CHAPTER 12

Hardware and Security: Vulnerabilities and Solutions

Gedare Bloom, Eugen Leontie, Bhagirath Narahari, Rahul Simha

12.1. INTRODUCTION

This chapter introduces the role that computer hardware plays for attack and defense in cyber-physical systems. Hardware security – whether for attack or defense – differs from software, network, and data security because of the nature of hardware. Often, hardware design and manufacturing occur before or during software development, and as a result, we must consider hardware security early in product life cycles. Yet, hardware executes the software that controls a cyber-physical system, so hardware is the last line of defense before damage is done – if an attacker compromises hardware then software security mechanisms may be useless. Hardware also has a longer lifespan than most software because after we deploy hardware we usually cannot update it, short of wholesale replacement, whereas we can update software by uploading new code, often remotely. Even after hardware outlives its usefulness, we must dispose of it properly or risk attacks such as theft of the data or software still resident in the hardware. So, hardware security concerns the entire lifespan of a cyber-physical system, from before design until after retirement.

In this chapter, we consider two aspects of hardware security: security in the processor supply chain and hardware mechanisms that provide software with a secure execution environment. We start by exploring the security threats that arise during the major phases of the processor supply chain (Section 12.2). One such threat is the Trojan circuit, an insidious attack that involves planting a vulnerability in a processor sometime between design and fabrication that manifests as an exploit after the processor has been integrated, tested, and deployed as part of a system. We discuss ways to test for Trojan circuits (Section 12.2.1), how design automation tools can improve the trustworthiness of design and fabrication to reduce the likelihood of successful supply chain attacks (Section 12.2.2), defensive techniques that modify a computer processor's architecture to detect runtime deviations (Section 12.2.3), and how software might check the correctness of its execution by verifying the underlying hardware (Section 12.2.4).

We begin the second aspect of hardware security – how hardware can support software to provide secure execution throughout a cyber-physical system's lifetime – by introducing how hardware supports secure systems (Section 12.3). One contribution of hardware is that it can implement security mechanisms with smaller performance penalties than software implementations. Memory isolation techniques rely on hardware to enforce isolation primitives efficiently (Section 12.3.1). Cryptographic accelerators use hardware parallelism to provide speedup to cryptography primitives and protocols (Section 12.3.3). Another contribution of hardware is that it can provide a trusted base for computing: if the hardware is trustworthy or secure, then we can build a secure system from trusted hardware primitives. Granted if the hardware is compromised by some of the attacks presented in Section 12.2, then the trusted base is not trustworthy and our system would be insecure. Secure coprocessors can be a trusted base for computing if they are manufactured properly, protected from physical compromise, and provide an isolated platform that implements security mechanisms – for example, cryptographic primitives – for use by the rest of the system (Section 12.3.4). Physical compromise of a computer introduces sophisticated attacks that target otherwise unobservable components like system bus traffic and memory contents; one countermeasure against physical attack is to place the processor in a tamper-proof enclosure and encrypt (decrypt) everything that exits (enters) the processor (Section 12.3.5). Even with the above security measures, if an application contains an exploitable security vulnerability, then malicious code can enter the system. Hardware techniques can mitigate the potential that software vulnerabilities are exploitable by protecting an application from the software-based attacks (Section 12.3.2). We conclude this chapter with some areas for future work and exercises that demonstrate the concepts of hardware security.

12.2. HARDWARE SUPPLY CHAIN SECURITY

For years, Moore's Law has predicted the success of the semiconductor integrated circuit (IC or chip) manufacturing business: a new IC can be produced, which has twice as many transistors as a similar IC made less than 2 years prior. With each new generation of smaller transistor sizes, IC fabrication plants, called foundries or fabs, must update their tools to support the new technology, an expensive cost to incur every other year.

The increase in costs and demand for new chips, coupled with decreased production costs overseas, has led to the globalization of the semiconductor design and fabrication industry, which in turn raises concerns about the vulnerability of ICs to subversion. In particular, how can chip manufacturers, who still design and sell but no longer fabricate chips, verify that the chips they sell are authentic to the designs? Currently, this problem is solved by trust: end users trust chip manufacturers, who trust chip fabricators. But what if chip fabrication is not trustworthy?

Concerns about IC fabrication seem to have originated in the military sector, with DARPA issuing a Broad Agency Announcement (BAA) in 2006, and again in 2007, requesting proposals for the TRUST in Integrated Circuits (TIC) program [1, 2]. Adee [3] gives an overview of the TIC program and IC supply chain problems. Semiconductor Equipment and Materials International (SEMI) also has acknowledged the problems inherent in outsourced fabrication and industry stratification [4].

Possible attacks on the IC supply chain include maliciously modifying ICs, copying of ICs to produce cloned devices, or stealing intellectual property (IP). A malicious modification to an IC is called a *Trojan circuit* or *Hardware Trojan*, because the modification hides itself within the IC and possibly even provides correct, useful functionality before it attacks, much like its eponymous myth. Cloned or counterfeit ICs lead to lost revenue and may be less reliable than the original, potentially affecting the reputation of the original device manufacturer and the security of the end user. IP theft is similar to counterfeiting but may be used to create competing devices without incurring the research and development costs. All of these attacks may cause financial harm to the original designer or device manufacturer, while

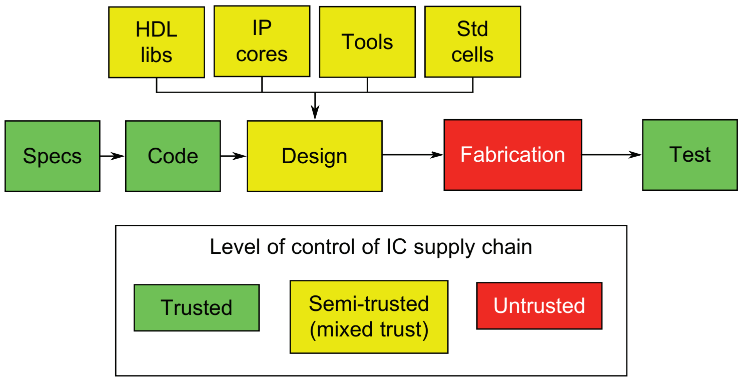

FIGURE 12-1 IC Supply Chain. End-users trust chip designers, whose name appears on the final product, but design houses outsource much of their work to lesser known companies, hence introducing semitrusted and untrusted phases. These phases introduce vulnerabilities that attackers can exploit before the designer incorporates its IC in a device destined for end-users.

also introducing potential risks to end users due to loss of reliability and security.

Supply chain attacks can happen during any of the untrusted phases prior to deployment within an electronic device. The primary phases of the IC supply chain, shown in Figure 12-1, are design, fabrication, and test. Categorical levels of trust are assigned based on the end user point-of-view, with the assumption that chip designers are trusted.

The IC design phase consists of all the code and other inputs to the tools that will generate the specification for the foundry processes. Some of the design processes are observable and auditable, and therefore trusted. However, third party IP is increasingly used in a protected manner such that designers and consumers cannot verify the contents of the IP. Thus, the design phase has mixed levels of trust, and IC design is vulnerable to supply chain attacks. Fabrication processes are similarly untrusted due to a lack of verifiability and trust at offshore foundries. Only the test phase is trusted in its entirety.

The test phase ensures each IC passes a set of test procedures that check for manufacturing faults based on the known design. Because the IC may be modified in unknown ways that may activate malicious activity under rare circumstances, the current test phase cannot detect Trojan circuits. Furthermore, IP theft and device cloning occur before the test phase, so it is not sufficient for detecting any of the above attacks, even if it is trusted.

As designing circuits has become more complicated, commercial pressures have driven technologies that improve time-to-market. One such technology is reconfigurable logic, for example, the field programmable gate array (FPGA) devices. A reconfigurable device can be reprogrammed after it is deployed. Another technology that has gained momentum is *soft IP*, which are components of circuit design that are deployed as source code, for example, as a library of hardware description language (HDL) files. Both reconfigurable logic and soft IP make IP theft and cloning easier.

Reconfigurable logic is both a boon and a burden for designing secure circuits. On the positive side, reconfigurability decouples manufacturing from design, in contrast to the sequential supply chain demonstrated in Figure 12-1. Thus, a sensitive design does not need to be sent to the foundry. On the negative side, the design and deployed device can be more easily accessed by an attacker than a fixed-logic design. For bitstream reconfigurable devices, such as FPGAs, the security of the design depends on the security of the bitstream. Trimberger [5] details some of the security concerns and solutions for FPGA designs of trusted devices.

The following sections discuss how research is attempting to solve the problems of the modern IC supply chain.

12.2.1. Testing for Trojan Circuits

One approach to Trojan circuit detection is to look for additions and modifications to an IC in a laboratory setting prior to deploying the final product. Lab-based detection augments the test phase of the supply chain with *silicon design authentication*, which verifies that an IC contains only the designer's intended functionality and nothing else [6]. Silicon design authentication is difficult due to increasing IC complexity, shrinking feature sizes, rarity of conditions for activating malicious logic, and lack of knowledge about what functionality has been added to the IC. Testing cannot capture the behavior of added functionality, so an IC could contain all of the intended functionality while having extra malicious logic that might not be activated or detected by tests. Detection by physical inspection is insufficient due to the complexity and size of modern ICs. Destructive approaches are too costly and cannot be applied to every IC. Detection based on physical characteristics of the IC are imprecise because fabrication introduces physical variations that increase as ICs shrink. Two promising directions that are active areas for research in the area of Trojan circuit detection at the silicon design authentication step are side-channel analysis and Trojan activation.

Side-channel analysis relies on variations in signals, usually in the analog domain, that an IC emits and a Trojan circuit would affect – changes to the behavior or physical characteristics of an IC may change non-behavioral traits when the IC is tested. For example, the presence of a Trojan circuit may cause deviations in a chip's power dissipation, current, temperature, or timing. Seminal work by Agrawal et al. [7] uses power analysis for detection of Trojan circuits; they also suggest some other side-channels to investigate. Other research expands on side-channel analysis by refining power-based approaches for Trojan detection [8–10], investigating other side-channel detection techniques [11, 12], and characterizing the gate-level behaviors of an IC [13, 14].

Trojan activation techniques attempt to trigger a Trojan circuit during silicon design authentication to make the malicious behavior observable or to improve side-channel analysis techniques. A motivating assumption is that attackers are likely to target the least-activated circuitry in an IC, so researchers have explored methods for generating inputs that activate an IC where Trojan circuits are likely to be hidden [15–17]. Other techniques attempt to explore the state space of the circuit in new ways, for example, via randomization [18] or by partitioning the IC into regions [9, 19] in which areas of the IC are isolated and then tested for Trojan activity.

The silicon design authentication techniques are useful for detecting the presence of malicious circuitry prior to deploying devices. Physical measures, such as leakage current or path delay, characterize chips and can expose anomalies that may indicate malicious logic. Usually, the characteristics under measurement must be known for a "golden" chip that is fabricated securely or modeled based on the IC specifications and manufacturing processes. Physical variations in the manufacturing process makes side-channel analysis probabilistic, and IC complexity hinders activation techniques. Approaches that combine the two techniques appear to offer good tradeoffs in terms of accuracy and cost. In the following section, techniques for circuit design are reviewed that may assist in silicon design authentication by improving the initial design phase of the IC supply chain.

12.2.2. Design for Hardware Trust

Computer-aided design (CAD) and electronic design automation (EDA) tools are crucial to the efficiency and success of circuit design. These tools, however, neglect IC supply chain problems. This section describes ways to improve design tools and processes to defend against IC supply chain attacks.

