAC and a nonce that form the root of the tree. The root MAC is sent to external memory but the nonce N is stored on-chip; this way the tree root is made tamper-resistant since an adversary cannot generate a new MAC without the secret key K stored on-chip or replay an old MAC since it will not have been generated with the current root nonce. Verifying a memory block D

[40] This may not always be possible, especially when latencies for external memory accesses are high and on-chip buffering resources are low.

in a PAT requires recomputing D's branch on-chip and verifying that the top-level MAC can indeed be obtained using the on-chip nonce. Whenever a block D is legitimately modified, the CPU re-computes D's branch using fresh nonces. The tree authentication procedure is parallelizable since all inputs (data and nonces) are available for all branch node verifications. The tree update procedure is also parallelizable because each branch tree node is computed from independently generated inputs: the nonces. In [Hall and Jutla 2005] the authors highlight that the birthday paradox implies the nonce does not need to be longer than *h*/2, with *h* being the MAC-size. Thus, the memory overhead MO_{PAT} of PAT is of (the extra 50% overhead on top of the regular Merkle tree overhead is to store a PAT nonce with each tree node, whose width is half the node size):

$$MO_{PAT} = \frac{1}{A-1} + \frac{1}{2}\frac{1}{A-1} = \frac{3}{2(A-1)}$$

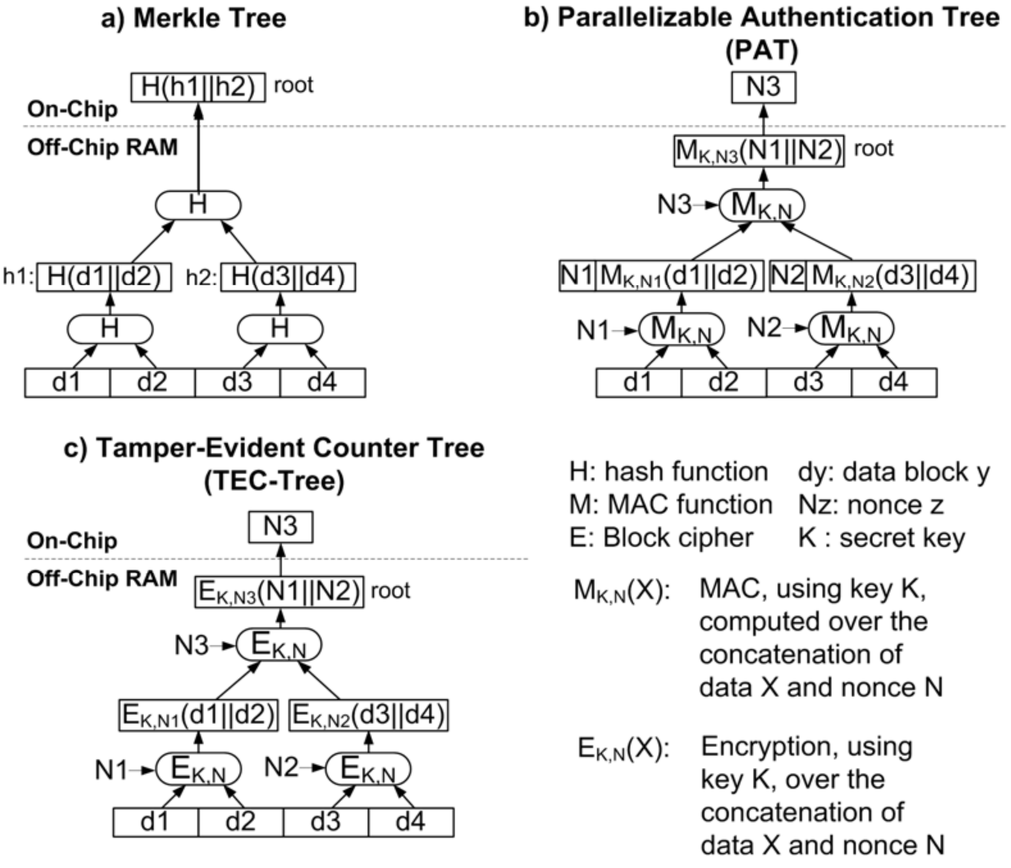


Figure A.5: Existing Integrity Trees

The Tamper-Evident Counter Tree (TEC-Tree). In TEC-Tree, the memory authentication scheme we introduced in [Elbaz et al. 2007], the authentication primitive *f* is the Block-level AREA. Thus, such authentication primitive tags its input (dy in Figure A.5-c) with a nonce N before ciphering it with a block encryption algorithm in ECB mode and a secret key K kept on-chip. The block-level AREA is first applied to the memory blocks to be stored off-chip, and then recursively over A-sized groups of nonces used in the last iteration of the recursion. The resulting ciphered blocks are stored in external memory and the nonce used in the ciphering of the last block created—i.e., the root of the TEC-Tree—is kept on-chip making the root tamper-resistant. Indeed, an adversary without the key cannot create a tree node and without access to the on-chip root nonce he cannot replay the tree root. During verification of a data block D, D's branch is brought on-chip and decrypted. The integrity of D is validated if:

1) each decrypted node bears a tag equal to the nonce found in the payload of the node in the tree level immediately above;

2) the nonce obtained by decrypting the highest level node matches the on-chip nonce.

The tree update procedure consists in:

1) loading D's branch

2) decrypting nodes

3) updating nonces

4) re-encrypting nodes.

TEC-Tree authentication and update procedures are both parallelizable because *f* operates on independently generated inputs: the nonces. The distinctive characteristic of TEC-Tree is that it allows for data confidentiality. Indeed, its authentication primitive being based on a block encryption function, the application of this primitive on the leaf nodes (data) encrypts them. The memory overhead[41] MO_{TEC} of TEC-Tree, derived in [Elbaz et al. 2007], is:

$$MO_{TEC} = \frac{2}{(A-1)}$$

Comparison. Table A.1 sums up the properties of the existing integrity trees. PAT and TEC-Tree are both parallelizable for the tree authentication and update procedure while preventing all the attacks described in the threat model. Parallelizability of the tree update process is an important feature when the write buffer is small (e.g., in embedded systems) to prevent bus contention due to write operation requests that may pile up. TEC-Tree additionally provides data confidentiality. However, TEC-Tree and PAT also have a higher off-chip memory overhead when compared to Merkle Tree, in particular because they require storage for additional metadata, the nonces.

[41] [Elbaz et al. 2007] gives a different formula for their memory overhead because they consider ways to optimize it (e.g. the use of the address in the construction of the nonce). For the sake of clarity, we give a simplified formula of the TEC-Tree memory overhead by considering that the nonce consists only of a counter value.

Table A.1: Summary of Existing Integrity Trees Properties

	Merkle Tree	**PAT** (Parallelizable Authentication Tree)	**TEC-Tree** (Tamper-Evident Counter Tree)
Splicing, Spoofing, Replay resistance	Yes	Yes	Yes
Parallelizability	Tree Authentication only	Tree Authentication **and** Update	Tree Authentication **and** Update
Data Confidentiality	No	No	Yes
Offchip Memory Overhead	1/(A-1)	3/2(A-1)	2/(A-1)

A.3 Integration of Integrity Trees in Computing Platforms

In this section we survey the implementation strategies proposed in the literature to efficiently integrate integrity trees in computing platforms when threat model 1 is considered (i.e., active physical attacks and trusted OS kernel). We first describe the tree traversal technique allowing for node addressing in memory. Then, a scheme leveraging caching to improve tree performance is detailed. Finally, the Bonsai Merkle Tree concept is described.

A.3.1 Tree Traversal Technique

One of the issues arising when integrating an integrity tree into a computing platform is making it possible for the processor to retrieve tree nodes in memory. [Gassend et al. 2003] proposes a method for doing so with Merkle trees, while [Elbaz et al. 2007] adapts it for TEC-Tree. The principle of this method is to first associate a numerical position (P_x in Figure A.4) to each tree node, starting at 0 for the root and incrementally up to the leaves. The position of a parent node P^l (at level l in the tree) can be easily found by subtracting one from its child at position P^{l-1} (on level l-1), by dividing the result by the tree arity A and by rounding down:

$$P^l = \left\lfloor \frac{P^{l-1} - 1}{A} \right\rfloor \qquad \text{(eq. A.1)}$$

Now that we know how to find a parent node position from a child position, the issue is to retrieve a position number from the address of a child or parent node. To solve this issue, [Gassend et al. 2003] proposes a simplified layout of the memory region to authenticate: the tree nodes are stored starting from the top of the tree (P1 in Figure A.4) down to the leaves, with respect to the order given by positions. By having all tree nodes be the same size, the translation from position to address or from address to position can be easily done by respectively multiplying or dividing the quantity to be translated by the node size.

