CHAPTER

# SYSTEM LEVEL ATTACKS & COUNTERMEASURES

16

## CONTENTS

## 16.1 INTRODUCTION

In modern computing systems, the hardware and software stack coordinate with each other to implement the system functionality. Whereas the previous chapters have focused on security issues of the hardware itself, they have not covered another important aspect of hardware security that concerns with providing an infrastructure for secure software execution. In particular, the role of hardware in protecting the assets stored in a chip or PCB (as defined earlier in Chapter 1) from malicious software have not been described in detail. Similarly, protection of the data/code of one application from another potentially malicious one, have not been addressed. The hardware needs to support security against software attacks, considering all levels of the software stack, from the operating system to application software. These attacks can be mounted through either functional or side-channel vulnerabilities. In this chapter, we will discuss various scenarios of software-induced attacks on hardware and possible countermeasures.

The software stack in a system runs on a central processing unit (CPU), which is most commonly implemented, nowadays, as an System on Chip (SoC) that integrates a processor IP. These systems increasingly integrate, along with a CPU, FPGAs and GPUs, which act as an accelerator for specific applications. This phenomenon has introduced increasingly complex hardware-hardware and hardware-software interactions, ultimately raising the question, “How do we secure our systems?” In this chapter, we first look into the security issues present in an SoC. Next, we focus on some requirements for designing a secure SoC. But before we move into system-level security issues, we need to understand the architecture of modern SoCs, and how hardware-software interaction takes place inside an SoC. We also need to understand the current practices for SoC security. The remainder of the chapter provides relevant background for SoC security, discusses various vulnerabilities and attack scenarios that can be mounted through hardware-software interactions, and presents various solutions.

## 16.2 BACKGROUND ON SoC DESIGN

The principal components of a standard, simplified SoC are shown in Fig. 16.1. It integrates IP blocks developed in the SoC design house, or acquired from various IP vendors, using an interconnection fabric to achieve desired functionality. Major IP blocks that are integrated into an SoC include a processor core (that runs the software stack), memory (that serves as the processor cache), crypto module (for functional security measures), the power management, and the communication module (for example, a USB module). The interconnect fabric (also called “fabric”) can be realized in one of the following three ways, or any combination of them: (1) a point-to-point connection among the IP blocks; (2) bus-based communication architecture that uses a shared bus with appropriate arbitration logic; (3) a network-on-chip (NoC) architecture, where IPs communicate through specially designed “routers”, which are responsible for transferring messages from one point to another. Figure 16.2 illustrates the three major types of communication architecture. In general, performance of the SoC design heavily depends upon the efficiency of its communication architecture.

Usually IP blocks in a SoC are designed with several standardized interfaces and communication protocols to interface with bus-based and NoC architectures. SoC integration process for the IPs with the fabric requires configuring the interfaces of IP blocks, and insertion of glue logic to connect them to the fabric. Standards of on-chip bus architecture were developed to facilitate the SoC integration pro-

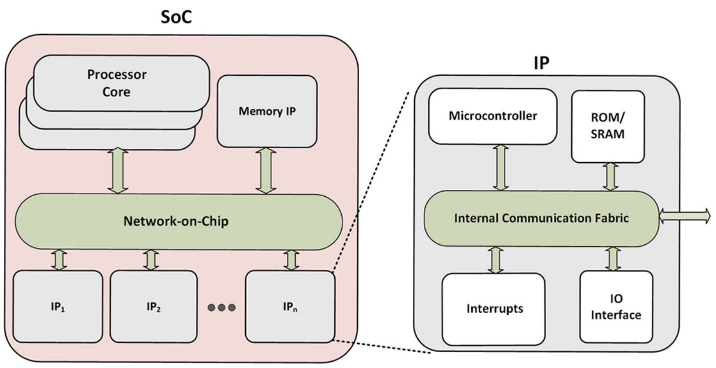

**FIGURE 16.1**

A modern SoC architecture that consists of multiple IP blocks connected by an interconnect fabric.