Design tools focus mainly on *design for test (DFT)* or *design for manufacturability (DFM)*. Researchers have proposed that the design phase be augmented to improve hardware trust, introducing the notion of *design for hardware trust (DFHT)*. *DFHT* seeks to prevent Trojan circuits from being introduced during design or fabrication. An example of a *DFHT* technique is to modify the design tools such that low probability transitions in a circuit are increased [20, 21], which presumably frustrates an attacker's job of finding an input that can trigger a Trojan circuit yet is unlikely to occur in testing.

Watermarking. Watermarking is an established technique for marking property such that counterfeit goods are detectable; the next chapter discusses digital watermarks for detecting software piracy at greater length. IC watermarking [22] helps to detect counterfeit devices and IP theft. Metadata are embedded within the IC design, so the manufactured product contains an identifiable watermark that does not affect the circuit's functionality. Effective watermarking requires that the watermark is not easy to copy. A downside to watermarking is that pirated devices must be re-acquired and verified by checking the watermark.

Lach et al. [23, 24] propose watermarks for FPGA design components with the intent of identifying a component's vendor and consumer. FPGA designs are challenging to watermark because of the flexibility of the architecture. Distributing the output of cryptographic hash functions across the design will raise the barrier to attacks that attempt to reconfigure the FPGA to copy or remove watermarks.

Soft IP cores ease the design process, but they also reveal the hardware design of a component in its entirety. A challenge is to watermark soft IP without losing the benefit of IP blocks. Yuan et al. [25] describe this problem and evaluate practical solutions at the Verilog HDL level, demonstrating that watermarks can be embedded in soft IP that is recoverable in the synthesized design. Lin et al. [26] address the challenge of combining FPGA and soft IP designs. Castillo et al. [27] consider a signature-based approach that can protect against IP theft of both soft and hard IP cores.

Fingerprints, PUFs, and Metering. Building on the notion of watermarking, a device fingerprint is a unique feature of a design applied to a device. Fingerprinting improves on watermarking by enabling tracing of stolen property so that after discovering a counterfeit device punitive measures may be taken. Caldwell et al. [28] show how to generate versions of IP cores that synthesize with identical functionality but have some other measurably unique properties, a key to good fingerprinting.

Variations inherent in the IC manufacturing processes give rise to techniques for device fingerprinting. Physical device fingerprinting via manufacturing variability [29, 30] and physically unclonable functions (PUFs) [31, 32] can provide IC authentication. A PUF can act in a challenge-response protocol to provide IC authentication based on unclonable physical characteristics resulting from the variation inherent in manufacturing processes. Suh and Devadas [33] demonstrate that PUFs are also useful as a secure source of a random secret, which can be used to generate cryptographic keying material, in addition to IC authentication. Simpson and Schaumont [34] show how to use PUFs to authenticate FPGA-based systems without an online verifier.

IC metering extends device fingerprinting to restrict production and distribution of ICs. The goal of metering is to prevent attackers from copying a design and creating cloned devices. Metering information is extracted from the IC via manufacturing variability or PUFs and registered with the design company. When an IC is recovered and suspected of being pirated, it can be checked against the registered ICs.

Metering can be either passive or active. Effective passive IC metering was proposed by Koushanfar et al. [35] and is sufficient to state whether or not a device is pirated. Active IC metering is proposed by Lee et al. [36], making use of wire delay characteristics to generate a secret key unique to the circuit.

Active IC metering techniques improve on passive metering by preventing pirated devices from functioning. Activation logic is embedded in the metered IC that requires keying material not included in the manufactured device. The intent of active IC metering is to prevent cloned devices from operating without receiving the keying material from the designer. For ASIC designs, the EPIC technique [37] is a comprehensive solution for active metering that combines chip locking and activation using public-key cryptography. For FPGAs, the work of Couture and Kent [38] makes a first step by supporting IP licensing with expirations. Baumgarten et al. [39] demonstrate how to prevent IC piracy by introducing barriers in the reconfigurable logic with small overhead. The main advantage of active metering combined with FPGA technology is that the IC design need not be shared with the foundry.

Verifying Design and Fabrication. Much of the research in security for the IC supply chain focuses on the foundry as untrusted and therefore the main source of IC supply chain attacks. Two directions push back on this assumption by supposing either (1) the design phase is also susceptible to attack or (2) the foundry is semi-trusted.

As an example of (1), Di and Smith [40, 41] consider attacks on the design side of the supply chain and introduce tools that work together to detect malicious circuitry that is added at design time. The goal of these tools is to protect from viruses in design software and from flawed third party IP.

Bloom et al. [42] consider how the foundry can add forensic information to its processes, similar to IC metering techniques, but with additional information to assist in auditing occurrences of piracy. Forensic information is gathered during fabrication and is embedded on the IC so that forensic recovery tools can extract information about counterfeit ICs that are recovered. This approach is not able to prevent piracy, much like watermarking and passive IC metering.

12.2.3. Architectural Techniques

In contrast to the design and verification techniques of the previous two sections, architectural support for silicon design authentication attempts to prevent or detect the malicious behavior of a Trojan circuit at run-time. One straightforward architectural solution is to measure the IC's physical characteristics, much like in side-channel analysis, and then use those measurements on-line by designing circuits that report the measurements [43]. Similarly, the well-studied fault-tolerant technique of replicating entire processing elements can help to detect some types of Trojan circuit attacks [44]. Researchers have also proposed some specific architectural techniques to detect Trojan circuits.

BlueChip [45] is a hybrid design-time and run-time system that introduces unused circuit identification to detect suspect circuitry and replaces suspect circuits with exception generation hardware. An exception handler emulates the elided circuit's functionality, thus avoiding the untested components of an IC in which Trojan circuits are likely to be hidden.

SHADE [46] is a hardware–software approach that uses multiple ICs as guards in a single board with the assumption at least two ICs come from different foundries, so malicious circuitry would not collude between two or more ICs. Hardware and software protocols use the two-guard architecture to protect applications with pseudorandom heartbeats and two layers of encryption. These techniques are meant to prevent data exfiltration and detect denial-of-service attacks.

Similar concern for sharing information is explored in the context of FPGA technology by Huffmire et al. [47]. They describe a simple isolation primitive (moat) that can ensure separation of cores after the place-and-route stage of chip manufacturing. Moats can isolate functionally independent chip components and provide a framework for designing secure systems out of untrusted components. To enable communication between the disparate cores, the authors present a shared memory bus (drawbridge).

Architectural techniques are complementary to design and verification techniques. Indeed, the use of multiple techniques may improve overall Trojan circuit detection. Side-channel analysis performs well against large modifications, but small Trojan circuits can hide in the noise of the side-channel. Thus, applying side-channel analysis can bound the size of undetected Trojan circuits and assist architectural techniques by imposing realistic assumptions on the possible functionality of the Trojan circuit.

12.2.4. Can Software Check Hardware?

Let us now consider how software might check the validity of the hardware on which it executes. A challenge in this direction is that, if the hardware is incorrect, software has difficulty in verifying the results of such checks.

For detecting Trojan circuits, Bloom et al. [48] propose adding some OS routines that cause off-chip accesses or checks. A secondary chip processes these checks to verify the primary CPU. Checks are an OS mechanism to challenge the CPU in collaboration with an on-board verifier.

Software can also check other aspects of the hardware. SoftWare-based attestation (SWATT) [49] is a technique for verifying the integrity of memory and establishing the absence of malicious changes to memory. SWATT relies on an external verifier in a challenge-response protocol. The verifier issues a challenge to an embedded device, which must compute and return the correct response. The correctness of the response depends on the device's clock speed, memory size and architecture, and the instruction set architecture (ISA). A limitation of SWATT's threat model is that physical attacks are not considered. In particular, an assumption is that an attacker is unable to modify the device's hardware. If the hardware itself is compromised or replaced wholesale, for example by a faster CPU or memory, then SWATT might verify the platform incorrectly. The problem is that SWATT relies on the timing of the hardware for its verification, but if the timing is different then the approach to verification may fail.

Deng et al. [50] propose a method for authenticating hardware by establishing tight performance bounds of real hardware and ensuring that the runtime characteristics of the hardware are at those bounds. Thus, a processor design is authenticated (and not emulated) if its performance matches the imposed constraints. This approach depends on the details of the hardware's microarchitecture and supposes that emulating the ISA with precise timing of the microarchitecture is difficult.

Software-based checks can measure hardware for correctness. A limiting assumption to all such approaches is that the correct measurements are known. Developing a predictive model for the measurement would be a positive step forward in this line of research. Integrating software-based measurements with trusted hardware designs can also improve platform security for high-assurance applications.

12.3. HARDWARE SUPPORT FOR SOFTWARE SECURITY

In Section 12.2, we saw that a determined and resourceful attacker can compromise hardware. Software is an easier target to attack. Compromised software has but one goal: to abuse computing resources. Such abuse can take multiple forms, including corruption of critical code or data, theft of sensitive data, and eavesdropping or participating in system activities. Despite the presence of malicious attacks, cyber-physical systems should be resilient and continue to operate correctly. In the following sections, we investigate how computer hardware can help to prevent malicious attacks from compromising software. We start with software-based attacks and defenses, then we discuss how hardware improves the performance and security of cryptography, and we finish with countermeasures to physical attacks that rely on hardware access.

A simple attack is to access an application's memory from another application. Code and data vulnerabilities entice attackers to try gaining unfettered access to execute malicious software or to steal sensitive information. Traditional defenses against memory-based attacks isolate execution contexts – for example, processes and virtual memory address spaces – to prevent one context from accessing another's code and data without permission. In Section 12.3.1, we discuss established and new techniques for isolating memory with hardware support.

Even if we isolate every application in a system and follow good security practices (strong passwords, good cryptography, physically secured hardware), attacks might still compromise system security by exploiting vulnerabilities within the applications. An exploitable security vulnerability in a software application is program behavior that an attacker can trigger to circumvent the system's security policies. One such vulnerability, the buffer overflow, is widely regarded as the most commonly exploited vulnerability (on par with database injections and cross-site scripting). Good programming practices – bounds checking, input validation – can prevent exploitable vulnerabilities, but enforcing such practices is not always easy. Software solutions for buffer overflow detection – for example, StackGuard [51] – provide decent protection with little overhead. Buffer overflow prevention, however, incurs non-negligible overheads with software-only solutions – for example, Return Address Defender [52] and dynamic instruction stream editing [53] – because protection relies on page tables and virtual memory protection. Saving return addresses to read-only memory pages can prevent a buffer overflow from subverting the control flow, but the calls to change page permissions incur a high overhead. When protecting the return address, permission changes twice for every function call, once to allow writing the return address and then back to read-only for the duration of function execution. Hardware can improve the speed and reduce the overhead of buffer overflow prevention. In Section 12.3.2, we analyze hardware techniques to mitigate buffer overflow attacks; we also examine hardware-based information flow tracking, which prevents untrusted (low-integrity) inputs from propagating to trusted (high-integrity) outputs thereby preventing unsanitized input from affecting program behavior.

After we secure our systems from software attacks, our next concern is the security of sensitive data in networked systems. Data communication networks have a wide attack surface that requires encryption for securing sensitive data. Transmission mediums do little to deter attacks: wireless communication is a broadcast medium and cables can be tapped with little effort. With cryptography, we can create end-to-end secure channels across an unsecured physical environment. Unfortunately, software implementations of cryptographic primitives come at a high cost, especially in resource-constrained devices. Perhaps worse than the performance cost is the possibility of side-channel vulnerabilities – see Section 12.2.1 – that arise due to power and timing variations outside the control of software. A dedicated hardware accelerator for cryptographic primitives can outperform most software implementations and can reduce the likelihood of side-channel vulnerabilities. We describe the advantages and challenges – maximize throughput and minimize latency while keeping resource utilization (size, weight, and power) small – of cryptographic accelerators further in Section 12.3.3.