This method imposes that the arity be a power of 2 to efficiently implement the division of eq. A.1 in hardware. Moreover, all data to authenticate must be contained in a contiguous memory region to allow for the children to parent position and address retrieval scheme to work.

A.3.2 Cached Trees

The direct implementation of integrity trees can generate a high overhead in terms of execution time due to the $\log_A(M)$ checks required on each load from the external memory. In [Gassend et al. 2003], they show that the performance slowdown can reach a factor of 10 with a Merkle Tree. To decrease this overhead, they propose to cache tree nodes. When a hash is requested in the tree authentication procedure, it is brought on-chip with its siblings in the tree that belong to the same cache block. This way, those siblings usually required for the next check in the authentication procedure are already loaded. However, the main improvement comes from the fact that once checked and stored in the on-chip cache, a tree node is trusted and can be considered as a local tree root. As a result, the tree authentication and update procedures are terminated as soon as a cached hash (or the root) is encountered. With this cached tree solution, Gassend et al. decreased the performance overhead of a Merkle Tree to less than 25%. By changing the hash function—from SHA-1[NIST 2002] to the GCM[NIST 2007]—[Yan et al. 2006] even claims to keep the performance overhead under 5%.

A.3.3 The Bonsai Merkle Tree

Memory authentication engines based on integrity trees should be designed for efficient integration into computing platforms. A recent engine proposed toward this objective has been presented in [Rogers et al. 2007] and is based on a concept called Bonsai Merkle Tree.

The idea behind the Bonsai Merkle Tree (BMT) is to reduce the amount of data to authenticate with a Merkle Tree in order to decrease its height, i.e., reduce the number of tree levels to obtain a smaller tree that can be quickly traversed (Figure A.6). To do so, [Rogers et al. 2007] proposes to compute a MAC M over every memory block C (i.e., every cache block) with a nonce, i.e., a counter *ctr* concatenated with the data address, as extra input of the MAC function MAC_K: $M = \text{MAC}_K\ (C, addr, ctr)$[42]. The counter *ctr* consists of a local counter *Lctr* concatenated with a global counter *Gctr*. Each memory block is associated with a local counter while each memory page is associated with a global counter. Each time a given memory block is updated off-chip, the corresponding *Lctr* is incremented. When *Lctr* rolls over, *Gctr* is incremented. In [Rogers et al. 2007], the authors proposed the use of a 7-bit long *Lctr* and of a 64-bit long *Gctr*; this way *ctr* never rolls over in practice. *ctr* counter values are made tamper-evident while stored off-chip using a Merkle tree, thus making the memory space protected by the MAC-with-nonce scheme also tamper-evident. To summarize, the authentication procedure for a memory bloc *C* consists in fetching the MAC *M* and the counter *ctr* associated with *C* (including a regular Merkle Tree authentication procedure to verify the integrity of *ctr* using the Bonsai Merkle Tree) and then recomputed a MAC over *C*, *C*'s address and *ctr* to check whether it matches *M*. The tree update procedure consists in incrementing the counter to *ctr'*, recomputing *M'* over the updated memory block *C'* and *ctr'*, and writing back to memory *M'*, *C'*, and *ctr'* (including a regular Merkle Tree update procedure for the Bonsai Merkle Tree).

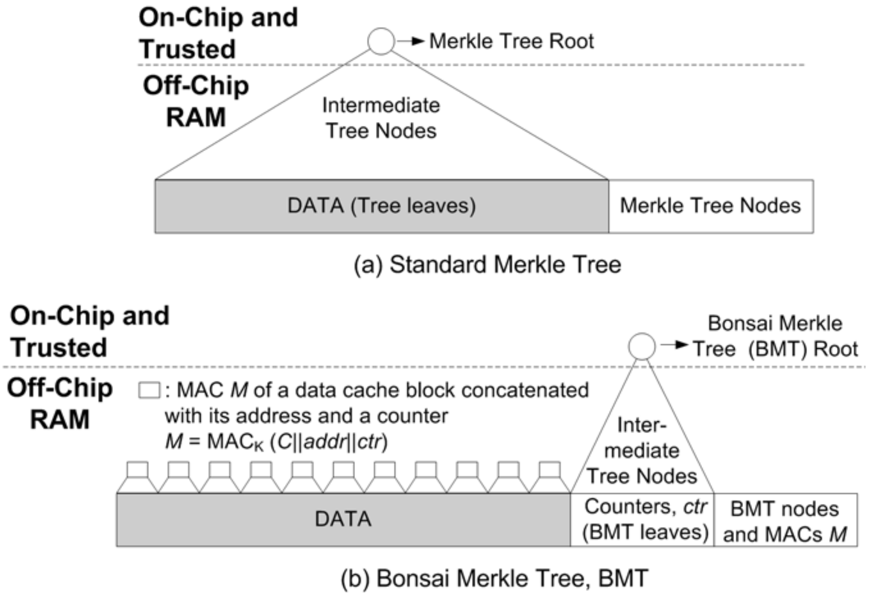
On average, an 8-bit counter is required for 4KB memory pages and 64B cache blocks. The amount of memory to authenticate with the Bonsai Merkle tree is thus greatly reduced when compared to a regular Merkle tree applied directly to the memory blocks. However, the shortcoming of this scheme is that a full page needs to be cryptographically processed every time a local counter rolls over. In other words, in this case the MACs of all memory blocks belonging to the page having *Gctr* updated must be recomputed as well as the tree branches corresponding to the tree leaves containing the updated *Gctr* and *Lctr*. Despite this, according to [Rogers et al. 2007], this approach decreases the execution time overhead of integrity trees from 12.1% to 1.8% and reduces external memory overhead for node storage from 33.5% to 21.5%.

Duc et al. proposed a similar architecture in [Duc and Keryell 2006] except that instead of using a nonce in the MAC computation, they include a random number; therefore, their architecture, called CryptoPage, is sensitive to replay with a success probability proportional to the size of the random value.

A.4 Memory Authentication with an Untrusted Operating System

In many scenarios, the security policy of a computing platform must exclude the operating system from the trusted computing base. With modern commodity operating systems in particular, it is practically impossible to verify that no exploitable software vulnerability exists in such a large, complex and extendable software system. As a result, the OS cannot be trusted to isolate a sensitive application from malicious software, hence it needs to be considered as untrusted in the threat model (as in our threat model 2). Our Bastion architecture solves this problem by introducing a hypervisor as a trusted manager

[42] The authentication primitive used in [Rogers et al. 2007] is basically the MAC function with nonces presented in section A.3.



(a) Standard Merkle Tree

(b) Bonsai Merkle Tree, BMT

Figure A.6: Bonsai Merkle Tree Principle

of the platform's integrity tree. In this Section, we describe the memory authentication problem in a context where such a trusted hypervisor is not available. We present the Branch Splicing Attack and the Reduced Address Space, two concepts we introduced in our prior work on memory authentication [Champagne et al. 2008].

The first step towards protecting the memory of a sensitive application running on an untrusted OS is to build an integrity tree which covers only pages belonging to the application and which can only be updated when the application itself is running. This precludes the OS, with certain well-defined exceptions for communication, from effecting modifications to the application's memory space that cause the tree to be updated, i.e., any OS write to the application's memory is considered as corruption. Although this approach prevents the OS from carrying out splicing, spoofing and replay attacks through direct writes to the application's memory (because such corruptions would be detected by the integrity tree scheme), [Champagne et al. 2008] shows that the OS can still perform splicing attacks indirectly, by corrupting the application's page table.

The Branch Splicing Attack. The page table is a data structure maintained by the OS, which maps a page's virtual address to its physical address. On a read operation, when a virtual-to-physical address translation is required by the processor, the page table is looked up[43] using the virtual address provided by the running process to obtain the corresponding physical address. The *branch splicing attack* presented in [Champagne et al. 2008] corrupts the virtual-to-physical address mapping for a given memory block. This causes the on-chip integrity verification engine not only to fetch the wrong block in physical memory, but also to use the wrong tree branch in verifying the integrity of the

[43] In a modern processor, the page table is cached on-chip in a Translation Lookaside Buffer. This does not affect the feasibility of the attack described here.

block fetched. [Champagne et al. 2008] shows that with threat model 2, building an integrity tree over the Physical Address Space (PAS tree) is insecure because it is vulnerable to the branch splicing attack. In a PAS tree, the physical address determines the authentication branch to load to re-compute the root during verification. As a result, the OS can corrupt block A's virtual-to-physical address translation to trick the integrity checking engine into using block B's authentication branch to verify block B, hence substituting B for A (In Figure A.7, substituting data at address @0 for data at address @4).