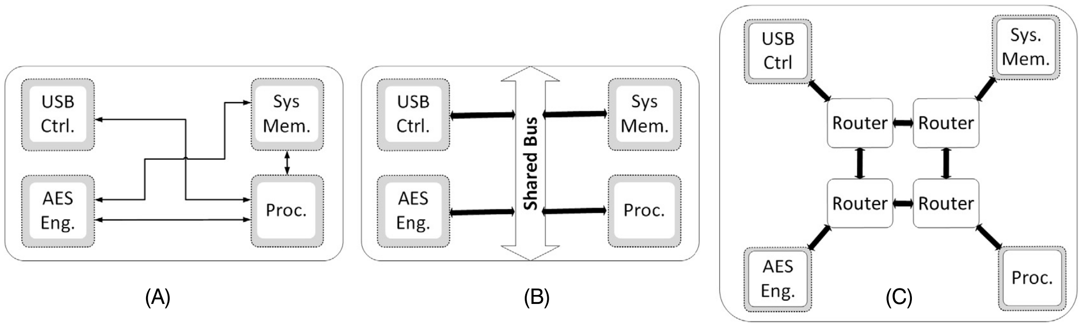

**FIGURE 16.2**

SoC architecture with (A) Point-to-point interconnect, (B) Shared bus interconnect, (C) Network-on-chip interconnect.

cess. These standards are often specific to the processor architecture, and consistent with the ecosystem produced by an IP vendor. For example, CoreConnect bus architecture from IBM [1] and AMBA from ARM [2] are tied to the corresponding processor, for example, PowerPC and ARM processor, respectively, and the IP ecosystem. Point-to-point architecture are not suitable for large complex SoCs due to lack of standardized IP interface, and integration process. They also suffer from scalability issues with increasing number of IPs.

## 16.3 SoC SECURITY REQUIREMENTS

In this section, we give an introduction into the security requirements SoC designers need to consider. These requirements are determined based on potential adversaries and attack vectors at different stages of SoC life cycle.

### 16.3.1 ASSETS IN SoC

SoC assets can be broadly defined as system-critical and security-sensitive information stored in the chips. With computing devices being employed for a large number of highly personalized activities (for example, shopping, banking, fitness tracking, and providing driving directions), these devices have access to a large amount of sensitive, personal information, which must be protected from unauthorized or malicious access. In addition to personalized end-user information, most modern computing systems contain highly confidential collateral from the architecture, design, and manufacturing, such as cryptographic and digital rights management (DRM) keys, programmable fuses, on-chip debug instrumentation, and defeature bits. It is crucial to our well-being that data in these devices are protected from unauthorized access and eventual corruption. Hence, security architecture, that is, mechanism to ensure protection of sensitive assets from malicious, unauthorized access, constitutes a crucial component of modern SoC designs.

### 16.3.2 ADVERSARIAL MODEL

To ensure that an asset is protected, the designer needs comprehension of the power of the adversary. Effectiveness of virtually all security mechanisms is critically dependent on how realistic the model of the adversary is. Conversely, most security attacks rely on breaking some of the assumptions made regarding constraints on the adversary. The notion of adversary can vary, depending on the asset being considered. For example, in case of protecting DRM keys, the end-user would be an adversary, whereas the content provider (and even the system manufacturer) may be included among adversaries in the context of protecting private information of the end-user. Rather than focusing on a specific class of users as adversaries, it is more convenient to model adversaries corresponding to each asset, and define protection and mitigation strategies with respect to that model. Defining and classifying the potential adversary is a creative process. It needs various considerations, such as whether the adversary has physical access, and which components they can observe, control, modify, or reverse engineer.