Although cryptographic accelerators improve the performance of cryptography and help to prevent side-channel attacks, the danger that the device might fall into the attacker's possession still remains. Even with moderately sophisticated equipment, an attacker with physical access to hardware can steal data or subvert security measures. One solution for defending against physical attack is secure coprocessing. General requirements for a secure coprocessor are (1) to protect secret information such as cryptographic keys from physical attack, (2) to provide a host system with cryptographic operations that resist side-channel analysis, and (3) to provide a safe execution environment for applications. The first requirement is typically achieved with physical protection techniques: fuses, temperature sensors, and other tamper detection techniques that can trigger destruction of all secrets. Built-in cryptographic primitives, the second requirement, are needed so that secret keys need not leave the coprocessor, since it implements the cryptography using those keys. The third requirement, a secure and authenticated execution environment, refines a secure coprocessor's architecture as a computing device with private memory, processor, and software that can execute a wide range of applications. Section 12.3.4 further explores the algorithms and example implementations of secure coprocessors and briefly discusses less complex alternatives.

Related to secure coprocessing are encrypted execution and data (EED) platforms, which encrypt all storage (including memory) and only decrypt within the CPU boundary. EED platforms counter attacks that probe or control system buses, control memory content [54], or attempt to reconstruct an application's control flow. Section 12.3.5 describes attacks on EED platforms and their countermeasures.

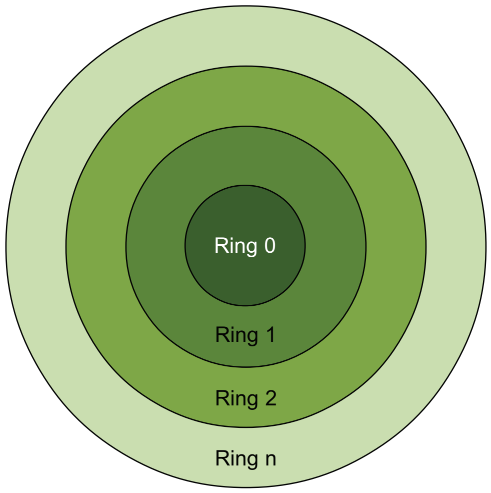

FIGURE 12-2 Privilege rings (Levels). The innermost ring is the highest privilege at which software can execute, normally used by the OS or hypervisor. The outermost ring is the lowest privilege, normally used by application software. The middle rings (if they exist) are architecture-specific and are often unused in practice.

12.3.1. Memory Protection

Security policies define how users of a computing system are allowed to use, modify, or share its resources. Traditionally, an operating system (OS) defines the security policies and relies on hardware to assist in enforcing them. This section reviews the role of hardware in supporting system security through isolation and control of resources.

Memory Protection in Commodity Systems. Consider an abstract computing device comprising two major components—the processor and memory. The processor fetches code and data from memory to execute programs. If both components are shared by all programs then a program might corrupt or steal another's data in memory, accidentally or maliciously, or prevent any other program from accessing the processor (denial-of-service). To avoid such scenarios, a privileged entity must control the allocation (time and space) of compute resources (processor and memory). Traditionally, the OS kernel is responsible for deciding how to allocate resources, and hardware enforces the kernel's decisions by supporting different privilege rings or levels (Figure 12-2).

By occupying the highest privilege ring, the kernel controls all compute resources. It can read and write from any memory location, execute any instruction supported by the processor, receive all hardware events, and operate all peripherals. User applications – residing in the lowest privilege level – have limited access to memory (nowadays enforced through virtual memory), cannot execute privileged instructions, and can only access peripherals by invoking OS services. A processor differentiates between kernel and user with bits in a special register, generically called the program status word and specifically called the Program Status Register in ARM architectures, Hypervisor State Bit in PowerPC architectures, Privileged Mode in the Processor State Register in SPARC architectures, or Current Privilege Level in the Code Segment Register in Intel architectures.

A hierarchy of control from the most to least privilege, combined with memory access controls, prevents user programs from performing any action outside a carefully sandboxed environment without invoking services (code) in a more privileged level. Control transfer between different privilege rings usually is done with interrupts or specialized control instructions; on more recent Intel architectures, the `sysenter` and `sysexit` instructions allow fast switching during system calls. The OS controls the data structures and policies that control high-level security concepts like users and files, and code running outside the highest privileged ring cannot manipulate them directly. A critical aspect of securing high-level code and data is memory protection.

In simple architectures, the privileged state also defines memory separation. An example of a simple policy could be that user code can only access memory in a specified range such as 0xF000 to 0xFFFF, whereas privileged code can access the full memory range. With one or more fixed memory partitions, privileged code can manage both the allocation and separation of memory among application tasks. Except for embedded and highly customized applications, static memory partitioning is impractical.

A multiprocessing system comprising dynamic tasks, each with distinct (often statically unknown) memory and execution demands, require dynamic memory management to limit memory fragmentation and balance resource utilization. Two practical solutions for dynamic memory management are to use fixed sized blocks (pages) or variable length segments. With either solution, memory accesses must be translated to a physical address and verified for correct access rights. Modern architectures usually provide some support for translation and verification of memory accesses, whether for pages or segments.

Virtual memory with paging is the norm in dynamic memory management. Paging provides each application (process) with a linear contiguous virtual address space. The physical location of data referenced by such an address is computed based on a translation table that the OS maintains. The memory management unit (MMU) is a special hardware unit that helps to translate from virtual to physical addresses, with acceleration in the form of a hardware lookup table called the translation look aside buffer (TLB) and assist in checking access permissions.

The OS maintains a page table for each virtual address space that contains entries of virtual-to-physical pages. Each page table entry contains a couple of protection bits, which are architecture-dependent and either one bit to distinguish between read/write permissions or two encoded bits (three independent bits) for read/write/execute permissions. A process gains access to individual pages based on the permission bits. Because each process has a different page table, the OS can control how processes access memory by setting (clearing) permission bits or by not loading a mapping into the MMU. During a process context switch, however, the OS must flush (invalidate) the hardware that accelerates translation (the translation lookaside buffer or TLB). So, hardware supports paging with protection bits, which generate an exception on invalid accesses and with the TLB for accelerating translations.

Although paged-based virtual memory systems allow for process isolation and controlled sharing, the granularity of permission is coarse – permissions can only be assigned to full pages, typically 4 KB. An application that needs to isolate small chunks of memory must place each chunk in its own page, leading to excessive internal fragmentation.

With segmentation, instead of one contiguous virtual address space per process, we can have multiple variable sized virtual spaces, each mapped, managed, and shared independently. Segmentation-based addressing is a two step process involving application code and processor hardware: code loads a segment selector into a register and issues memory accesses, then, for each memory access, the processor uses the selector as an index in a segment table, obtains a segment descriptor, and computes the access relative to the base address present in the segment descriptor. Access to the segment descriptor is restricted to high-privilege code.

Consider the Intel architecture as an example of segmentation. It uses the code segment (CS) register as a segment selector and stores the current privilege level (CPL) as its two lower bits. When the executing code tries to access a data segment, the descriptor privilege level (DPL) is checked. Depending on whether the loaded segment is data, code, or a system call, the check ensures the CPL allows loading the segment descriptor based on the DPL. For example, a data segment DPL specifies the highest privilege level (CPL) that a task can have in order to be allowed access, so if DPL is 2 then access is granted only to tasks with CPL of 0, 1, or 2. A third privilege level, the requested privilege level (RPL), is used when invoking OS services through call gates and prevents less privileged applications from elevating privileges and gaining access to restricted system segments.

Most open or commercial OSs ignore or make limited use of segmentation. OS/2 [55] is a modern commercial OS that uses the full features of segmentation. Some virtualization techniques (such as the VMWare ESX hypervisor) do reserve a segment for a resident memory area and rely on segment limit checks to catch illegal accesses.

Research Directions in Memory Protection. State-of-the-art research in memory protection generally takes one of two directions. The first builds security applications on top of the popular paging mechanisms. One way to build secure high-level applications with commodity hardware is to use inter-process communication (IPC) mechanisms that rely on process isolation primitives. The Chromium project [56] splits a monolithic user application – a web browser – into isolated processes that can only communicate using OS moderated IPC. The main browser process interfaces most system interactions such as drawing the GUI, storing cookies and history, and accessing the network. Web page rendering executes in a separate process so that any vulnerability or injection attack will not have direct access to the main browser memory space.

The second general research direction aims to address limitations of current architectural support, such as the granularity of protection in paging or the poor performance from the two-step addressing of segmentation. Most work in this direction considers how to support efficient fine-grained data and task security controls.

InfoShield [57] makes a significant step to ensure that only legitimate code can gain access to protected memory regions at a word-level granularity. A hardware reference monitor holds access permissions for a limited number of memory locations. New instructions manage the permissions and allow for creating, verifying, clearing, and securely moving them. InfoShield assumes that only a few security-critical memory locations exist during a process's execution. In typical use cases, InfoShield protects secret keys, passwords, and application-specified secret data from improper access.

A hardware-based table of permissions motivates work to improve how the table is created, stored, and manipulated in memory without degrading performance of permission verification. A data-centric approach designates permissions for every address range representing each memory object. On a memory reference, the corresponding permission is fetched with the content of the referenced memory and hardware checks the access permissions. Permissions can be cached by using the existing hardware [58] or by using a separate caching structure [59, 60].

Mondrian Memory Protection [59] uses a hierarchical trie structure that provides word-level permission granularity. Hardware traverses the trie to check permissions. A TLB-like caching structure accelerates permission checks based on locality. Permissions are encoded compactly, thus minimizing the amount of memory reserved for permission storage. Unfortunately, compaction complicates permission changes and frequent changes to permissions results in frequent trie node insertions and deletions that are hard to support in hardware and slow to run in software. Mondrian aims to protect large code modules whose interaction is mediated by an OS. As such, the OS is invoked every time a security context switch occurs.

Hardware containers [61, 62] are a memory protection mechanism for fine-grained separation of code that maintains word-level data permissions. Containers isolate code regions at the granularity of functions and offer a strict sandbox for faulty code that aids in a controlled recovery from failure. With the aid of compiler tools and runtime instrumentation, a security manifest that dictates permissions for each code region is enforced by a hardware reference monitor. The hardware detects improper use of system resources (unauthorized memory accesses or denial-of-service) and enables applications to undertake a hardware-supervised recovery procedure. Caching and architectural optimizations help to reduce the performance overhead of security enforcement.

Arora et al. [58] show a solution for enforcing application-specified data properties at runtime by tagging data addresses with an additional security tag. For example, using a single bit SECTAG can mark fine grained read-only memory regions. The checker (reference monitor) can be set to interpret and enforce the value of the security tag as specified by a program-wide security policy. One policy example is to label code zones with specific security attributes. All zones with the same label share the access on all data regions with the SECTAG equal or less in value. Caching the security tags changes the cache hierarchy in two ways. First, the L2 cache automatically loads security tags when a cache line is stored. The L1 cache controller expands the cache line with the security attributes for faster checking. The checker directly uses the L1 security tag information for enforcing the policy. Even though the SECTAG length can vary based on the security policy, the length is fixed by the processor architecture: any changes in SECTAG length means changing the internal cache layout and size.

Reconfigurable hardware, like the FPGA, poses a similar challenge to protecting the memory space. While processor-based systems often have some separation mechanisms for protecting processes in sharing the memory space, current systems built based on reconfigurable hardware have a flat memory addressing scheme. Since soft cores are synthesized from a wide source of IP providers, vulnerabilities or malicious cores can issue unauthorized memory requests, corrupting, or revealing sensitive data. One solution is to compile an access policy directly to a synthesized circuit that can act as an execution monitor and enforce a memory access policy [63]. The specialized compiler transforms the access policy from a formal specification to hardware description language that represents the hardware reference monitor.