Building a Tree over the Virtual Address Space (VAS-Tree). To defend against this attack, the integrity tree can be built over the Virtual Address Space (VAS tree). In this case, the virtual address generated by the protected application is used to traverse the tree so page table corruption has no effect on the integrity verification process. The VAS tree, unlike a PAS tree, protects application pages that have been swapped out to disk by the OS paging mechanism since it can span all pages within the application's virtual memory space. PAS trees only span physical memory pages: they do not keep track of memory pages sent to the on-disk page file, hence they require a separate mechanism to protect swapped out pages, e.g., [Rogers et al. 2007]. Without such a mechanism, a PAS tree scheme cannot detect corruption of data that might occur during paging—i.e. between the time a page is moved from its physical page frame to the on-disk page file and the time that page is fetched again by the OS, from disk back into a physical page frame.

The VAS tree is not a panacea however, as it presents two major shortcomings: it must span a huge region of memory and it requires one full-blown tree for each application requiring protection, rather than a single tree protecting all software in physical memory. In addition, this solution requires extra tag storage for the virtual address [Suh et al. 2003] in the last level of cache (when implemented) which is usually physically tagged and indexed. Indeed, on cache evictions, the virtual address is required to traverse and update the integrity tree. These shortcomings, and the overheads described next, are the main reasons why the Bastion integrity tree is constructed as a PAS tree, with a separate mechanism (based on the dMap) for securing virtual pages swapped to disk.

Impractical VAS-Tree Overheads. The extra security afforded by the VAS tree over the PAS tree comes at the cost of very large memory capacity and initialization overheads. Application code and data segments are usually laid out very far apart from one another in order to avoid having dynamically growing segments (e.g., the heap and stack) overwrite other application segments. The VAS tree must thus span a very large fraction of the virtual address space in order to cover both the lowest and highest virtual addresses that may be accessed by the application during its execution. The span of the tree is then several orders of magnitude larger than the cumulative sum of all code and data segments that require protection. In the case of a VAS tree protecting a 64-bit address space, the tree span can be so enormous as to make VAS tree impractical, i.e., VAS tree is not scalable. Indeed, it not only requires allocating physical page frames for the 2^{64} bytes of leaf nodes that are defined during initialization, but also requires allocating memory for the non-leaf tree nodes, which represent 20% to 100% of the leaf space size depending on the memory overhead of the underlying integrity tree [Hall and

Jutla 2005, Elbaz et al. 2007, Gassend et al. 2003]. The CPU time required to initialize such a tree is clearly unacceptable in practice.

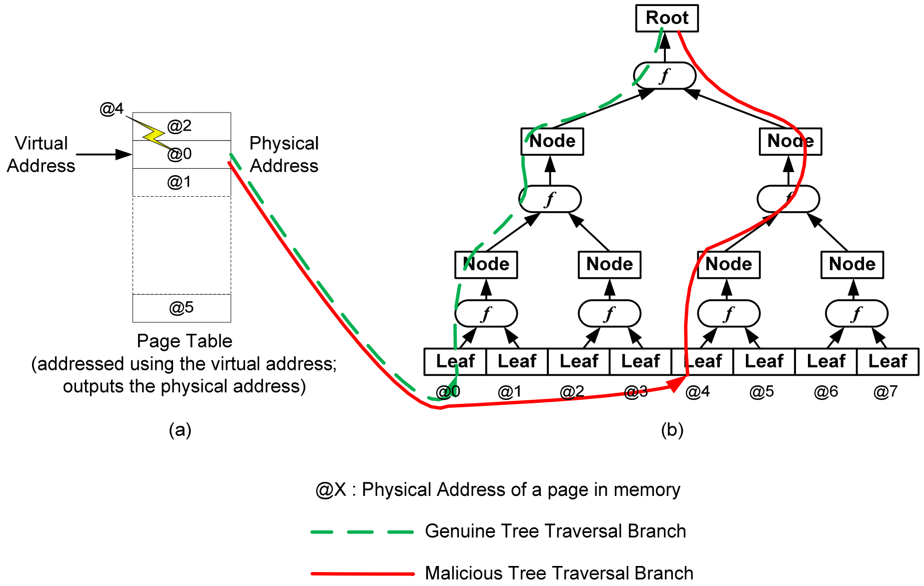

Root
f
Node
Node
f
f
Node
Node
Node
Node
f
f
f
f
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
Leaf
@0
@1
@2
@3
@4
@5
@6
@7
@4
@2
@0
@1
@5
Virtual Address
Physical Address
Page Table
(addressed using the virtual address; outputs the physical address)
(a)
(b)
@X : Physical Address of a page in memory
Genuine Tree Traversal Branch
Malicious Tree Traversal Branch

Figure A.7: The Branch Splicing Attack. (a) The OS tampers with the Page Table and changes the second entry from physical address @0 to @4. (b) data at @4 are verified instead of data at @0; however, since the physical address—which is corrupted—is used to retrieved the branch nodes required to verify the data read, the attack is undetected and data at @4 are considered genuine.

The Reduced Address Space. To overcome these problems, [Champagne et al. 2008] introduces a new processor hardware unit which builds and maintains an integrity tree over a *Reduced Address Space* (RAS). At any point in time, the RAS contains only those pages needed for the application's execution; it grows dynamically as this application memory footprint increases. Virtual pages are mapped into the RAS using an index[44] that follows the page's translation. The new hardware builds an integrity tree over the application's RAS (a RAS tree), carries out the integrity verification and tree update procedures and expands the tree as the underlying RAS grows. Because the RAS contains the application's memory footprint in a contiguous address space segment, the RAS tree does not suffer from the overheads of the VAS tree, built over a sparse address space. The virtual address of a block is used in computing the value of tree nodes along the block's verification path so corruption of the RAS index or the physical address translation by the OS is detected.

With both the VAS and RAS trees, a full-blown integrity tree must be built and maintained for every protected application, as opposed to a single PAS tree for all software. Therefore, a scheme must be designed to manage tree roots for a Merkle Tree scheme, or the on-chip roots and secret keys for PAT and TEC-Tree. [Suh et al. 2003]

[44] This index is a number identifying the page within the RAS that has been assigned to a given virtual page.

presents an implementation where the processor hardware manages the roots of several Merkle Trees stored in dedicated processor registers. To provide some scalability despite a fixed amount of hardware resources, the authors suggest implementing spill and fill mechanisms between these registers and a memory region protected by a master integrity tree.

A.5 Memory Authentication without a Tree Structure

Several engines that are not based on tree schemes have been proposed in the literature to ensure memory authentication. In most cases however, these techniques intentionally decrease the security of the computing platforms to cope with the overhead they generate. This section surveys these techniques.

Lhash. The researchers who proposed the cached tree principle also designed a memory authentication technique not based on a tree and called Log hash (Lhash [Suh et al. 2003b]). Lhash has been designed for applications requiring integrity checking after a sequence of memory operations (as opposed to checking off-chip operations on every memory access as in tree schemes). Lhash relies on an incremental multiset hash function (called Mset-Add-Hash) described by the authors in [Clarke et al. 2003]). This family of hash takes as an input a message of an arbitrary size and outputs a fixed size hash as a regular hash function, but the ordering of the chunks constituting the message is not important. The Lhash scheme has been named after the fact that a write and a read log of the memory locations to be checked are maintained at runtime using an incremental multiset hash function and stored on-chip. The write and read logs are respectively called ReadHash and WriteHash.

The Lhash scheme works as follows for a given sequence of operations. At initialization, WriteHash is computed over the memory chunks belonging to the memory region that needs to be authenticated; WriteHash is then updated at runtime when an off-chip write is performed or when a dirty cache block is evicted from cache. This way, WriteHash reflects the off-chip memory state (and content) at anytime. ReadHash is computed the first time a chunk is brought in cache and updated on each subsequent off-chip read operations. When checking the integrity of the sequence of operations, all the blocks belonging to the memory region to authenticate and not present in cache are read so that the number of chunks in the read log equals the number of chunks in the write log. If ReadHash happens to be different from WriteHash, it means that the memory has been tampered with at runtime. Re-ordering attacks are prevented by including a nonce in each hash computation. Authors performed a comparison of Lhash with a cached hash tree (4-ary). They showed that overhead can be decreased from 33% to 6.25% in term of off-chip memory and from 20-30% to 5-15% at runtime. However, the scheme does not suit threat models such as the ones defined in this appendix since an adversary is able to perform an active attack before the integrity checking takes place.

PE-ICE. A Parallelized Encryption and Integrity Checking Engine, PE-ICE, based on the block-level AREA technique was proposed in [Elbaz et al. 2006] to encrypt and authenticate off-chip memory. However, to avoid re-encryption of the whole memory when the nonce reaches its limit (e.g., a counter that rolls over), the authors propose to replace the nonce with the chunk address concatenated with a random number. For each memory block processed by PE-ICE, a copy of the random value enrolled is kept on-chip to make it tamper-resistant and secret. The drawback of using a random number is that a replay attack can be performed with a probability p of success inversely proportional to the bit length r of the random number ($p = 1/2^r$). Since a small random number is advised [Elbaz 2006] to keep the on-chip memory overhead reasonable, the scheme is likely to be insecure with respect to replay attacks.