### 16.3.3 DESIGN-FOR-DEBUG IN SoC

As mentioned in Chapter 1, security requirements for SoC often represent a conflict with Design-for-test (DFT) and Design-for-debug (DfD) infrastructure. DfD refers to on-chip hardware for facilitating post-silicon validation of a chip's functional and security properties. A key requirement for post-silicon validation is observability and controllability of internal signals during silicon execution. DfD in modern SoC designs includes facilities to trace critical hardware signals, dump contents of registers and memory arrays, patch microcode and firmware, and to create user-defined triggers and interrupts. To reduce the risk of an adversary from snooping on data flowing through debug infrastructure (for example, from a crypto IP to a processor IP), data should be protected using standard cryptography primitives. In the case of off-chip key generation for the SoC, the key bits must be protected from the potential snooping from other IPs, especially any untrusted IP. This can be achieved by creating a security-aware test and debug infrastructure, which involves commensurate modification to local test/debug cells of an IP that effectively blocks other IPs from observing key bits [3]. Figure 16.3 shows such sample modifications.

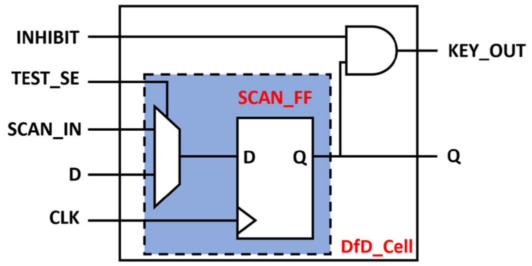

**FIGURE 16.3**

Modified scan cell to allow for secure key transfer by masking the output with the *INHIBIT* signal [3].

## 16.3.4 INTRODUCTION TO SoC SECURITY POLICIES

SoC security is driven by the requirement to protect system assets against unauthorized access. Such access control can be defined by confidentiality, integrity, and availability (CIA) requirements [4]. The goal of a security policy is to map the requirements to "actionable" design constraints that can be used by IP implementers, or SoC integrators, to develop protection mechanisms. Next, we present two examples of SoC security policies.

- *Example 1:* During boot time, data transmitted by the cryptoengine cannot be observed by any IP in the SoC other than its intended target.
- *Example 2:* A programmable fuse containing a secure key can be updated during manufacturing, but not after production.

Example 1 is a confidentiality requirement, whereas Example 2 is an integrity constraint. However, the policies provide concrete conditions to be checked by the design for accessing an asset. Furthermore, access to an asset may vary, depending on the state of execution (for example, boot time or normal execution), or position in the development lifecycle. Following are some representative policy classes. They are not exhaustive, but illustrate the diversity of policies employed.

**Access control:** This is the most common class of policies, and specifies how different agents in an SoC can access an asset at different points of the execution. Here, an "agent" can be a hardware or software component in any IP of the SoC. Examples 1 and 2 above are examples of such policy. Furthermore, access control forms the basis of many other policies, including information flow, integrity, and secure boot.

**Information flow:** Values of secure assets can sometimes be inferred without direct access, through indirect observation or "snooping" of intermediate computation, or communications of IPs. Information flow policies restrict such indirect inference. An example information-flow policy might be the following:

- *Key obliviousness:* A low-security IP cannot infer the cryptographic keys by snooping only the data from crypto engine on a low-security communication fabric.

Information-flow policies are difficult to analyze. They often require highly sophisticated protection mechanisms and advanced mathematical arguments for correctness, typically involving hardness or complexity results from information security. Consequently, they are employed only on critical assets with very high confidentiality requirements.

**Liveness:** These policies ensure that the system performs its functionality without "stagnation" throughout its execution. A typical liveness policy is that a request for a resource by an IP is followed by an eventual response, or grant. Deviation from such a policy can result in system deadlock or livelock, consequently compromising system availability requirements.

**Time-of-check vs. time-of-use (TOCTOU):** This refers to the requirement that any agent accessing a resource requiring authorization is indeed the agent that has been authorized. A critical example of TOCTOU requirement is in firmware update; the policy requires that firmware eventually installed on update is the same firmware that has been authenticated as legitimate by the security, or crypto engine.