An area of work related to memory protection is capability-based systems, out of which many modern notions of protection arise. Capabilities in their full form have had limited commercial success, however, so they are not a very active research area. We refer interested readers to Levy's book on the subject [64] for a thorough introduction and history of the topic.

12.3.2. Architectural Support for Control Flow Security

Vulnerabilities in software applications continue to be targets for memory-based attacks – for example buffer overflows – that attempt to gain illegitimate access to code and data. In the following, we examine how hardware helps to prevent buffer overflows. Then, we explain information flow tracking, a general technique for detecting and preventing memory-based attacks.

Architectural Support for Buffer Overflow Defense. Hardware-assisted approaches to buffer overflow protection improve upon accuracy and performance of software-only schemes for dynamic attack detection by using a variety of techniques. One common solution is to maintain a shadow of the return address in hardware (Figure 12-3) by creating a return address stack or monitoring the location of the return address for any unauthorized modifications [65–70]. Other hardware-supported solutions protect all control flow in general, including branches and jumps [71–75].

Secure Return Address Stack (SRAS) [65] implements a shadow stack in hardware with processor modifications including: ISA (Instruction Set Architecture) changes, additional logic, and protected storage. Unlike the usual call stack, the shadow stack only holds return addresses. On a function call (CALL), the return address is pushed to the regular stack and the shadow stack. On a return (RET), SRAS pops and compares the return address from both stacks. To handle function call nesting, the OS is modified to handle secure overflow storage. The spill-over of the

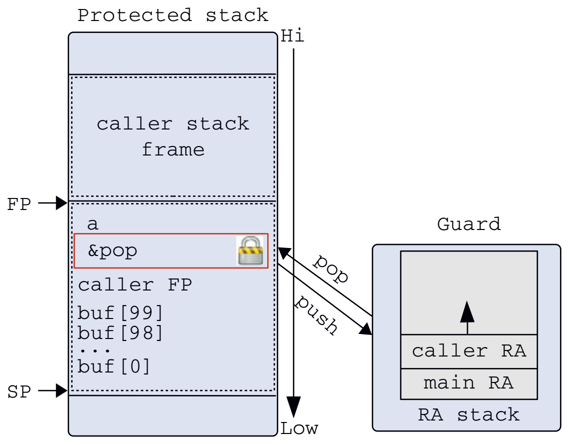

FIGURE 12-3 Stack memory protection moves (stores) the return address to protected storage when a function starts and then restores (checks) the return address before the function returns.

secure stack is stored in a special part of memory that is accessible only to the kernel. The kernel is responsible for managing a secure spill-over stack for each process.

SmashGuard [66], like SRAS, modifies the processor and OS. The new semantics of CALL and RET instructions store a return address inside a memory-mapped hardware stack that is checked on the function's return. The OS includes security functions during context switching and to handle spill-over stack growth.

Xu et al. [67] present a scheme that splits the stack into two pieces: the *control* stack to store the return addresses, and the *data* stack to store the rest. They propose a software solution involving compiler modification, and a hardware solution that modifies the processor and semantics of CALL and RET. In the software-only solution, the compiler allocates and manages the additional control stack. During each function prologue, the compiler saves the return address to the control stack, which resides in memory that the OS securely manages. The compiler restores the return address from the control stack onto the system stack in the function epilog. Simulation yielded significant performance overheads, which led the researchers to a hardware solution.

Secure Cache (SCache) [68] uses cache memory to protect the return address by providing replica cache lines that shadow the return address. A corrupted return address will have a different value in cache than its replica. A drawback to using cache space for storing the return address is that performance is sensitive to cache parameters and behavior.

Kao and Wu [69] present a scheme to detect return address modifications without explicitly storing the good return address for verification. Two registers are added to store the current return address and the previous frame pointer. A valid bit that acts as an exception flag is also added. If a memory store is issued to either a return address on the stack or to the frame pointer, then the flag is raised to signal a violation. When the function returns, if the return address is to the local stack or to the current return address then execution is halted. The authors claim that only two levels of function calls need monitoring.

Using reconfigurable logic such as an FPGA to implement a hardware guard, Leontie et al. [76] provide a secure stack with little change to the processor microarchitecture. The return address is copied to a memory region in a hardware guard that is inaccessible through any direct memory calls. This memory region is called the *return address (RA) stack*, and it is automatically managed by a modified compiler. The compiler-inserted instructions to manage the RA stack synchronize the state of the RA stack with the program stack independent of code locality and caching policies. On each CALL, the return address is stored in the RA stack, and at each RET, the guard checks if an overflow has occurred and ensures that the correct address is returned to the processor. Thus, the guard maintains a copy of the valid return address and ensures that the correct address is always returned regardless of any buffer overflows.

Heap overflows are as dangerous as – if less common than – stack buffer overflows. Dynamically allocated data is stored in memory interleaved with management control data structures and metadata. Heap allocation functions (malloc, free, and new) maintain these structures to track available and occupied memory regions. A heap overflow might just destroy these structures and crash the program, but some allocators are vulnerable to more powerful attacks [77]. When the allocator primitive tries to consolidate freed memory using corrupted heap metadata, an adversary might trick the allocator to write to arbitrary locations in memory. Then, the attacker can override pointers used in indirect jumps to redirect a program's control flow. Solutions to the problems involving heap overflows include fine-grained memory protection, discussed in Subsection 12.3.1, software mechanisms discussed in the next chapter, hardware-assisted bounds checking [78], and hardware-supported programmatic debugging [79].

Information Flow Tracking. Vulnerabilities in software can be hard to find. Since attacks usually enter a system through input channels, researchers propose that instead of verifying code, systems should verify data. Information flow tracking techniques monitor data as it enters a system and prevents any potentially damaging activity that uses untrusted input. Based on the source of data, trust levels are associated with the data. Simply put, data comes from either "trusted" or "untrusted" sources. Depending on program semantics, the information channels are tracked in order to determine if untrusted data affects (taints) the trusted information, or the reverse, if trusted – sensitive or secure – data leaks into untrusted information that can divulge secrets to the insecure channels. Preventing sensitive data from leaking means identifying the communication channels, whether explicit (direct variable assignments) or implicit (control flow, timing, cache content, power etc.), which the data can influence.

Traditional approaches to track information flow through static and dynamic tracking [80–82] have the disadvantage that just a few untrusted sources can *taint* trusted data and cause almost all computations to become untrusted. Maintaining trusted execution in real systems that track taint with many input channels becomes extremely difficult.

Gate level information flow tracking (GLIFT) [83] achieves full control over the information leakage although with limitations on the computation model and increased resource utilization. GLIFT shadows traditional gates with logic to track trust. AND, OR, and MUX gates are augmented to compute the trust level of their output. Higher-level logic is built using these basic gates. Much of the architectural changes and limitations of GLIFT deal with program counter (PC) manipulations. The resulting processor is a slower, larger, and limited core, but one that can guarantee complete information flow tracking.

Flexible (limited programmability) policies for the propagation of taint is possible through architectural modifications [84]. Taint information is associated with each memory word not by widening the width of memory cells but by keeping a compact flat data structure (array of bits) in a separate memory segment. Hardware support includes new pipeline stages and modifications to handle taint. Because the taint algorithm can degrade performance, common operations are cached using a custom unit.

12.3.3. Cryptographic Accelerators

Cryptographic primitives are demanding in terms of computation resources: public key cryptography requires expensive exponentiations; symmetric ciphers use multiple iterations of dictionary lookups and permutations that are sequentially ordered; secure hashes repeat iterative rounds of shifts and permutations. With more consumer applications requiring cryptographic operations in their algorithms for security and privacy, hardware manufacturers propose hardware implementations of these popular primitives. The advantages of using hardware are lower latency for operations, higher throughput for high volume transactions, and lower overall power consumption. As with most hardware implementations, the cost is higher complexity and cost of the hardware, and less flexibility, as silicon space is reserved for the fixed operations. Because hardware design issues are considered when cryptographic standards are chosen, academic and commercial

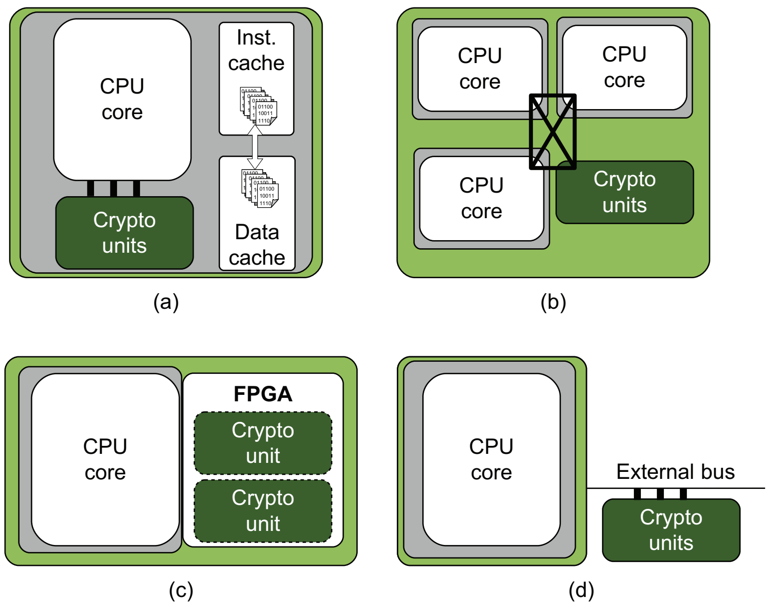

FIGURE 12-4 Configurations for connecting a cryptographic accelerator to a CPU: (a) tightly coupled to the pipeline, (b) attached over internal interconnect, (c) synthesized in specialized reconfigurable logic, and (d) attached as a coprocessor.

implementations of cryptographic protocols are common [85–87].

The challenge and multitude of approaches and solutions are in implementing units that are optimized to offer a maximum throughput and minimum size, latency, and power. Figure 12-4 shows a couple of integration options for connecting a cryptographic accelerator with a CPU, from tightly coupled execution units in the processor pipeline to bus-connected coprocessors.

Some processors are optimized for cryptographic operations [88]. Their execution units have dedicated resources to handle key storage and specific arithmetic operations such as exponentiation or modulus arithmetic. The instruction set supports the cryptographic operations directly. As these are not standalone implementations of cryptographic algorithms, they require support from the compiler to generate optimized code [89]. The main advantage of such an approach is that it is generic to a large set of algorithms and not only cryptography. The disadvantage is cryptographic operations will still exert pressure on the processor pipeline. Other implementation options for cryptographic accelerators include small cores that implement a dedicated function, a generic cryptographic coprocessor that can handle different types of operations, or a general purpose core that is reserved for certain cryptographic algorithms.

One popular integration solution is for the cryptographic engine to work as a coprocessor. Typically connected on standard interfaces with the rest of the system, the processor can off-load specific operations to the coprocessor. Direct memory access (DMA) enables the coprocessor to make data accesses to memory. In contrast to the secure coprocessors described in Section 12.3.4, cryptographic accelerators do not offer guarantees on the physical security of the keying material used and may not even have manufacturer-installed keys.

As with most hardware-implemented algorithms, cryptographic accelerators are exposed to the following problems: bugs in the implementation can cause expensive recalls, changes in standards render old hardware obsolete, and the development cost and time to market are significant. FPGA technology is appealing for implementing cryptographic primitives for several reasons [90–94]. First, although generally slower than application-specific integrated circuit (ASIC) implementations, FPGAs outperform software implementations. Second, the design and deployment cycle is short and cost-effective at small scale. Third, the reconfigurability property of FPGAs allows post-deployment patches for protocol modifications, optimizations, and bug fixes.