A.6 Data Authentication in Symmetric Multi-Processors (SMP)

High-end computing platforms are increasingly based on multi-processor technology. In this section, we highlight the new security challenges related to runtime data authentication that arise on such shared memory platforms.

We have defined memory authentication as “verifying that data read from memory by the processor at a given address is the data it last wrote at this address”. However, in a multiprocessor system with shared memory, the definition of data authentication cannot be restricted to that of memory authentication as defined above for single processors: additional security issues must be considered. First, every processor core of a SMP platform may legitimately update data in memory; thus each processor must be able to differentiate a legitimate write operation done by a core of the system from malicious data corruption. Moreover, in addition to the traditional memory authentication, data authentication in an SMP platform must also consider bus transaction authentication on cache-to-cache transfers required in cache coherency protocols (Figure A.8). [Zhang et al. 2005] notes that to take into consideration bus transaction authentication, an additional active attack must be considered: message dropping. In SMP platforms, when a processor sends a data to another core, it broadcasts the same data to all other SMP’s cores to maintain cache coherency. Thus, message dropping takes place upon those broadcasting cache-to-cache communications and consists in blocking a message destined to one of the processor cores (In Figure A.8, CPU2 is temporarily disconnected from the bus to perform a message dropping).

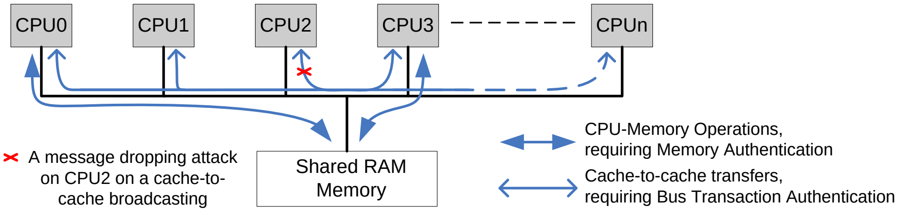

CPU0
CPU1
CPU2
CPU3
CPUn
A message dropping attack on CPU2 on a cache-to-cache broadcasting
Shared RAM Memory
CPU-Memory Operations, requiring Memory Authentication
Cache-to-cache transfers, requiring Bus Transaction Authentication

Figure A.8: Data Authentication Requirements in a Symmetric Multi-Processors (SMP) Platform

[Zhang et al. 2005] and [Shi et al. 2004] propose solutions for data authentication in SMP platforms. However, as highlighted in [Zhang et al. 2005], [Shi et al. 2004] is sensitive to message dropping. Moreover, [Zhang et al. 2005] and [Shi et al. 2004] focus on SMP systems with a small number of processors and do not evaluate the scalability of their solutions. Thus, data authentication at runtime in SMP platforms is still an open research topic.

Active research efforts [Rogers et al. 2006, Lee et al. 2007, Rogers et al. 2008] are also being carried out in the field of data authentication in multiprocessor systems with Distributed Shared Memory (DSM). Data authentication issues in these systems are similar to those mentioned for SMP platforms, except that designers of security solutions have to deal with additional difficulties due to the intrinsic characteristics of the DSM processor. First, a typical DSM system does not include a shared bus that could help synchronization of metadata (e.g., counter values) between the processors participating in a given memory authentication scheme. Also, as mentioned in [Rogers et al. 2008], the interconnection network that enables processor-to-memory communications is usually exposed at the back of server racks, providing an adversary with an easier way to physically connect to the targeted system compared to a motherboard in an SMP platform.

A.7 Appendix Summary

In this appendix, we presented a survey of techniques and engines for memory authentication—i.e., the runtime protection of main memory integrity against physical attacks. We set the emphasis on two solutions we presented in past publications. Our survey presents a framework for comparing the different integrity trees, which is helpful in exploring alternatives to the simple Bastion memory integrity mechanism presented in this thesis.

Bibliography

[Acquisti and Gross 2009] Acquisti, A. and Gross, R. Predicting social security numbers from public data, Proceedings of the National Academy of Science, To be Published, 2009.

[Adams and Agesen 2006] Adams, K. and Agesen, O. 2006. A comparison of software and hardware techniques for x86 virtualization. In Proceedings of the 12th international Conference on Architectural Support For Programming Languages and Operating Systems (San Jose, California, USA, October 21 - 25, 2006). ASPLOS-XII. ACM, New York, NY, 2-13.

[Allen 1978] Allen, J. 1978 Anatomy of LISP. Book. McGraw-Hill, Inc.

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

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

[Ames et al. 1983] Ames, S. R., Gasser, M., and Schell, R. R. 1983. Security Kernel Design and Implementation: An Introduction. Computer 16, 7 (Jul. 1983), 14-22.

[Anderson 1972] Anderson, J. "Computer security technology planning study," Air Force Elec. Syst. Div. Rep. ESD-TR-73-51, Oct. 1972.

[Anderson et al. 2007] Anderson, M.J., Moffie M., and C.I. Dalton. Towards Trustworthy Virtualisation Environments: Xen Library OS Security Service Infrastructure. Technical Report HPL-2007-69, Hewlett-Packard Development Company, L.P., April 2007.

[Anderson and Kuhn 1996] Anderson, R. and Kuhn, M. 1996. Tamper resistance: a cautionary note. In Proceedings of the 2nd Conference on Proceedings of the Second USENIX Workshop on Electronic Commerce - Volume 2 (Oakland, California, November 18 - 21, 1996). USENIX Workshop on Electronic Commerce. USENIX Association, Berkeley, CA, 1-1.

[Alves and Felton 2004] T. Alves and D. Felton. Trustzone: Integrated hardware and software security. ARM white paper, July 2004.

[Arbaugh et al. 1997] Arbaugh, W. A., Farber, D. J., and Smith, J. M. 1997. A secure and reliable bootstrap architecture. In Proceedings of the 1997 IEEE Symposium on Security and Privacy (May 04 - 07, 1997). SP. IEEE Computer Society, Washington, DC, 65.

[Armstrong et al. 2005] Armstrong, W. J., Arndt, R. L., Boutcher, D. C., Kovacs, R. G., Larson, D., Lucke, K. A., Nayar, N., and Swanberg, R. C. 2005. Advanced virtualization capabilities of POWER5 systems. IBM J. Res. Dev. 49, 4/5 (Jul. 2005), 523-532.

[Badrignans et al. 2009] Benoît Badrignans, David Champagne, Reouven Elbaz and Lionel Torres, "SARFUM: Security Architecture for Remote FPGA Update and Monitoring", (submitted to ACM TRETS, Transactions on Reconfigurable Technology and Systems – 2nd Revision submitted), 2009.

[Barham et al. 2003] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. 2003. Xen and the art of virtualization. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles(Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM, New York, NY, 164-177.

[BBC 2009] British Broadcasting Corporation (BBC), http://news.bbc.co.uk/2/hi/uk_-news/7581540.stm, 2009.

[Bellare and Namprempre 2008] Bellare, M. and Namprempre, C. 2008. Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm. J. Cryptol. 21, 4 (Sep. 2008), 469-491.

[Berger et al. 2006] Berger, S., Cáceres, R., Goldman, K. A., Perez, R., Sailer, R., and van Doorn, L. 2006. vTPM: virtualizing the trusted platform module. In Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (Vancouver, B.C., Canada, July 31 - August 04, 2006). USENIX Security Symposium. USENIX Association, Berkeley, CA

[Biryukov et al. 2003] Biryukov, A., Lano, J. and Preneel, B. Cryptanalysis of the Alleged SecurID Hash Function, http://eprint.iacr.org/2003/162/, 12 Sep, 2003.

[Blum et al. 1991] Blum, M., Evans, W., Gemmell, P., Kannan, S. and Naor, M., Checking the correctness of memories,Proc. 32nd IEEE Symposium on Foundations of Computer Science, pages 90–99, 1991.

[Borders and Prakash 2007] Borders, K. and Prakash, A. 2007. Securing network input via a trusted input proxy. In Proceedings of the 2nd USENIX Workshop on Hot Topics in Security (Boston, MA). T. Jaeger, M. Blaze, A. D. Keromytis, P. McDaniel, F. Monrose, N. Provos, R. Sailer, L. van Doorn, H. Wang, and S. Zdancewic, Eds. USENIX Association, Berkeley, CA, 1-5.

[Bratus et al. 2008] Bratus, S., D'Cunha, N., Sparks, E., and Smith, S. W. 2008. TOCTOU, Traps, and Trusted Computing. In Proceedings of the 1st international Conference on Trusted Computing and Trust in information Technologies: Trusted Computing - Challenges and Applications (Villach, Austria, March 11 - 12, 2008). P. Lipp, A. Sadeghi, and K. Koch, Eds. Lecture Notes In Computer Science, vol. 4968. Springer-Verlag, Berlin, Heidelberg, 14-32.