**Secure boot:** Booting a system entails communication of significant security assets, for example, efuse configurations, access control priorities, cryptographic keys, firmware updates, and post-silicon observability information. Consequently, boot imposes stringent security requirements on IPs and communications. Individual policies during boot can be access control, information flow, and TOCTOU requirements. However, it is often convenient to coalesce them into a unified set of boot policies.

Most system-level policies are defined at the risk assessment phase by system architects. However, they continue to be refined along different phases of the architecture, and even during early design and implementation activities, as new knowledge and constraints come to light. For example, during architecture definition of a specific product, one may realize that the key obliviousness policy cannot be implemented as stated for that product, since several IPs need to be connected on the same NoC as the cryptographic engine due to resource constraints. This may lead to a refinement in the policy definition by marking some IPs to be "safe" for observing some of the keys. Policies may also need to be refined or updated in response to changing customer or product needs. Such refinements may make it highly challenging to develop a validation methodology, or even a disciplined security architecture. To exacerbate the issue, security policies are rarely specified in any formal, analyzable form. Some policies are described in natural language in different architecture documents, and many (particularly, refinements identified later in the system lifecycle) remain undocumented.

In addition to the system-level policies, there are "lower-level" policies, for example, communication among IPs is specified by fabric policies. Following are some obvious fabric policies:

**Message immutability:** If IP $\mathcal{A}$ sends a message $m$ to IP $\mathcal{B}$ then the message received by $\mathcal{B}$ must be exactly message $m$.

**Redirection and masquerade prevention:** If $\mathcal{A}$ sends a message $m$ to $\mathcal{B}$, then the message must be delivered to $\mathcal{B}$. In particular, it should be impossible for a (potentially rogue) IP $\mathcal{C}$ to masquerade as $\mathcal{B}$, or for the message to be redirected to a different IP $\mathcal{D}$ in addition to, or instead of, $\mathcal{B}$.

**Nonobservability:** A private message from $\mathcal{A}$ to $\mathcal{B}$ must not be accessible to another IP during transit.

The above description does not adequately describe the complexity involved in implementing policies. Consider the SoC configuration shown in Fig. 16.4. Suppose that IP0 needs to send a message to the DRAM. Ordinarily, the message would be routed through Router3, Router0, Router1, and Router2. However, such a route permits message redirection via software. Each router includes a base address register (BAR), which is used to route messages for specific destinations. One of the routers in the proposed path, Router0 is connected to the CPU. The BARs in this router are subject to potential

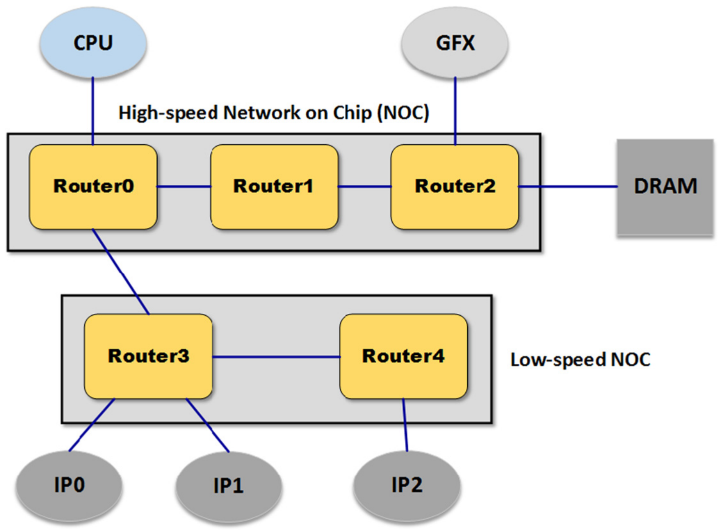

**FIGURE 16.4**

An illustrative simple SoC configuration. SoC designs include several on-chip fabrics with differing speed and power profiles. For this configuration, there is a high-speed fabric with three routers connected linearly, and a low-speed fabric with two routers also connected linearly.

overwrite by the host operating system, which can redirect a message passing through Router0 to a different destination. Consequently, a secure message cannot be sent through this route unless the host operating system is trusted. Note that understanding the potential of redirection requires knowledge of fabric operation, routers design (for example, the use of BARs), and the capabilities of the software in an adversarial role.