12.3.4. Secure Coprocessing

How can sensitive computations be executed in remote devices such that the results obtained are trustworthy? Programs can be altered, data can be modified, records and logs deleted, or secrets revealed. Simulators and debuggers offer mechanisms for observing memory content and execution patterns; memory can be frozen and read; operating systems and compilers may be altered; and even the hardware might harbor a Trojan circuit.

Designers of secure coprocessors argue that a system's root of trust has to be a computational device that can execute its tasks correctly and without disclosing secrets despite any attack (physical or programmatic) [95]. A fast, general purpose computer in a tamper-proof physical enclosure would suffice, but that is hard to achieve. High-end processors consume a lot of power and generate a lot of heat that is difficult to dissipate in a tamper-proof enclosure. Thus, secure coprocessors are limited to low power, limited bandwidth devices incapable of processing at high throughput. Figure 12-5 shows a secure coprocessor as a peripheral to a host computer. The services that the coprocessor offers include safekeeping of cryptographic keys, cryptographic operations, a secure execution environment, and attestation of the host's execution state. A secure coprocessor ensures that a remote device produces trustworthy results.

The enclosure of a secure coprocessor must deter a sophisticated attacker from obtaining any of the internal memory content. Fuses and sensors help to determine whether the physical enclosure has been breached [96]. The usual response to an attack is to erase all memory content. Since extreme temperatures could potentially disable the erasure mechanisms and freeze memory content, designers add thermal sensors to detect such conditions. Other side channels may reveal the secure coprocessor's internal state. Electromagnetic radiation is detected or prevented by the enclosure, but power and heat dissipation remain a concern. Although a limited internal power exists to ensure the erasure procedures, operational power is drawn from the host. To minimize the opportunity for power analysis attacks [97], the coprocessor is equipped with filters on the power supply. Well-designed shielding on the enclosure obfuscates the heat signature.

Having a secure physical enclosure is insufficient to guarantee a unique, unclonable device. At the least, the manufacturer initializes the device with a uniquely certified root key-pair. All other keying material can be derived from this root key and stored in the device memory for application use. Other cryptographic primitives in a secure coprocessor may include a secure random number generator [98], one-way counters, software or hardware implementations of hash functions, or symmetric and asymmetric ciphers [99, 100].

The operational requirements of the secure coprocessors are key management, communication, encryption services, internal resource management, and programmability. These requirements are difficult to implement with dedicated hardware alone. Most solutions come with a complex software stack composed of a secure kernel, hardware assisted protection, resource partitioning, layers of operational services, and user applications. The IBM 4758 [101] is built using a general purpose (x86-compatible) processor, a secure boot mechanism, and a high-level OS. As a complex software stack is hard to validate for correctness, the designers tried to create an architecture that would resist attacks from malicious software executed on the device. They implemented hardware locks that are used to create partitions of memory and

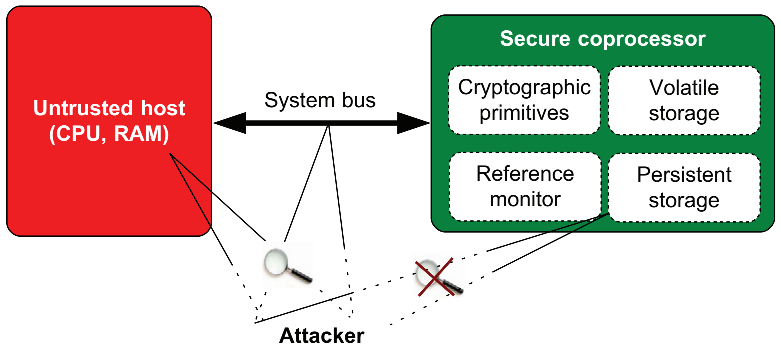

FIGURE 12-5 Secure coprocessing relies on a tamper-proof coprocessor to provide security services to a host system. An attacker is unable to examine the internals of the coprocessor, so secrets stored within are safe.

resources that untrusted software – including privileged OS services in the highest-privilege ring 0 – cannot break [102].

The power of secure coprocessors shows in their application use cases: host integrity validation, secure logs, audit trails, digital rights management (DRM), electronic currency, sandboxing, and attestation for remote agents. We briefly review some of these examples in the following.

Host integrity. Trust in a computing system requires both tamper-proof hardware and privileged software. With viruses, trojans, and malware masquerading in a system, software cannot easily prove itself authentic. If trust in the secure coprocessor is established, then that trust enables authenticating the host. The secure coprocessor must control or verify the initial boot process of the host and take snapshots of the memory state. With this ability, the coprocessor can compute secure hashes of the boot and kernel images, as well as memory states at different milestones in the host execution. By comparing the secure hashes with known-good values, the secure coprocessor can detect memory corruptions.

Secure logs. Logs are an important target for attackers. Forging entries to eliminate intrusion traces is essential for a stealth attack. With the aid of a trusted coprocessor, logs can be made tamper-evident. Cryptographic checksums are the main mechanism, but for sensitive information, such as financial logs, encryption primitives can be used for secrecy.

DRM. Two key properties make secure coprocessors appealing in copyright protection applications. The first is the ability to attest that a host software system is tamper-free, including the operating system and licensed applications. If an attacker (user) cannot circumvent the license validations, then commercial software can reliably check for correct registration procedures. Second are the trustworthy cryptographic primitives, including unique unforgeable identifiers. With these primitives, content providers can deliver encrypted content to the host, limiting the attacker's (including the end user) ability to obtain access to original digital content.

Despite the usefulness of secure coprocessors, their cost and limitations prevent widespread deployment and motivate lighter mechanisms, such as the smartcard and the trusted platform module (TPM). Smartcards have as their main role the safekeeping of a private key. With only a small amount of memory and limited computational power, smartcards store one or more manufacturer-provided cryptographic key pairs and perform basic cryptographic operations. In this regard, they are similar to TPMs. Smartcards connect via standard plug-and-play protocols (USB, infra-red, and short distance radio waves) with the host and offer their services on-demand to the host for authentication, signatures, and other protocols. This is where they differ from TPMs. A typical TPM is closely integrated with its host, plugged into the system buses, and may be able to pause the host execution and take memory snapshots. Smartcards are intended to travel with a person and help authenticate the owner, whereas the TPM remains with the host device and helps to verify its authenticity. Both smartcards and TPMs are cheap because they lack tamper-proof mechanisms, but by the same token they are not secure from an adversary that can mount a physical attack.

12.3.5. Encrypted Execution and Data (EED) Platforms

A physical attack assumes that the attacker has direct physical access to the hardware and is sophisticated and resourceful enough to examine and manipulate instruction and data buses. A simple countermeasure is to encrypt data while it is in memory or storage and only decrypt data as it is fetched by the processor. A platform that encrypts all data entering and exiting the CPU boundary is called an encrypted execution and data (EED) platform.

Figure 12-6 shows a typical layout for an EED platform with an encryption/decryption unit interposing on off-chip interfaces. Since the communication protocol between the processors and external memory does not necessarily align with the typical use of standard cryptographic primitives, adaptations typically are needed for encrypted execution. We motivate the additional data integrity, control flow validation, and authorization mechanisms in the following.

We begin by describing an unauthenticated EED platform that simply encrypts cache blocks and shows why such a mechanism is insufficient against a determined attacker. Since memory chips are external from the CPU, the attacker can supply the processor with arbitrary blocks of data. The most effective form of attack tries to supply the processor with an unexpected block; in doing so, an attacker might then observe the outcome and use that advantageously. For example, an attacker might notice that skipping a certain block leads to skipping a license check. A detailed description of known attacks against this unauthenticated EED platform follows, after which we introduce scholarly work that mitigates such attacks and describe architectural techniques to alleviate performance overheads caused by encryption and authentication.

Code Injection/Execution Disruptions. An attacker may try to modify or replace a portion of an encrypted block of instructions. If the key has not been broken, this attack merely places random bits into a cache block. Nonetheless, these random bits will be decrypted into possibly valid instructions, whose outcome can be observed by a sophisticated attacker. If the instruction set uses n bits for each opcode, there are a total of 2^n possible instructions. If, among these, v is the number of valid instructions, and if the encryption block contains k instructions, then the probability that the decryption will result in at least one invalid instruction in the block is $1-(v/2^n)^k$. Since a good processor architecture avoids wasting opcode space with unused instructions, it is highly probable that the attacker can supply a random block that will be decrypted and executed without detection. For example, if we consider an encryption block size of 16 bytes and if 90% of the opcode space is used for valid instructions, the probability of an undetected disrupted execution is 19%. We term this type of attack *execution disruption through instruction/code injection*. The attacker will not be able to execute arbitrary code, but by observing the program's behavior, the attacker can deduce information about the program's execution. Such code injection attacks suggest the need for run-time code integrity checking in EED platforms, for example by the use of signatures embedded into the executable.

Instruction Replay. An attacker may re-issue a block of encrypted instructions from memory. This can be accomplished either by freezing the bus and replacing the memory read value with an old one, overriding the address bus with a different memory location than the one the processor requested, or simply overwriting the memory at the targeted address. This is illustrated in Figure 12-7(a). In this example, the processor requests blocks A, B, and C from addresses $0x100$, $0x200$, and $0x300$, respectively.

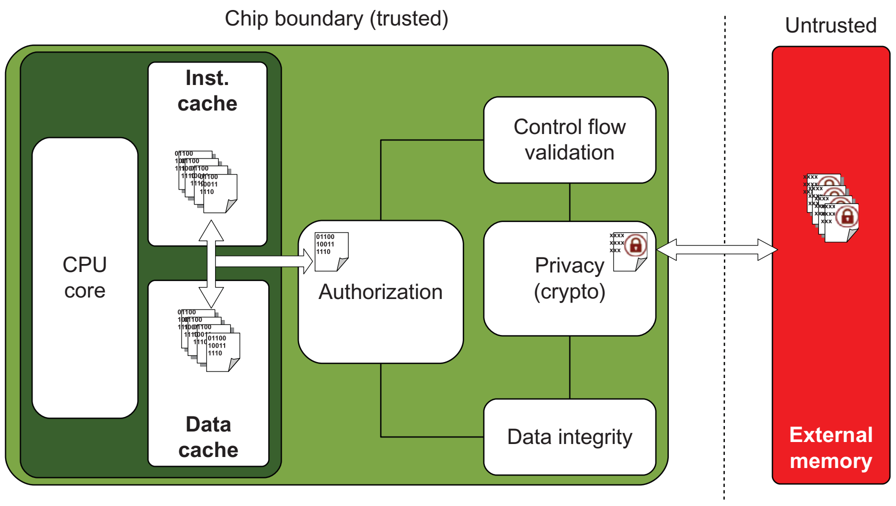

FIGURE 12-6 Encrypted execution and data (EED) platform. Everything that leaves the chip boundary is encrypted, and everything that enters it is decrypted. Control flow validation and data integrity checking, together with proper authorization policies, defend against physical attacks that compromise encryption-only EED platforms.

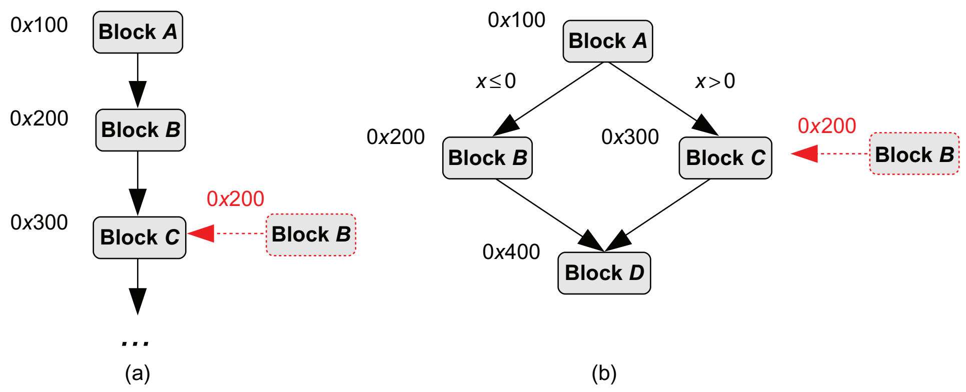

FIGURE 12-7 Example of (a) instruction replay and (b) control flow attacks.