[Caceres et al. 2005] Cáceres, R., Carter, C., Narayanaswami, C., and Raghunath, M. 2005. Reincarnating PCs with portable SoulPads. In Proceedings of the 3rd international Conference on Mobile Systems, Applications, and Services (Seattle, Washington, June 06 - 08, 2005). MobiSys '05. ACM, New York, NY, 65-78.

[Carr 2005] Carr, N.G. The end of corporate computing, MIT Sloan Management Review 46 (3), 2005, pp. 67-73.

[Champagne et al. 2008] Champagne, D., Elbaz, R., and Lee, R. B. 2008. The Reduced Address Space (RAS) for Application Memory Authentication. In Proceedings of the 11th international Conference on information Security (Taipei, Taiwan, September 15 - 18, 2008). T. Wu, C. Lei, V. Rijmen, and D. Lee, Eds. Lecture Notes In Computer Science, vol. 5222. Springer-Verlag, Berlin, Heidelberg, 47-63.

[Champagne et al. 2008b] Champagne, D., Elbaz, R. and Lee, R.B. “Forward-Secure Content Distribution to Reconfigurable Hardware”, In Proceedings of the Int’l Conference on ReConFigurable Computing and FPGAs (Reconfig’08), December 2008.

[Champagne et al. 2009] Champagne, D., Tiwari, A., Wu, W., Hughes, C. J., Kumar, S., Lu, S. “Providing metadata in a Translation Lookaside Buffer (TLB)”, United States Patent Application 20090172243, 2009.

[Champagne and Lee 2010] Champagne, D., Lee, R. B.2010 “Scalable Architectural Support for Trusted Software”, In Proceedings of the 16th IEEE International Symposium on High-Performance Computer Architecture (HPCA), Bangalore, India - January 9-14, 2010. Nominated for Best Paper Award.

[Champagne and Lee 2010b] Champagne, D. and Lee, R. B. “Processor-Based Trustworthy Computing”, Submitted for Review, Usenix Security Symposium, 2010.

[Chari et al. 1999] Chari, S., Jutla, C. S., Rao, J. R., and Rohatgi, P. 1999. Towards Sound Approaches to Counteract Power-Analysis Attacks. In Proceedings of the 19th Annual international Cryptology Conference on Advances in Cryptology (August 15 - 19, 1999). M. J. Wiener, Ed. Lecture Notes In Computer Science, vol. 1666. Springer-Verlag, London, 398-412.

[Chen et al. 2008] Chen, X., Garfinkel, T., Lewis, E. C., Subrahmanyam, P., Waldspurger, C. A., Boneh, D., Dwoskin, J., and Ports, D. R. 2008. Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems. In Proceedings of the 13th international Conference on Architectural Support For Programming Languages and Operating Systems (Seattle, WA, USA, March 01 - 05, 2008). ASPLOS XIII. ACM, New York, NY, 2-13.

[Chen and Lee 2009] Chen, Y. and Lee, R. B. 2009. Hardware-Assisted Application-Level Access Control. In Proceedings of the 12th international Conference on information Security (Pisa, Italy, September 07 - 09, 2009). P. Samarati, M. Yung, F. Martinelli, and C. A. Ardagna, Eds. Lecture Notes In Computer Science, vol. 5735. Springer-Verlag, Berlin, Heidelberg, 363-378.

[Cihula 2007] Cihula, J., Trusted Boot: Verifying the Xen Launch, Xen Summit 07, 2007.

[Clark et al. 2005] Clark, C., Fraser, K., Hand, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. 2005. Live migration of virtual machines. In Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2 (May 02 - 04, 2005). USENIX Association, Berkeley, CA, 273-286.

[Clarke et al. 2003] D. Clarke, S. Devadas, M. van Dijk, B. Gassend, and G. E. Suh. Incremental multiset hash functions and their application to memory integrity checking. In Asiacrypt 2003 Proceedings, November 2003.

[Cox et al. 2009] Mark J. Cox, Ralf S. Engelschall, Stephen Henson, and Ben Laurie. openSSL. http://www.openssl.org/, 2009.

[Crosby et al. 2002] Crosby, S., Goldberg, I., Johnson, R., Song, D. X., and Wagner, D. 2002. A Cryptanalysis of the High-Bandwidth Digital Content Protection System. In Revised Papers From the ACM Ccs-8 Workshop on Security and Privacy in Digital Rights Management T. Sander, Ed. Lecture Notes In Computer Science, vol. 2320. Springer-Verlag, London, 192-200.

[Dewan et al. 2008] Dewan, P., Durham, D., Khosravi, H., Long, M., and Nagabhushan, G. 2008. A hypervisor-based system for protecting software runtime memory and persistent storage. In Proceedings of the 2008 Spring Simulation Multiconference (Ottawa, Canada, April 14 - 17, 2008). Spring Simulation Multiconference. Society for Computer Simulation International, San Diego, CA, 828-835.

[Dierks 1999] T. Dierks, "The TLS Protocol Version 1.0", RFC 2246, January 1999.

[DietLibC 2010] DietLibC, http://www.fefe.de/dietlibc/, Accessed on 1/25/2010.

[Duc and Keryell 2006] Duc, G. and Keryell, R. 2006. CryptoPage: An Efficient Secure Architecture with Memory Encryption, Integrity and Information Leakage Protection. In Proceedings of the 22nd Annual Computer Security Applications Conference (December 11 - 15, 2006). ACSAC. IEEE Computer Society, Washington, DC, 483-492.

[Dwoskin and Lee 2007] Dwoskin, J. S. and Lee, R. B. 2007. Hardware-rooted trust for secure key management and transient trust. In Proceedings of the 14th ACM Conference on Computer and Communications Security (Alexandria, Virginia, USA, October 28 - 31, 2007). CCS '07. ACM, New York, NY, 389-400.

[Dyer et al. 2001] Dyer, J. G., Lindemann, M., Perez, R., Sailer, R., van Doorn, L., Smith, S. W., and Weingart, S. 2001. Building the IBM 4758 Secure Coprocessor. Computer 34, 10 (Oct. 2001), 57-66.

[Elbaz 2006] Elbaz, R., “Hardware Mechanisms for Secured Processor Memory Transactions in Embedded Systems”, PhD Thesis, University of Montpellier, December 2006.

[Elbaz et al. 2006] Elbaz, R., Torres, L., Sassatelli, G., Guillemin, P., Bardouillet, M. and Martinez, A., “A Parallelized Way to Provide Data Encryption and Integrity Checking on a Processor-Memory Bus”, Proceedings of the 43rd Design Automation Conference DAC, July 2006.

[Elbaz et al. 2007] Elbaz, R., Champagne, D., Lee, R. B., Torres, L., Sassatelli, G., and Guillemin, P. 2007. TEC-Tree: A Low-Cost, Parallelizable Tree for Efficient Defense Against Memory Replay Attacks. In Proceedings of the 9th international Workshop on Cryptographic Hardware and Embedded Systems (Vienna, Austria, September 10 - 13, 2007). P. Paillier and I. Verbauwhede, Eds. Lecture Notes In Computer Science, vol. 4727. Springer-Verlag, Berlin, Heidelberg, 289-302.

[Elbaz et al. 2009] Elbaz, R., Champagne, D., Gebotys, C., Lee, R. B., Potlapally, N., and Torres, L. 2009. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. In Transactions on Computational Science Iv: Special Issue on Security in Computing, M. L. Gavrilova, C. J. Tan, and E. D. Moreno, Eds. Lecture Notes In Computer Science, vol. 5430. Springer-Verlag, Berlin, Heidelberg, 1-22.

[Elkaduwe et al. 2008] Elkaduwe, D., Klein, G., and Elphinstone, K. 2008. Verified Protection Model of the seL4 Microkernel. In Proceedings of the 2nd international Conference on Verified Software: theories, Tools, Experiments (Toronto, Canada, October 06 - 09, 2008). N. Shankar and J. Woodcock, Eds. Lecture Notes In Computer Science, vol. 5295. Springer-Verlag, Berlin, Heidelberg, 99-114.

[Elphinstone et al. 2007] Elphinstone, K., Klein, G., Derrin, P., Roscoe, T., and Heiser, G. 2007. Towards a practical, verified kernel. In Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (San Diego, CA, May 07 - 09, 2007). G. Hunt, Ed. USENIX Association, Berkeley, CA, 1-6.

[England et al. 2003] England, P., Lampson, B., Manferdelli, J., Peinado, M., and Willman, B. 2003. A Trusted Open Platform. Computer 36, 7 (Jul. 2003), 55-62.