In addition to the above generic policies, SoC designs include asset-specific communication constraints. A potential fabric policy relevant to secure boot is listed below. This policy ensures that a key generated by the fuse controller cannot be sniffed during propagation to the crypto engine for storage.

- **Boot-time key nonobservability:** During the boot process, a key from the fuse controller to the crypto engine cannot be transmitted through a router to which any IP with user-level output interface is connected.

## 16.4 SECURITY POLICY ENFORCEMENT

This module is responsible for enforcing security policies that are imperative for ensuring security at the hardware level. The following sections discuss a number of security policies and a “Centralized Policy Definition Architecture,” which is responsible for enforcing security policies.

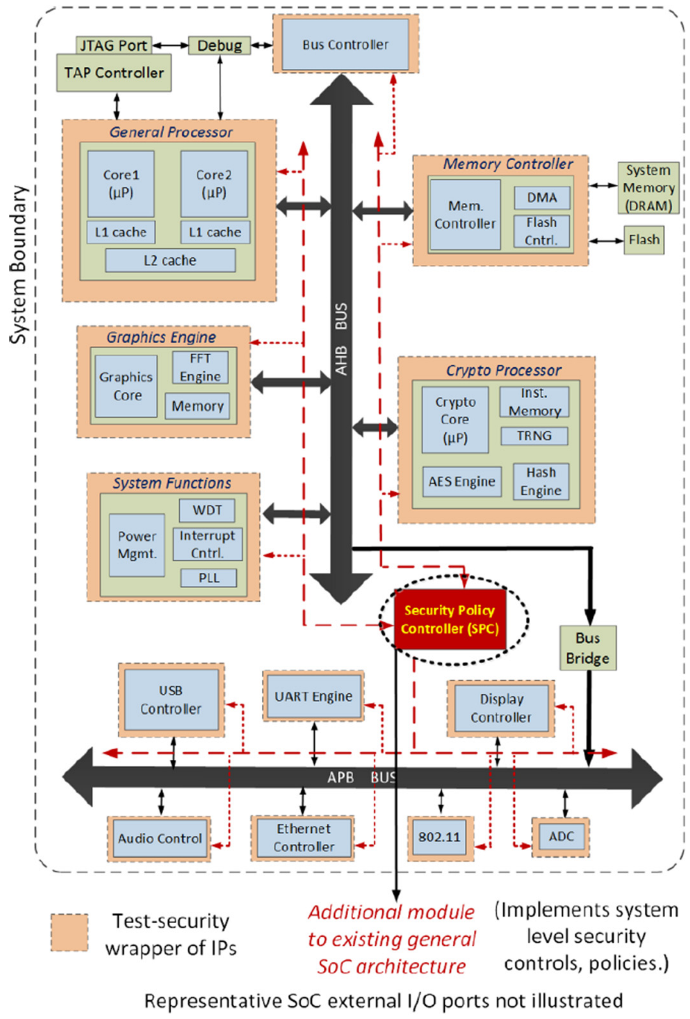

**FIGURE 16.5**

SoC security architecture based on E-IIPS for efficient implementation of diverse security policies.