However, when the processor requests block from address 0*x*300, the attacker injects block *B*, which will be correctly decrypted and executed. The incorrect block is decrypted into valid executable code. If the replayed block has an immediate observable result, the attacker can store the block and replay it at any time during program execution. Also, by observing the results of a multitude of replay attacks, the attacker can catalog information about the results of individual replay attacks and use such attacks in tandem for greater disruption or to get a better understanding of the application code. The vulnerability of EED platforms to such replay attacks suggests the need to validate that the contents are indeed from the memory location requested by the processor. We note that simply storing the signature inside the code block does not prevent such attacks, since the code block is unaltered.

Control Flow attacks. An attacker can determine the control-flow structure of a program by watching memory accesses. This control-flow information can lead to both information leakage as well as attacks on the application. As described in Refs. [103, 104], obtaining the block level control flow graph (CFG) can lead to information leakage since CFGs can serve as unique fingerprints of the code and can detect code reuse. By watching the interaction and calling sequences, the attacker can learn more about the application code. Additionally, knowledge of the CFG can also compromise a secret key and leak sensitive data. Since branches compare values, the attacker could force which path to take in the application code. To disrupt the execution or steer the execution in the desired direction, the attacker can transfer control out of a loop, transfer control to a distant part of the executable, or force the executable to follow an existing execution path. For example, consider Figure 12-7(b). During normal execution, block A (at address $0x100$) can transfer control to either block B (address $0x200$) or block C (address $0x300$) depending on the value of some variable x. An attacker who has observed this control flow property can substitute block C when B is requested and observe the outcome as a prelude to further attack. Thus, they can successfully bypass any check condition that may be present in block A. Additionally, another type of attack is when blocks A and B together form a loop. Then, upon observing this once without interference and recording the blocks, the attacker can substitute blocks from an earlier execution to prevent the loop from being completely executed. To protect against control flow attacks requires a mechanism by which the correct control flow, as specified by the application, can be embedded and validated at run-time.

Data injection/modification. By examining the pattern of data write-backs to RAM, an attacker can guess the location of the runtime stack, as commonly used software stack structures are usually simple. Since the attacker has physical access to the device, stack data can be injected and, even though the data will be decrypted into a random stream, the effect on program behavior can be observed. The attacker may still be able to disrupt the execution or learn about the executable.

Data substitution. The attacker can substitute a requested data block for another block, which is also currently valid, and observe the program's behavior. Unlike instructions which are limited to the valid opcodes in the instruction set, any data injected by the attacker will be correctly decrypted and passed to the processor. Thus, encryption in this case provides no protection other than privacy of the application's data. To address this issue, EED platforms typically include a location label (typically a memory address) as part of the validation information.

Stale data replay. An attacker that sniffs the bus can keep snapshots of data and replay old versions of data blocks when newer versions are requested. Data is even more sensitive to replays than code is, as its content changes in time. Any change in the content of the block creates a new valid encryption of that block in memory. To address stale data replay, a time-stamp mechanism records data "freshness" and the validation rejects any old data blocks that are no longer valid.

As described above, these EED attacks point out that encryption is not sufficient to guarantee security against physical attacks, and these types of attacks can go undetected without explicit countermeasures. There are several projects that address the design of EED platforms and provide both data and code integrity against physical attacks. Examples include the XOM architecture [105] and several other tamper-resistant processors [106, 107]; these techniques require re-design of the processor. Under a physical threat model, these systems are still vulnerable to attack.

Others minimize processor changes by using reconfigurable logic [108, 109]. The AEGIS [110] architecture greatly improves on XOM, its predecessor, and presents techniques for control-flow protection, privacy, and prevention of data tampering. AEGIS also includes an optional secure OS that is responsible for interfacing with the secure hardware. The code and memory protection schemes use cached hash trees, and a log-hash integrity verification mechanism is used to verify integrity of memory accesses. The level of confidentiality and integrity protection further improves by merging encrypted execution with access control validation [60].

An attacker can extract patterns of access in an EED platform and match those patterns against a database of known patterns extracted from open-source software or from unencrypted executables run inside a debugger. Algorithms can be identified by observing memory access patterns, and this signature pattern can itself lead to both information leakage as well as additional types of attacks. Address randomization can foil such attacks, and specific architectures for address randomization have been proposed to address this problem [103, 104].

Physical probing of devices is also investigated by a strong community that focuses its research on differential fault (power) analysis (DFA) [97]. Although the attacks target encryption keys of common encryption algorithms [111, 112], other control flow or algorithm designs can be revealed by this approach. EED platforms are still vulnerable to power analysis and benefit from countermeasures such as the DFA resistant implementations of DES and AES [113] or secure IC design [114]. Specific techniques to protect FPGA circuits from DFA also exist [115].

12.4. CONCLUSIONS AND FUTURE WORK

We have described two aspects of computer hardware security: the security of the hardware itself and how hardware enables secure software. Threats to the processor supply chain, such as the Trojan circuit, are emerging as a fundamental problem that faces security practitioners. How do we ensure that attacks do not succeed in the supply chain before our systems are even deployed? How might we create a trustworthy system comprising untrusted components? Traditionally, computer security has relied on hardware as a trusted base for security. We have reviewed how hardware can provide such security even in the face of determined attackers that capture the protected computing devices.

Directions for future work in hardware security abound: the problems are far from solved. Instead of trying to rank future work directions, we discuss possible improvements and paths forward in the order that we presented the topics in this chapter.

Some possible areas to improve security against supply chain attacks include developing Trojan circuit detectors that do not need a reference implementation, improving side-channel analysis so it detects inclusions that try to hide in the noise, incentives for vendors to include DFHT in EDA tool chains, and changing the trust assumptions that software makes about the security of the underlying hardware.

Research in memory protection seems to take one of two directions: use the existing hardware (paging) to build secure systems or improve the existing hardware to support stronger security primitives; in spite of years of research and development, paging dominates commercial systems. Whether fine-grained memory protection can be made efficient, cost-effective, and commercially viable (for both hardware and software vendors) is an open problem. Although we did not discuss capabilities at length, they are well-matched to object-oriented concepts and may prove useful in developing systems-of-systems that integrate multiple object models – related to multiple physical entities – in large-scale deployments. Unfortunately, the history of capabilities indicates that research in them is a dead end short of fundamental breakthroughs.

Although buffer overflows are still prevalent in the wild, network-oriented attacks like cross-site scripting and database injection are the new memory-based attacks of choice. Hardware can efficiently defend against buffer overflows, so naturally we should consider hardware approaches to defend against these other memory-based attacks. Information flow tracking seems to be a good approach, but solving its inherent problems, namely the unbounded propagation of taint, drives ongoing research. Fine-grained permission checking also may help, but managing fine-grained permissions well is also an open problem.

Room for improvement exists in better securing and integrating cryptographic accelerators, secure coprocessors, and EED platforms. Especially important is considering how multicore multiprocessing affects hardware-based security primitives. One of the exercises asks you to consider some of the ramifications of multiprocessing with EED cores. Performance of the tamper-proof components is low compared with what unsecured hardware can achieve; mostly to blame is that tamper-proof enclosures dissipate heat slowly, and to prevent burning, the enclosed processor must execute at lower frequencies than if unenclosed.

Low-cost, highly secure hardware remains a lofty yet important goal that may involve multiple fields of science, engineering, and mathematics. Importantly, we should think about how software and hardware can together play a role in securing the systems of tomorrow.

EXERCISES

1. Explain the relationship between the terms trusted, untrusted, trustworthy, and untrustworthy in the context of secure computing. Give a hardware example for each term.
2. An attacker determines how to get a Trojan circuit past every detection technique and into a processor such that it will accept arbitrary commands remotely from the attacker. Suppose an armed military UAV has been compromised by this Trojan circuit in its main processor. As an attacker, what is a good strategy to use to gain an advantage from the Trojan circuit: would you wait for a critical moment to strike or perhaps try to slowly sabotage the UAV? As a defender, how might the military prevent the compromise of the processor from compromising the UAV? How might your defensive solution affect the performance of the UAV? Does the UAV still provide services in spite of an active cyber-attack?
3. Explain how a hardware Trojan circuit compares with software malware. What are the advantages and disadvantages that each has from the perspective of an attacker? What are techniques for protecting systems against each?
4. Some manufacturers ship integrated circuits with reconfigurable logic attached directly to a processor core. Explain how the reconfigurable logic might be used to improve the performance and effectiveness of system security.
5. Take an example for each of the ARM, Intel, PowerPC, and SPARC architectures and identify how many privilege rings (levels) it supports and how it encodes a privileged processor state in a control structure. In practice, only two privilege rings really matter: high and low. State some reasons why the rest are ignored.
6. Suppose you are developing an application for a limited resource embedded system. The memory protection mechanism offered by the architecture is only capable of protecting four distinct statically defined memory ranges. How would you partition your application code and data among the four memory ranges? What permissions would you set to each range?
7. In critical systems where physical access cannot be easily restricted, some form of encrypted execution can protect data and code on buses and memory. Consider such a system that has multiple processors, each with its own cache, and shared main memory. Identify some of the issues that can occur in such a system. Discuss how sharing some (but not all) data among some (but not all) of the processors complicates system design.
8. Smartcards and TPMs have a similar internal architecture, yet they are used in different application domains. Discuss the differences in how the two interface with another system and describe two specific applications, one that motivates the use of smartcards and one that motivates TPMs.

REFERENCES

[1] DARPA. BAA 06-40 – solicitations – microsystems technology office. http://www.darpa.mil/mto/solicitations/baa06-40/, 2006.
[2] DARPA. BAA 07-24 – solicitations – microsystems technology office. http://www.darpa.mil/MTO/solicitations/baa07-24/index.html, 2007.