[Freier et al. 1996] A. Freier, P.Karlton and P.Kocker, "The SSL Protocol, Version 3.0", Internet Draft, 1996.

[Fruhwirth 2005] C. Fruhwirth, “New Methods in Hard Disk Encryption”, Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, 2005.

[Garfinkel and Shelat 2003] Garfinkel, S. L. and Shelat, A. 2003. Remembrance of Data Passed: A Study of Disk Sanitization Practices. IEEE Security and Privacy 1, 1 (Jan. 2003), 17-27.

[Garfinkel et al. 2003] Garfinkel, T., Pfaff, B., Chow, J., Rosenblum, M., and Boneh, D. 2003. Terra: a virtual machine-based platform for trusted computing. In Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (Bolton Landing, NY, USA, October 19 - 22, 2003). SOSP '03. ACM, New York, NY, 193-206.

[Garfinkel et al. 2007] Garfinkel, T., Adams, K., Warfield, A., and Franklin, J. 2007. Compatibility is not transparency: VMM detection myths and realities. In Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (San Diego, CA, May 07 - 09, 2007). G. Hunt, Ed. USENIX Association, Berkeley, CA, 1-6.

[Garriss et al. 2007] Garriss, S., Cáceres, R., Berger, S., Sailer, R., van Doorn, L., and Zhang, X. 2007. Towards Trustworthy Kiosk Computing. In Proceedings of the Eighth IEEE Workshop on Mobile Computing Systems and Applications (March 08 - 09, 2007). HOTMOBILE. IEEE Computer Society, Washington, DC, 41-45.

[Gassend et al. 2003] Gassend, B., Suh, G. E., Clarke, D., van Dijk, M., and Devadas, S. 2003. Caches and Hash Trees for Efficient Memory Integrity Verification. In Proceedings of the 9th international Symposium on High-Performance Computer Architecture (February 08 - 12, 2003). HPCA. IEEE Computer Society, Washington, DC, 295.

[Gebotys 2006] Gebotys, C. H. 2006. A split-mask countermeasure for low-energy secure embedded systems. ACM Trans. Embed. Comput. Syst. 5, 3 (Aug. 2006), 577-612.

[Goguen and Meseguer 1982] Goguen, J. A. and Meseguer, J. 1982. Security policies and security models. In Proceedings of the 1982 IEEE Computer Society Symposium on Research in Security and Privacy (Oakland, CA, May), IEEE Computer Society Press, Los Alamitos, CA, 11-20.

[Goldreich 1987] Goldreich, O. 1987. Towards a theory of software protection and simulation by oblivious RAMs. In Proceedings of the Nineteenth Annual ACM Symposium on theory of Computing (New York, New York, United States). A. V. Aho, Ed. STOC '87. ACM, New York, NY, 182-194.

[Gosling et al. 2000] Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000 Java Language Specification, Second Edition: the Java Series. 2nd. Book. Addison-Wesley Longman Publishing Co., Inc.

[Grohoski 2006] Grohoski, G., Niagara-2: A highly threaded server-on-a-chip. In HotChips 18: Proceedings of the Eighteenth IEEE HotChips Symposium on High-Performance Chips, 2006.

[Halcrow 2007] Halcrow, M. 2007. eCryptfs: a stacked cryptographic filesystem. Linux J. 2007, 156 (Apr. 2007), 2.

[Halderman et al. 2008] Halderman, J. A., Schoen, S. D., Heninger, N., Clarkson, W., Paul, W., Calandrino, J. A., Feldman, A. J., Appelbaum, J., and Felten, E. W. 2008. Lest we remember: cold boot attacks on encryption keys. In Proceedings of the 17th Conference on Security Symposium (San Jose, CA, July 28 - August 01, 2008). USENIX Association, Berkeley, CA, 45-60.

[Hall and Jutla 2005] W. E. Hall and C. S. Jutla. Parallelizable authentication trees. In Selected Areas in Cryptography SAC 2005: 95-109

[Hoglund and Butler 2005] Hoglund, G. and Butler, J. 2005 Rootkits: Subverting the Windows Kernel. Book. Addison-Wesley Professional.

[Huang 2003] Huang, A. B. 2003 Hacking the Xbox: an Introduction to Reverse Engineering. Book. No Starch Press.

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

[Intel 2007] Intel, “Intel® Trusted Execution Technology: Preliminary Architecture Specification”, http://www.intel.com, August 2007.

[Intel 2009] Intel 64 and IA-32 Architectures Software Developer’s Manual. 2009.

[Intel 2009b] Intel, Intel® 4 Series Chipset Family Datasheet, Document Number 319970-006, October 2009.

[Intel et al. 2006] Intel Corporation, International Business Machines Corporation, Matsushita Electric Industrial Co., Ltd., Microsoft Corporation, Sony Corporation, Toshiba Corporation, The Walt Disney Company, Warner Bros., “Advanced Access Content System (AACS) - Blu-ray Disc Prerecorded Book”, Revision 0.912, July 27, 2006, http://www.aacsla.com/specifications/AACS_Spec_BD_Prerecorded_0.912.pdf.

[Ishai et al. 2003] Ishai, Y., Sahai, A. and Wagner, D. Private circuits: Securing hardware against probing attacks. In Dan Boneh, editor, Advances in Cryptology—CRYPTO 2003, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[Ivens 2008] Ivens, K. 2008 Quickbooks 2008: the Official Guide. 1. Book. McGraw-Hill, Inc. 2008.

[Jaeger et al. 2006] Jaeger, T., Sailer, R., and Shankar, U. 2006. PRIMA: policy-reduced integrity measurement architecture. In Proceedings of the Eleventh ACM Symposium on Access Control Models and Technologies (Lake Tahoe, California, USA, June 07 - 09, 2006). SACMAT '06. ACM, New York, NY, 19-28.

[Kammller 2008] Kammller, F. 2008 Interactive Theorem Proving in Software Engineering. Book. VDM Verlag.

[Karger et al. 1991] Karger, P. A., Zurko, M. E., Bonin, D. W., Mason, A. H., and Kahn, C. E. 1991. A Retrospective on the VAX VMM Security Kernel. IEEE Trans. Softw. Eng. 17, 11 (Nov. 1991), 1147-1165.

[Karger and Schnell 2002] Karger, P. A. and Schell, R. R. 2002. Thirty Years Later: Lessons from the Multics Security Evaluation. In Proceedings of the 18th Annual Computer Security Applications Conference (December 09 - 13, 2002). ACSAC. IEEE Computer Society, Washington, DC, 119.

[Kauer 2007] Kauer, B. 2007. OSLO: improving the security of trusted computing. In Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium (Boston, MA, August 06 - 10, 2007). N. Provos, Ed. USENIX Association, Berkeley, CA, 1-9.

[Kocher et al. 2004] Kocher, P., Lee, R., McGraw, G., and Raghunathan, A. 2004. Security as a new dimension in embedded system design. In Proceedings of the 41st Annual Design Automation Conference (San Diego, CA, USA, June 07 - 11, 2004). DAC '04. ACM, New York, NY, 753-760.

[Kongetira et al. 2005] Kongetira, P., Aingaran, K., and Olukotun, K. Niagara: A 32-way multithreaded Sparc processor. IEEE Micro, 25(2):21--29, 2005.

[Kuhn 1998] Kuhn, M. G. 1998. Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP. IEEE Trans. Comput. 47, 10 (Oct. 1998), 1153-1157.

[Kumar et al. 2008] Kumar, A., Patel, M.B., Tseng, K, Thomas, R.M., Tallam, M., Chopra, A., Smith, N.M., Grawrock, D.W., Champagne, D., Method and Apparatus to Re-create trust model after sleep state, United States Patent Application 20080244292, October 2008.

[Kwan and Durfee 2007] Kwan, P. C. S. and Durfee, G. Practical uses of virtual machines for protection of sensitive user data. In Information Security Practice and Experience Conference (ISPEC'07), Springer, 2007, 145-161.

[Laudon 2006] Laudon, J., UltraSPARC T1: Architecture and Physical Design of a 32-threaded General Purpose CPU, Pro-ceedings of the ISSCC Multi-Core Architectures, Designs, and Implementation Challenges Forum, 2006.

[Lee and Champagne 2010] Lee, R. B. and Champagne, D. “System and Method for Processor-Based Security”, United States Patent Application, 2010.

[Lee et al. 2005] Lee, R. B., Kwan, P. C., McGregor, J. P., Dwoskin, J., and Wang, Z. 2005. Architecture for Protecting Critical Secrets in Microprocessors. In Proceedings of the 32nd Annual international Symposium on Computer Architecture (June 04 - 08, 2005). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, 2-13.

[Lee et al. 2007] Lee, M., Ahn, M., Kim, E. J. I2SEMS: Interconnects-Independent Security Enhanced Shared Memory Multiprocessor Systems. PACT 2007: p. 94-103.