## 16.4.1 A CENTRALIZED POLICY DEFINITION ARCHITECTURE

Current industrial practice in implementing security policies follow a distributed ad-hoc implementation. Such an approach, however, typically comes at high design and verification cost. Recent work [5, 30], has attempted to develop a centralized, flexible architecture called E-IIPS for implementing security policies in a disciplined manner. The idea is to provide an easy-to-integrate, scalable infrastructure IP that serves as a centralized resource for SoC designs to protect against diverse security threats, at minimal design effort and hardware overhead. Figure 16.5 shows the overall architecture of E-IIPS. It includes a microcontroller-based firmware upgradable module called security policy controller (SPC) that realizes system-level security policies of various forms and types using firmware code, following existing security policy languages. The SPC module interfaces with the constituent IP blocks in a SoC using "security wrappers" integrated with the IPs. These security wrappers extends the existing test (for instance, IEEE 1500 boundary scan based wrapper [32]) and debug wrapper (for example, ARM's CoreSight interface [31]) of an IP. These security wrappers detect local events relevant to the implemented policies and enable communication with the centralized SPC module. The result is a flexible architecture and approach for implementing highly complex system-level security policies, including those involving interoperability requirements, and trade-offs with debug, validation, and power management. The architecture is realizable with modest area and power overhead [5]. Furthermore, more recent work has shown that the existing design instrumentations, such as for DfD, could be exploited in implementing the architecture [30]. Of course, the architecture itself is only one component of the policy definition. Several challenges remain, including: 1) defining a language for security policy specification that can be efficiently compiled to SPC microcode; 2) study of bottlenecks related to routing and congestion across communication fabrics in implementing the architecture; and 3) implementing security policies involving potentially malicious IPs (including malicious security wrappers or Trojans in the SPC itself). Nevertheless, the approach shows a promising direction toward systematizing policy implementations. Furthermore, by enclosing the policy definitions to a centralized IP, it enables security validation to focus on a narrow component of the design, thereby potentially reducing validation time.

## 16.5 SECURE SoC DESIGN PROCESS

Modern-day SoCs are an efficient amalgamation of internal (in-house) and external (third-party) IPs to incorporate numerous functionalities in one single chip. Contemporary SoC design processes involve a systematic progression through several major phases of development lifecycle. Building a secure SoC, however, requires security considerations, and iterative evaluation from very early stages of the product development. Figure 16.6 illustrates how a secure development lifecycle of an SoC can be designed by incorporating security evaluation at every major phase of the design flow. The entire security analysis process of modern-day SoCs can be broadly classified into three crucial phases, that is, early security validation, pre-silicon security validation, and post-silicon security validation. A brief description of each of the phases with associated security analyses is provided below:

### 16.5.1 EARLY SECURITY VALIDATION

Early security validation includes a set of additional steps integrated with the conventional SoC design flow to ensure secure SoC design from the very beginning. Such validation is performed during the architecture and design stages of the development lifecycle. The first task of early security validation is to review the specification of the SoC, and conduct a security analysis. This process involves identifying the security assets in the system, their ownership, and protection requirements, collectively defined as security policies. The result of this process is typically the generation of a set of documents, often referred to as, product security specification (PSS), which provides the requirements for downstream architecture, design, and validation activities. At this phase, the SoC designers incorporate microarchi-

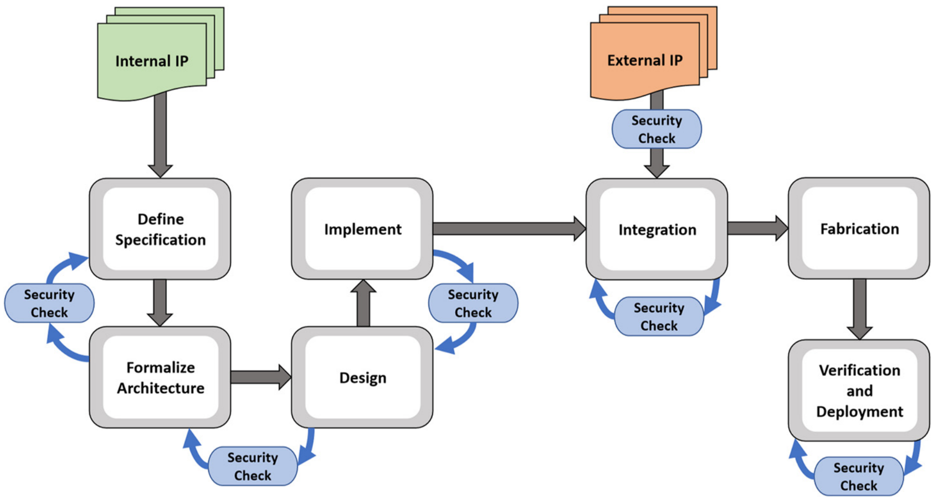

**FIGURE 16.6**

Secure development lifecycle of System on Chips.