[3] S. Adee, The hunt for the kill switch, Spectr. IEEE 45 (5) May (2008) 34–39.
[4] Semiconductor Equipment and Materials Industry (SEMI). Innovation at risk: Intellectual property challenges and opportunities. http://www.semi.org/en/Issues/IntellectualProperty/index.htm, 2008.
[5] S. Trimberger, Trusted design in FPGAs, in: Design Automation Conference, 2007. DAC '07. 44th ACM/IEEE, ACM, New York, NY, USA, 2007, pp. 5–8.
[6] M. Tehranipoor, F. Koushanfar, A survey of hardware trojan taxonomy and detection, IEEE Des. Test Comput. 27 (1) (2010) 10–25.
[7] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, B. Sunar, Trojan detection using IC fingerprinting, IEEE Symposium on Security and Privacy, IEEE, Berkeley, CA, 2007, pp. 296–310.
[8] R. Rad, J. Plusquellic, M. Tehranipoor, Sensitivity analysis to hardware trojans using power supply transient signals, in: IEEE International Workshop on Hardware-Oriented Security and Trust, 2008, HOST 2008, pp. 3–7.
[9] M. Banga, M. S. Hsiao, A region based approach for the identification of hardware trojans, in: Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, IEEE Computer Society, Anaheim, CA, 2008, pp. 40–47.
[10] S. Narasimhan, D. Du, R.S. Chakraborty, S. Paul, F. Wolff, C. Papachristou, K. Roy, et al., Multiple-parameter side-channel analysis: A non-invasive hardware trojan detection approach, in: Hardware-Oriented Security and Trust (HOST), 2010 IEEE International Symposium on, IEEE, Anaheim, CA, 2010, pp. 13–18.
[11] J. Li, J. Lach, At-speed delay characterization for IC authentication and trojan horse detection, in: IEEE International Workshop on Hardware-Oriented Security and Trust, 2008, HOST 2008, pp. 8–14.
[12] Y. Jin, Y. Makris, Hardware trojan detection using path delay fingerprint, in: Proceedings of the 2008 IEEE International Workshop on Hardware-Oriented Security and Trust, IEEE Computer Society, Anaheim, CA, 2008, pp. 51–57.
[13] M. Potkonjak, A. Nahapetian, M. Nelson, T. Massey, Hardware trojan horse detection using gate-level characterization, in: Proceedings of the 46th Annual Design Automation Conference, ACM, San Francisco, CA, 2009, pp. 688–693.
[14] S. Wei, S. Meguerdichian, M. Potkonjak, Gate-level characterization: foundations and hardware security applications, in: Proceedings of the 47th Design Automation Conference, ACM, Anaheim, CA, 2010, pp. 222–227.
[15] F. Wolff, C. Papachristou, S. Bhunia, R. S. Chakraborty, Towards trojan-free trusted ICs: problem analysis and detection scheme, in: Proceedings of the Conference on Design, Automation and Test in Europe, ACM, Munich, Germany, 2008, pp. 1362–1365.
[16] R. S. Chakraborty, S. Paul, S. Bhunia, On-demand transparency for improving hardware trojan detectability, in: IEEE International Workshop on Hardware-Oriented Security and Trust, IEEE, Anaheim, CA, 2008, HOST 2008, pp. 48–50.
[17] R. S. Chakraborty, F. Wolff, S. Paul, C. Papachristou, S. Bhunia, MERO: a statistical approach for hardware trojan detection, in: Proceedings of the 11th International Workshop on Cryptographic Hardware and Embedded Systems, Springer-Verlag, Lausanne, Switzerland, 2009, pp. 396–410.
[18] S. Jha, S. K. Jha, Randomization based probabilistic approach to detect trojan circuits, in: Proceedings of the 2008 11th IEEE High Assurance Systems Engineering Symposium, IEEE Computer Society, Nanjing, China, 2008, pp. 117–124.
[19] M. Banga, M. S. Hsiao, A novel sustained vector technique for the detection of hardware trojans, in: Proceedings of the 22nd International Conference on VLSI Design. IEEE Computer Society, Los Alamitos, CA, 2009, pp. 327–332.
[20] H. Salmani, M. Tehranipoor, J. Plusquellic, New design strategy for improving hardware trojan detection and reducing trojan activation time, in: Proceedings of the 2009 IEEE International Workshop on Hardware-Oriented Security and Trust, IEEE Computer Society, Los Alamitos, CA, 2009, pp. 66–73.
[21] M. Banga, M. S. Hsiao, VITAMIN: voltage inversion technique to ascertain malicious insertions in ICs, in: IEEE International Workshop on Hardware-Oriented Security and Trust, IEEE Computer Society, Los Alamitos, CA, 2009, pp. 104–107.
[22] E. Charbon, I. Torunoglu, Watermarking techniques for electronic circuit design, in: Digital Watermarking, 2003, pp. 347–374.
[23] J. Lach, W. H. Mangione-Smith, M. Potkonjak, FPGA fingerprinting techniques for protecting intellectual property, in: Proceedings of the IEEE Custom Integrated Circuits Conference, IEEE, Santa Clara, CA, 1998, pp. 299–302.
[24] J. Lach, W. H. Mangione-Smith, M. Potkonjak, Robust FPGA intellectual property protection through multiple small watermarks, in: Proceedings of the 36th Annual ACM/IEEE Design Automation Conference, ACM,

New Orleans, Louisiana, United States, 1999, pp. 831–836.

[25] L. Yuan, P. Pari, G. Qu, Soft IP protection: Watermarking HDL codes, in: Information Hiding, 2005, pp. 224–238.

[26] M. Lin, G.-R. Tsai, C.-R. Wu, C.-H. Lin, Watermarking technique for HDL-based IP module protection, in: Third International Conference on Intelligent Information Hiding and Multimedia Signal Processing, 2007. IIHMSP 2007. vol. 2, IEEE Computer Society, Los Alamitos, CA, USA, 2007, pp. 393–396.

[27] E. Castillo, U. Meyer-Baese, A. Garcia, L. Parrilla, A. Lloris, IPP@HDL: efficient intellectual property protection scheme for IP cores, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 15 (5) (2007) 578–591.

[28] A. E. Caldwell, H.-J. Choi, A. B. Kahng, S. Mantik, M. Potkonjak, G. Qu, et al., Effective iterative techniques for fingerprinting design IP, IEEE Trans. Comput. Aided Des. Integr. Circuits. Syst. 23 (2) (2004) 208–215.

[29] K. Lofstrom, W. R. Daasch, D. Taylor, IC identification circuit using device mismatch, IEEE International Solid-State Circuits Conference, 2000. Digest of Technical Papers. ISSCC 2000, pp. 372–373.

[30] S. Maeda, H. Kuriyama, T. Ipposhi, S. Maegawa, M. Inuishi, An artificial fingerprint device (AFD) module using poly-Si thin film transistors with logic LSI compatible process for built-in security, Electron Devices Meeting, 2001. IEDM Technical Digest. International, 2001, pp. 34.5.1–34.5.4.

[31] B. Gassend, D. Clarke, M. van Dijk, S. Devadas, Silicon physical random functions, in: Proceedings of the 9th ACM Conference on Computer and Communications Security, ACM, Washington, DC, 2002, pp. 148–160.

[32] B. Gassend, D. Lim, D. Clarke, M. van Dijk, S. Devadas, Identification and authentication of integrated circuits, Concurrency. Computat. Pract. Exper. 16 (11) (2004) 1077–1098.

[33] G. E. Suh, S. Devadas, Physical unclonable functions for device authentication and secret key generation, in: Proceedings of the 44th Annual Design Automation Conference, DAC '07, ACM, New York, NY, 2007, 9–14. ACM ID: 1278484.

[34] E. Simpson, P. Schaumont, Offline Hardware/Software authentication for reconfigurable platforms, in: Cryptographic Hardware and Embedded Systems – CHES 2006, 2006, pp. 311–323.

[35] F. Koushanfar, G. Qu, Hardware metering, in: Proceedings Design Automation Conference, 2001, pp. 490–493.

[36] J. W Lee, D. Lim, B. Gassend, G. E. Suh, M. van Dijk, S. Devadas, A technique to build a secret key in integrated circuits with identification and authentication applications, in: Proceedings of the IEEE VLSI Circuits Symposium, IEEE, 2004, pp. 176—179.

[37] J. A. Roy, F. Koushanfar, I. L. Markov, EPIC: ending piracy of integrated circuits, in: Design, Automation and Test in Europe, 2008. DATE '08, 2008, pp. 1069–1074.

[38] N. Couture, K. B. Kent, Periodic licensing of FPGA based intellectual property, in: IEEE International Conference on Field Programmable Technology, IEEE, 2006, FPT 2006, pp. 357–360.

[39] A. Baumgarten, A. Tyagi, J. Zambreno, Preventing IC piracy using reconfigurable logic barriers, IEEE Des. Test. Comput. 27 (1) (2010) 66–75.

[40] J. Di, S. Smith, A hardware threat modeling concept for trustable integrated circuits, in: Region 5 Technical Conference, 2007 IEEE, 2007, pp. 354—357.

[41] S. C. Smith, J. Di, Detecting malicious logic through structural checking, in: IEEE Region 5 Technical Conference, IEEE, 2007, pp. 217–222.

[42] G. Bloom, B. Narahari, R. Simha, Fab forensics: Increasing trust in IC fabrication, in: IEEE International Conference on Technologies for Homeland Security (HST), IEEE, Waltham, MA, November 2010.

[43] X. Wang, M. Tehranipoor, J. Plusquellic, Detecting malicious inclusions in secure hardware: Challenges and solutions, in: IEEE International Workshop on Hardware-Oriented Security and Trust, IEEE, HOST 2008, pp. 15–19.

[44] S. T. King, J. Tucek, A. Cozzie, C. Grier, W. Jiang, Y. Zhou, Designing and implementing malicious hardware, in: Proceedings of the 1st Usenix Workshop on Large-Scale Exploits and Emergent Threats, USENIX Association, San Francisco, CA, 2008, pp. 1–8.

[45] M. Hicks, M. Finnicum, S. T. King, M. M. K. Martin, J. M. Smith, Overcoming an untrusted computing base: Detecting and removing malicious hardware automatically, in: IEEE International Conference on Security and Privacy (SP), IEEE, 2010, pp. 159–172.

[46] G. Bloom, B. Narahari, R. Simha, J. Zambreno, Providing secure execution environments with a last line of defense against trojan circuit attacks, Comput. Secur. 28 (7) (2009) 660–669.

[47] T. Huffmire, B. Brotherton, G. Wang, T. Sherwood, R. Kastner, T. Levin, et al., Moats and drawbridges: An isolation primitive for reconfigurable hardware based systems, in: IEEE