[Levin et al. 2009] Levin, T. E., Dwoskin, J. S., Bhaskara, G., Nguyen, T. D., Clark, P. C., Lee, R. B., Irvine, C. E., and Benzel, T. V. 2009. Securing the Dissemination of Emergency Response Data with an Integrated Hardware-Software Architecture. In Proceedings of the 2nd international Conference on Trusted Computing (Oxford, UK, April 06 - 08, 2009). L. Chen, C. J. Mitchell, and A. Martin, Eds. Lecture Notes In Computer Science, vol. 5471. Springer-Verlag, Berlin, Heidelberg, 133-152.

[Levy 2004] Levy, E. 2004. Approaching Zero. IEEE Security and Privacy 2, 4 (Jul. 2004), 65-66.

[Lie et al. 2000] Lie, D., Thekkath, C., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., and Horowitz, M. 2000. Architectural support for copy and tamper resistant software. SIGPLAN Not. 35, 11 (Nov. 2000), 168-177.

[Liedtke 1995] Liedtke, J. 1995. Address space sparsity and fine granularity. SIGOPS Oper. Syst. Rev. 29, 1 (Jan. 1995), 87-90.

[Loscocco and Smalley 2001] Loscocco, P. and Smalley, S. 2001. Integrating Flexible Support for Security Policies into the Linux Operating System. In Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (June 25 - 30, 2001). C. Cole, Ed. USENIX Association, Berkeley, CA, 29-42.

[Lutz 2006] Lutz, M. 2006 Programming Python. Book. O'Reilly Media, Inc.

[Lyle 2002] J. Lyle, “HDCP: what it is and how to use it,” EDN, http://www.edn.com/index.asp?layout=articlePrint&articleID=CA209091, 18 Apr. 2002. Accessed 11/18/2009

[Madnick and Donovan 1973] Madnick, S. E. and Donovan, J. J. 1973. Application and analysis of the virtual machine approach to information system security and isolation. In Proceedings of the Workshop on Virtual Computer Systems (Cambridge, Massachusetts, United States, March 26 - 27, 1973). ACM, New York, NY, 210-224.

[Marchesini et al. 2004] Marchesini, J., Smith, S. W., Wild, O., Stabiner, J., and Barsamian, A. 2004. Open-Source Applications of TCPA Hardware. In Proceedings of the 20th Annual Computer Security Applications Conference (December 06 - 10, 2004). ACSAC. IEEE Computer Society, Washington, DC, 294-303.

[McCarty 2004] McCarty, B. 2004 Selinux: Nsa's Open Source Security Enhanced Linux. Book. O'Reilly Media, Inc.

[McCune et al. 2006] McCune, J. M., Perrig, A., and Reiter, M. K. 2006. Bump in the ether: a framework for securing sensitive user input. In Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference (Boston, MA, May 30 - June 03, 2006). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 17-17.

[McCune et al. 2008] McCune, J. M., Parno, B., Perrig, A., Reiter, M. K., and Seshadri, A. 2008. How low can you go?: recommendations for hardware-supported minimal TCB code execution. In Proceedings of the 13th international Conference on Architectural Support For Programming Languages and Operating Systems (Seattle, WA, USA, March 01 - 05, 2008). ASPLOS XIII. ACM, New York, NY, 14-25.

[McCune et al. 2009] McCune, J. M., Perrig, A., and Reiter, M. K. 2009. Seeing-Is-Believing: using camera phones for human-verifiable authentication. Int. J. Secur. Netw. 4, 1/2 (Feb. 2009), 43-56.

[McGrew and Viega 2004] McGrew, D. and Viega, J. The Galois/Counter Mode of Operation (GCM), NIST Modes of Operation Process, 2004.

[Menon et al. 2005] Menon, A., Santos, J. R., Turner, Y., Janakiraman, G., and Zwaenepoel, W. 2005. Diagnosing performance overheads in the xen virtual machine environment. In Proceedings of the 1st ACM/USENIX international Conference on Virtual Execution Environments (Chicago, IL, USA, June 11 - 12, 2005). VEE '05. ACM, New York, NY, 13-23.

[Merkle 1980] R.C. Merkle, “Protocols for Public Key Cryptosystems,” IEEE Symposium on Security and Privacy, 1980, 122-134.

[Microsoft 2009] Microsoft. BitLocker Drive Encryption Overview. http://technet.microsoft.com/en-us/library/cc732774.aspx. Accessed on 11/18/2009.

[Musa 2004] Musa, J. D. 2004 Software Reliability Engineering: More Reliable Software Faster and Cheaper. Book. Authorhouse.

[Naor and Shamir 1994] Naor, M., Shamir, A. Visual cryptography, in: A. De Santis (Ed.), Advances in Cryptology---EUROCRYPT'94, Lect. Notes Comput. Sci., vol. 950, Springer, Berlin, 1994, pp. 1-12.

[NCLYS 2009] Hardware-Enabled Trust, National Cyber Leap Year Summit 2009, Co-Chairs’ Report, August 17-19, 2009.

[Nelson et al. 2005] Nelson, M., Lim, B., and Hutchins, G. 2005. Fast transparent migration for virtual machines. In Proceedings of the Annual Conference on USENIX Annual Technical Conference (Anaheim, CA, April 10 - 15, 2005). USENIX Annual Technical Conference. USENIX Association, Berkeley, CA, 25-25.

[NICTA 2010] National Information and Communications Technology Australia, Proof Statistics (L4.verified security OS kernel), http://ertos.nicta.com.au/research/l4.verified/numbers.pml, Accessed on 2/17/2010.

[Nipkow et al. 2002] Nipkow, T. et al. Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Book. Springer Verlag, 2002.

[NIST 1995] National Institute of Science and Technology (NIST). Secure hash standard. In Federal Information Processing Standard. National Institute of Standards and Technology, April 1995.

[NIST 2002] NIST, FIPS PUB 180-2: “Secure Hash Standard”, August 2002.

[NIST 2007] NIST Special Publication SP800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC, November 2007.

[Oney 2003] Oney, W. Programming the Microsoft Windows Driver Model, Book. Second ed. Microsoft Press, Redmond, WA. 2003

[Oprea et al. 2004] Oprea, A., Balfanz, D., Durfee, G., and Smetters, D. K. 2004. Securing a Remote Terminal Application with a Mobile Trusted Device. In Proceedings of the 20th Annual Computer Security Applications Conference (December 06 - 10, 2004). ACSAC. IEEE Computer Society, Washington, DC, 438-447.

[Osvik et al. 2005] Osvik, D. A., Shamir A. and Tromer, E. "Cache attacks and Countermeasures: the Case of AES", Cryptology ePrint Archive, Report 2005/271, 2005.

[Ozment and Schechter 2006] Ozment, A. and Schechter, S. E. 2006. Milk or wine: does software security improve with age?. In Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (Vancouver, B.C., Canada, July 31 - August 04, 2006). USENIX Security Symposium. USENIX Association, Berkeley, CA

[Pering et al. 2003] Pering, T., Sundar, M., Light, J., and Want, R. 2003. Photographic Authentication through Untrusted Terminals. IEEE Pervasive Computing 2, 1 (Jan. 2003), 30-36.

[PGP 2009] PGP Corporation, PGP Whole Disk Encryption.
http://www.pgp.com/products/wholediskencryption/index.html, Accessed on 12/30/2009

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

[Rogers et al. 2006] Rogers, B., Prvulovic, M. and Solihin, Y. Effective Data Protection for Distributed Shared Memory Multiprocessors, In Proc. of International Conference of Parallel Architecture and Compilation Techniques (PACT), September 2006.

[Rogers et al. 2007] Rogers, B., Chhabra, S., Prvulovic, M., and Solihin, Y. 2007. Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors OS- and Performance-Friendly. In Proceedings of the 40th Annual IEEE/ACM international Symposium on Microarchitecture (December 01 - 05, 2007). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 183-196.

[Rogers et al. 2008] Rogers, B., Yan, C., Chhabra, S., Prvulovic, M. and Solihin, Y. Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors, In Proc. of the 14th International Symposium on High Performance Computer Architecture (HPCA)," 2008.

[RSA 2009] RSA SecurID. http://www.rsa.com/node.aspx?id=1156 Accessed on 12/30/2009.

[Rutkowska 2006] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. Presented at Black Hat USA, Aug. 2006.

[Sadeghi and Stüble 2004] Sadeghi, A. and Stüble, C. 2004. Property-based attestation for computing platforms: caring about properties, not mechanisms. In Proceedings of the 2004 Workshop on New Security Paradigms (Nova Scotia, Canada, September 20 - 23, 2004). NSPW '04. ACM, New York, NY, 67-77.