Symposium on Security and Privacy, 2007. SP ’07, IEEE, 2007, pp. 281–295.
[48] G. Bloom, B. Narahari, R. Simha, OS support for detecting trojan circuit attacks, in: IEEE International Workshop on Hardware-Oriented Security and Trust, IEEE Computer Society, Los Alamitos, CA, 2009, pp. 100–103.
[49] A. Seshadri, A. Perrig, L. van Doorn, P. Khosla, SWATT: SoftWare-based ATTestation for embedded devices. in: Proceedings of the IEEE Symposium on Security and Privacy, 2004.
[50] D. Y. Deng, A. H. Chan, G. E. Suh, Hardware authentication leveraging performance limits in detailed simulations and emulations, in: Proceedings of the 46th Annual Design Automation Conference, ACM, San Francisco, CA, 2009, pp. 682–687.
[51] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, et al., Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks, in: USENIX Security Symposium, USENIX Society, 1998.
[52] T. Chiueh, F. Hsu, Rad: A compile-time solution to buffer overflow attacks, in: 21st IEEE International Conference on Distributed Computing Systems (ICDCS’01), IEEE, 2001.
[53] M. L. Corliss, E. C. Lewis, A. Roth, Using DISE to protect return addresses from attack, in: Workshop on Architectural Support for Security and Anti-Virus, 2004.
[54] D. Samyde, S. Skorobogatov, R. Anderson, J.-J. Quisquater, On a new way to read data from memory, in: Security in Storage Workshop, 2002. Proceedings of the First International IEEE, Greenbelt, Maryland, December 2002, pp. 65–69.
[55] H. M. Deitel, M. S. Kogan, The Design of OS/2, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1992.
[56] C. Reis, A. Barth, C. Pizano, Browser security: lessons from google chrome, Commun. ACM 52 (2009) 45–49.
[57] W. Shi, J. B. Fryman, G. Gu, H. H. S. Lee, Y. Zhang, J. Yang, Infoshield: a security architecture for protecting information usage in memory, in: The Twelfth International Symposium on High-Performance Computer Architecture, 2006, pp. 222–231.
[58] D. Arora, A. Raghunathan, S. Ravi, N. K. Jha, Architectural support for run-time validation of program data properties, IEEE Trans. Very Large Scale Integration (VLSI) Systems 15 (5) (2007) (546–559).
[59] E. Witchel, J. Cates, K. Asanovic, Mondrian memory protection, in: Proceedings of ASPLOS-X, ACM, New York, NY, USA, October 2002.
[60] W. Shi, C. Lu, H.-H. S. Lee, Memory-centric security architecture. High performance embedded architectures and compilers, Barcelona, Spain, November 17–18, 2005.
[61] E. Leontie, G. Bloom, B. Narahari, R. Simha, J. Zambreno, Hardware containers for software components: A trusted platform for COTS-based systems, in: Proceedings of The 2009 IEEE International Symposium on Trusted Computing, TrustCom09, Vancouver, Canada, August 2009.
[62] E. Leontie, G. Bloom, B. Narahari, R. Simha, J. Zambreno, Hardware-enforced fine-grained isolation of untrusted code, in: Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code, ACM, Chicago, IL, 2009, pp. 11–18.
[63] T. Huffmire, S. Prasad, T. Sherwood, R. Kastner, Policy-driven memory protection for reconfigurable hardware. Lecture Notes in Computer Science, 2006, pp. 461–478.
[64] H. M. Levy, Capability-Based Computer Systems, Butterworth-Heinemann, Newton, MA, USA, 1984.
[65] R. B. Lee, D. K. Karig, J. P. Mcgregor, Z. Shi, Enlisting hardware architecture to thwart malicious code injection, in: Proceedings of the 2003 International Conference on Security in Pervasive Computing, Springer Verlag, Boppard, Germany, 2003, pp. 237–252.
[66] H. Ozdoganoglu, T. N. Vijaykumar, C. E. Brodley, B. A. Kuperman, A. Jalote, SmashGuard: A hardware solution to prevent attacks on the function return address. IEEE Transactions on Computers, 2006.
[67] J. Xu, Z. Kalbarczyk, S. Patel, R. K. Iyer, Architecture support for defending against buffer overflow attacks, in: Proceedings of the 2nd Workshop Evaluating and Architecting System Dependability (EASY-2002), 2002.
[68] K. Inoue, Energy-security tradeoff in a secure cache architecture against buffer overflow attacks, SIGARCH Comput. Archit. News 33 (1) (2005) 81–89.
[69] W. Kao, S. F. Wu, Lightweight hardware return address and stack frame tracking to prevent function return address attack, in: CSE ’09: Proceedings of the 2009 International Conference on Computational Science and Engineering, IEEE Computer Society, Washington, DC, 2009, pages 859–866.
[70] Z. Shao, J. Cao, K. C. C. Chan, C. Xue, E. H.-M. Sha, Hardware/software optimization for array & pointer boundary checking against buffer overflow attacks, J. Parall. Distrib. Comput. 66 (9) (2006) 1129–1136.

[71] J. R. Crandall, S. F. Wu, F. T. Chong, Minos: Architectural support for protecting control data, ACM Trans. Archit. Code Optim. 3 (4) (2006) 359–389.
[72] A. Smirnov, T. Chiueh, Dira: Automatic detection, identification, and repair of control-hijacking attacks, in: NDSS, 2005.
[73] G. E. Suh, J. W. Lee, D. Zhang, S. Devadas, Secure program execution via dynamic information flow tracking, in: ASPLOS-XI: Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, 2004, pp. 85–96.
[74] J. Zambreno, A. Choudhary, R. Simha, B. Narahari, N. Memon. Safe-ops: An approach to embedded software security, ACM Trans. Embed. Comput. Syst. 4 (1) (2005) 189–210.
[75] M. Abadi, M. Budiu, Ú. Erlingsson, J. Ligatti, Control-flow integrity, in: Proceedings of the 12th ACM Conference on Computer and Communications Security, ACM, November 2005, pp. 340–353.
[76] E. Leontie, G. Bloom, O. Gelbart, B. Narahari, R. Simha, A compiler-hardware technique for protecting against buffer overflow attacks, J. Inf. Assur. Secur. 5 (1) (2010).
[77] M. Kharbutli, X. Jiang, Y. Solihin, G. Venkataramani, M. Prvulovic, Comprehensively and efficiently protecting the heap, ACM SIGOPS Operating Systems Review, Proceedings of the 2006 ASPLOS Conference, December 2006, pp. 207–218.
[78] D. Arora, A. Raghunathan, S. Ravi, N. K. Jha, Architectural support for safe software execution on embedded processors, in: Proceedings of the fourth International Conference on Hardware/Software Codesign and System Synthesis, ACM, New York, NY, USA, 2006, pp. 106–111.
[79] G. Venkataramani, B. Roemer, Y. Solihin, M. Prvulovic, Memtracker: Efficient and programmable support for memory access monitoring and debugging. High-Performance Computer Architecture, International Symposium on, 0:273–284, 2007.
[80] G.E. Suh, J. Lee, S. Devadas, Secure program execution via dynamic information flow tracking. Architectural support for programming languages and operating systems, 2004, pp. 85–96.
[81] M. Dalton, H. Kannan, C. Kozyrakis, Raksha: A flexible information flow architecture for software security, in: International Symposium on Computer Architecture, ACM, New York, NY, USA, June 2007, pp. 482–493.
[82] F. Qin, C. Wang, Z. Li, H. S. Kim, Y. Zhou, Y. Wu, Lift: A low-overhead practical information flow tracking system for detecting security attacks, in: 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06), IEEE, 2006, pp. 135–148.
[83] M. Tiwari, H. Wassel, B. Mazloom, S. Mysore, F. Chong, T. Sherwood, Complete information flow tracking from the gates up, in: Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, New York, NY, USA, March 2009.
[84] G. Venkataramani, I. Doudalis, Y. Solihin, M. Prvulovic, Flexitaint: A programmable accelerator for dynamic taint propagation, in: International Symposium on High Performance Computer Architecture, IEEE, 2008, pp. 173–184.
[85] T. Grembowski, R. Lien, K. Gaj, N. Nguyen, P. Bellows, J. Flidr, et al., Comparative analysis of the hardware implementations of hash functions SHA-1 and SHA-512, in: Proceedings of the International Conference on Information Security (ISC), Springer-Verlag, London, UK, 2002, pp. 75–89.
[86] K. U. Jarvinen, M. T. Tommiska, J. O. Skytt, A fully pipelined memoryless 17.8 Gbps AES-128 encryptor, in: Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA), 2003, pp. 207–215.
[87] J. Kaps, C. Paar, Fast DES implementation for FPGAs and its application to a universal key-search machine, in: Proceedings of the Annual Workshop on Selected Areas in Cryptography (SAC), Springer-Verlag, London, UK, 1998, pp. 234–247.
[88] O. Kocabas, E. Savas, J. Grosschadl, Enhancing an embedded processor core with a cryptographic unit for speed and security, in: International conference on Reconfigurable Computing and FPGAs, 2008, ReConFig '08. December 2008, IEEE, pp. 409–414.
[89] J. Burke, J. McDonald, T. Austin, Architectural support for fast symmetric-key cryptography, SIGOPS Oper. Syst. Rev. 34 (2000) 178–189.
[90] J. Zambreno, D. Nguyen, A. Choudhary, Exploring area/delay tradeoffs in an AES FPGA implementation, in: Proceedings of the International Conference on Field Programmable Logic and Applications (FPL), Springer, 2004, pp. 575–585.
[91] S. Okada, N. Torii, K. Itoh, M. Takenaka, Implementation of elliptic curve cryptographic coprocessor over GF(2^m) on an FPGA, in: Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag, London, UK, 2000, pp. 25–40.

[92] M. Ernst, M. Jung, F. Madlener, S. Huss, R. Blumel, A reconfigurable system on chip implementation for elliptic curve cryptography over GF(2^n), in: Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag, London, UK, 2002, pp. 381–399.

[93] A. Hodjat, I. Verbauwhede, A 21.54 Gbits/s fully pipelined AES processor on FPGA, 12th Field-Programmable Custom Computing Machines (FCCM), 2004.

[94] A. Dandalis, V. K. Prasanna, J. D. P. Rolim, An adaptive cryptographic engine for ipsec architectures, in: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2000, pp. 132–144.

[95] B. Yee, J. D. Tygar, Secure coprocessors in electronic commerce applications, in: Proceedings of the USENIX Workshop on Electronic Commerce, USENIX Association, Berkeley, CA, USA, July 1995, pp. 155–170.

[96] S. H. Weingart, Physical security for the μabyss system, IEEE Symposium on Security and Privacy, vol. 52, 1987.

[97] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: Proceedings of Advances in Cryptology (Crypto), Springer-Verlag, London, UK, 1999, pp. 388–397.

[98] S. R. White, Abyss: A trusted architecture for software protection, IEEE Symposium on Security and Privacy, vol. 38, 1987.

[99] J. D. Tygar, B. Yee, Dyad: A system for using physically secure coprocessors, in: Proceedings of the Harvard-MIT Workshop on Protection of Intellectual Property, April 1993.

[100] E. Palmer. An introduction to Citadel – a secure crypto coprocessor for workstations. Research Report RC 18373, IBM T. J. Watson Research Center, 1992.

[101] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S.W. Smith, Building the IBM 4758 secure coprocessor, Computer, 34 (10) (2001) 57–66.

[102] S. W. Smith, Secure coprocessing applications and research issues, in: Los Alamos Unclassified Release LA-UR-96-2805, Los Alamos National Laboratory, Los Alamos, NM, 1996.

[103] X. Zhuang, T. Zhang, H.-H. S. Lee, S. Pande, Hardware assisted control flow obfuscation for embedded processors. in: Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES), ACM, New York, NY, USA, September 2004.

[104] X. Zhuang, T. Zhang, S. Pande. HIDE: An infrastructure for efficiently protecting information leakage on the address bus. Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2004.

[105] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, Dan Boneh, John Mitchell, et al., Architectural Support for Copy and Tamper Resistant Software. Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, New York, NY, USA, 2000.

[106] M. Milenkovic, A. Milenkovic, E. Jovanov, Hardware support for code integrity in embedded processors. CASES, 2005.

[107] D. Kirovski, M. Drinić, M. Potkonjak, Enabling trusted software integrity, ASPLOS, 30 (5) October (2002).

[108] O. Gelbart, P. Ott, B. Narahari, R. Simha, A. Choudhary, J. Zambreno, CODESEAL: Compiler/FPGA approach to secure applications, in: Proc. IEEE, ISI, IEEE, 2005.

[109] O. Gelbart, E. Leontie, B. Narahari, R. Simha, Architectural support for securing application data in embedded systems, in: IEEE International Conference on Electro/Information Technology, IEEE, May 2008, pp. 19–24.

[110] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, S. Devadas, AEGIS: Architecture for tamper-evident and tamper-resistant processing, in: P17 International Conference on Supercomputing (ICS), ACM, New York, NY, USA, 2003.

[111] F.-X. Standaert, S. B. Ors, J.-J. Quisquater, B. Preneel, Power analysis attacks against FPGA implementations of the DES, in: Proceedings of the International Conference on Field-Programmable Logic and its Applications (FPL), 2004, pp. 84–94.

[112] S. B. Ors, F. Gurkaynak, E. Oswald, B. Preneel, Power-analysis attack on an ASIC AES implementation, in: Proceedings of the International Conference on Information Technology (ITCC), IEEE Computer Society, Washington, DC, USA, 2004.

[113] M.-L. Akkar, C. Giraud, An implementation of DES and AES secure against some attacks, in: Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems (CHES), Springer-Verlag, London, UK, May 2001, pp. 309–318.

[114] K. Tiri, I. Verbauwhede, A digital design flow for secure integrated circuits, IEEE Trans. Comput.-Aided Design of Integrated Circuits and Systems, July 2006, pp. 1197–1208.

[115] P. Yu, P. Schaumont, Secure FPGA circuits using controlled placement and routing. International Conference on Hardware/Software Codesign and System Synthesis, October 2007.