[Sailer et al. 2004] Sailer, R., Zhang, X., Jaeger, T., and van Doorn, L. 2004. Design and implementation of a TCG-based integrity measurement architecture. In Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13 (San Diego, CA, August 09 - 13, 2004). USENIX Security Symposium. USENIX Association, Berkeley, CA, 16-16.

[Sailer et al. 2005] Sailer, R., Jaeger, T., Valdez, E., Caceres, R., Perez, R., Berger, S., Griffin, J. L., and Doorn, L. v. 2005. Building a MAC-Based Security Architecture for the Xen Open-Source Hypervisor. In Proceedings of the 21st Annual Computer Security Applications Conference (December 05 - 09, 2005). ACSAC. IEEE Computer Society, Washington, DC, 276-285.

[Saulsbury 2008] A. Saulsbury, Inc. UltraSPARC Virtual Machine Specification. Sun MicroSystems. May 2008.

[Seagate 2006] Seagate Corporation. Drivetrust technology: A technical overview. http://www.seagate.com/docs/pdf/whitepaper/TP564_ DriveTrust_Oct06.pdf.

[Shankar et al. 2006] Shankar, U. Jaeger, T. and Sailer, R. Toward automated information-flow integrity for security-critical applications. In Proceedings of the 13th Annual Network and Distributed Systems Security Symposium Internet Society, 2006.

[Sharp et al. 2006] Sharp, R., Scott, J. and Beresford, A. Secure Mobile Computing via Public Terminals. In Proc. of the International Conference on Pervasive Computing, 2006.

[Shi et al. 2004] W. Shi, H.-H. Lee, M. Ghosh, and C. Lu. Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems. In Proc. of the 13th International Conference on Parallel Architectures and Compilation Techniques, 2004.

[Smith and Austel 1998] Smith, S. W. and Austel, V. 1998. Trusting trusted hardware: towards a formal model for programmable secure coprocessors. In Proceedings of the 3rd Conference on USENIX Workshop on Electronic Commerce - Volume 3 (Boston, Massachusetts, August 31 - September 03, 1998). USENIX Workshop on Electronic Commerce. USENIX Association, Berkeley, CA, 8-8.

[Smith and Nair 2005] Smith, J. E. and Nair, R. 2005. The Architecture of Virtual Machines. Computer 38, 5 (May. 2005), 32-38.

[Strongin 2005] G. Strongin, “Trusted computing using AMD “pacifica” and “presidio” secure virtual machine technology,” Elsevier, Information Security Technical Report, vol. 10, no. 2, 2005.

[Sugerman et al. 2001] Sugerman, J., Venkitachalam, G., and Lim, B. 2001. Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor. In Proceedings of the General Track: 2002 USENIX Annual Technical Conference (June 25 - 30, 2001). Y. Park, Ed. USENIX Association, Berkeley, CA, 1-14.

[Suh et al. 2003] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M., and Devadas, S. 2003. AEGIS: architecture for tamper-evident and tamper-resistant processing. In Proceedings of the 17th Annual international Conference on Supercomputing (San Francisco, CA, USA, June 23 - 26, 2003). ICS '03. ACM, New York, NY, 160-171.

[Suh et al. 2003b] Suh, G. E., Clarke, D., Gassend, B., van Dijk, M. and Devadas, S. Efficient Memory Integrity Verification and Encryption for Secure Processors, In Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO36), pages 339-350, San Diego, CA, December 2003.

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

[Sun 2006] Sun Microsystems, Inc., OpenSPARC T1 Microarchitecture Specification, Part No. 819-6650-10, August 2006, Revision A.

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

[Sun 2010] Sun Microsystems, Inc., Sun Studio C, C++ and Fortran Compilers and Tools, http://developers.sun.com/sunstudio, accessed on 1/5/2010.

[Tanenbaum 2007] Tanenbaum, A. 2007 Modern Operating Systems. 3rd. Book. Prentice Hall Press.

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

[TCG 2009] Trusted Computing Group, http://www.trustedcomputing-group.org , Accessed 11/18/2209

[Tiri and Verbauwhede 2005] Tiri, K. and Verbauwhede, I. 2005. Design Method for Constant Power Consumption of Differential Logic Circuits. In Proceedings of the Conference on Design, Automation and Test in Europe - Volume 1 (March 07 - 11, 2005). Design, Automation, and Test in Europe. IEEE Computer Society, Washington, DC, 628-633.

[Verbauwhede and Schaumont 2007] Verbauwhede, I. and Schaumont, P. 2007. Design methods for security and trust. In Proceedings of the Conference on Design, Automation and Test in Europe (Nice, France, April 16 - 20, 2007). Design, Automation, and Test in Europe. EDA Consortium, San Jose, CA, 672-677.

[Toegl 2009] Toegl, R. 2009. Tagging the Turtle: Local Attestation for Kiosk Computing. In Proceedings of the 3rd international Conference and Workshops on Advances in information Security and Assurance (Seoul, Korea, June 25 - 27, 2009). J. H. Park, H. Chen, M. Atiquzzaman, C. Lee, T. Kim, and S. Yeo, Eds. Lecture Notes In Computer Science, vol. 5576. Springer-Verlag, Berlin, Heidelberg, 60-69.

[Venners 1999] Venners, B. 1999 Inside the Java Virtual Machine. 1st. Book. McGraw-Hill Professional.

[Verisign 2008] VeriSign® Managed Public Key Infrastructure Service, Verisign Data Sheet 2008.

Verisign 2009] http://www.verisign.com

[VI 2009] Virtual Internet, http://www.vi.net/

[Wang and Lee 2007] Wang, Z. and Lee, R. B. 2007. New cache designs for thwarting software cache-based side channel attacks. In Proceedings of the 34th Annual international Symposium on Computer Architecture (San Diego, California, USA, June 09 - 13, 2007). ISCA '07. ACM, New York, NY, 494-505.

[Wang and Lee 2008] Zhenghong Wang and Lee, R. B. 2008. A novel cache architecture with enhanced performance and security. In Proceedings of the 2008 41st IEEE/ACM international Symposium on Microarchitecture (November 08 - 12, 2008). International Symposium on Microarchitecture. IEEE Computer Society, Washington, DC, 83-93.

[Watson 2008] Watson, J. 2008. VirtualBox: bits and bytes masquerading as machines. Linux J. 2008, 166 (Feb. 2008), 1.

[Weaver 2008] David L. Weaver, “OpenSPARC Internals”, Lulu, 2008.

[Weingart 2000] Weingart, S. H. 2000. Physical Security Devices for Computer Subsystems: A Survey of Attacks and Defences. In Proceedings of the Second international Workshop on Cryptographic Hardware and Embedded Systems (August 17 - 18, 2000). Ç. K. Koç and C. Paar, Eds. Lecture Notes In Computer Science, vol. 1965. Springer-Verlag, London, 302-317.

[Wheeler 2010] Wheeler, D., SLOCCount, http://www.dwheeler.com/sloccount/, Accessed on 1/22/2010.

[Wojtczuk and Rutkowska 2009] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel® Trusted Execution Technology. Black Hat DC, February 18-19, 2009

[Wright et al. 2002] Wright, C., Cowan, C., Smalley, S., Morris, J., and Kroah-Hartman, G. 2002. Linux Security Modules: General Security Support for the Linux Kernel. In

Proceedings of the 11th USENIX Security Symposium (August 05 - 09, 2002). D. Boneh, Ed. USENIX Security Symposium. USENIX Association, Berkeley, CA, 17-31.

[Yan et al. 2006] Yan, C., Englender, D., Prvulovic, M., Rogers, B., and Solihin, Y. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In Proceedings of the 33rd Annual international Symposium on Computer Architecture (June 17 - 21, 2006). International Symposium on Computer Architecture. IEEE Computer Society, Washington, DC, 179-190.

[Ye and Smith 2002] Ye, Z. and Smith, S. 2002. Trusted Paths for Browsers. In Proceedings of the 11th USENIX Security Symposium (August 05 - 09, 2002). D. Boneh, Ed. USENIX Security Symposium. USENIX Association, Berkeley, CA, 263-279.

[Zeldovich et al. 2006] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D. 2006. Making information flow explicit in HiStar. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (Seattle, Washington, November 06 - 08, 2006). Operating Systems Design and Implementation. USENIX Association, Berkeley, CA, 263-278.

[Zhang et al. 2005] Zhang, Y., Gao, L., Yang, J., Zhang, X. and Gupta, R.. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. In Proc. of the 11th International Symposium on High-Performance Computer Architecture, 2005.

[Zhuang et al. 2004] Zhuang, X., Zhang, T., and Pande, S. 2004. HIDE: an infrastructure for efficiently protecting information leakage on the address bus. In Proceedings of the 11th international Conference on Architectural Support For Programming Languages and Operating Systems (Boston, MA, USA, October 07 - 13, 2004). ASPLOS-XI. ACM, New York, NY, 72-84.