tectural changes that facilitates design-for-security (DFS) and validation. The primary purpose of DFS and validation is to make the SoC immune to vulnerabilities, starting from the rudimentary levels of the microarchitecture to bottom up. The second task is threat modeling and risk mitigation. Development of the threat model includes breaking down of discrete security requirements, and analyzing the risk mitigation techniques. The third task is to review the high level design. This task involves generation of test cases to validate the implementation. The practice of employing good design methods and avoiding potential pitfalls are also vital parts of the SoC security validation process at early stage. A well-defined set of design security rules that can be verified by the autochecking tools facilitates such validation.

## 16.5.2 PRE-SILICON SECURITY VALIDATION

The pre-silicon security validation is performed at the implementation phase of the SoC development cycle. During this stage, the SoC architects perform static analysis of the design. Static analysis includes manually reviewing the RTL code. However, the process of reviewing the RTL is a one-time task, as the code changes over the design cycle. The designers also employ design automation tools for conducting the static analysis. In addition to static analysis, the SoC designers create targeted test cases, and run simulation to validate the desired output. The inherent problem with targeted testbench is that the verifiability has a very limited scope, and the process is hardly scalable beyond individual IPs. Apart from the simulation-based testing, the designers also employ formal verification tools to get exhaustive coverage. The formal verification tools, however, are challenged by the complexity of

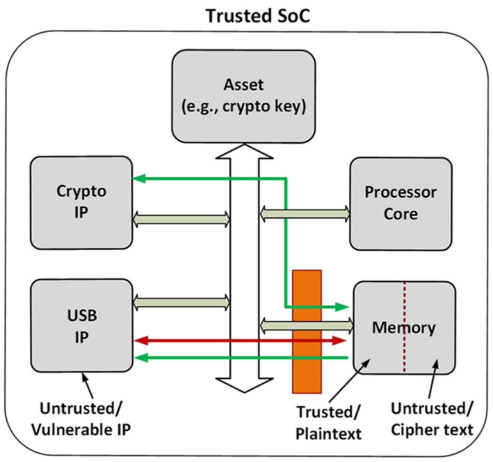

**FIGURE 16.7**

Illustrative SoC model depicting secure/insecure information flow.

modern SoCs, and fail to scale at system-level verification. Prototyping the SoC platform is often done to increase the speed of testing, and validate early software flows.

### 16.5.3 POST-SILICON SECURITY VALIDATION

In the SoC platform, the components or IP modules communicate with each other over the interconnect fabric. Testing and validating the interaction among the platform components is crucial, as illegal access of IPs to secure regions can cause security breaches. Such intercomponent analysis is performed during the post-silicon validation of the SoCs. The SoC designers employ debug and validation tools to probe deeper into the silicon. Verification tools are utilized at this phase to check and analyze system-level flows. The tests are performed by tools that can generate scenarios involving multiple concurrent transactions. Secure information-flow checking is an example of such system-level flow analysis, where the security engineers check for possibility of a security-critical signal being propagated to insecure peripherals, or untrusted IPs. The SoC designers also employ advanced hacking techniques to breach the security, including blackbox and white-box fuzzing. Lastly, a comprehensive software testing is done to finalize the post-silicon validation.

### 16.5.4 CASE SCENARIO: ENSURING A SECURE INFORMATION FLOW

To provide better insights about the criticality and significance of secure information flow, an illustrative example is presented in this section with an illustrative SoC model (shown in Fig. 16.7 [33]). The SoC model is designed with a processor core, a crypto IP, a memory IP as a platform component, a key storage IP for preserving the assets, and a USB IP. The memory addresses are classified into trusted and untrusted regions. The SoC is designed to operate in a manner that IPs with valid crypto ID are given access to the plain text sent to the crypto-engine. The cipher text is stored in the untrusted region

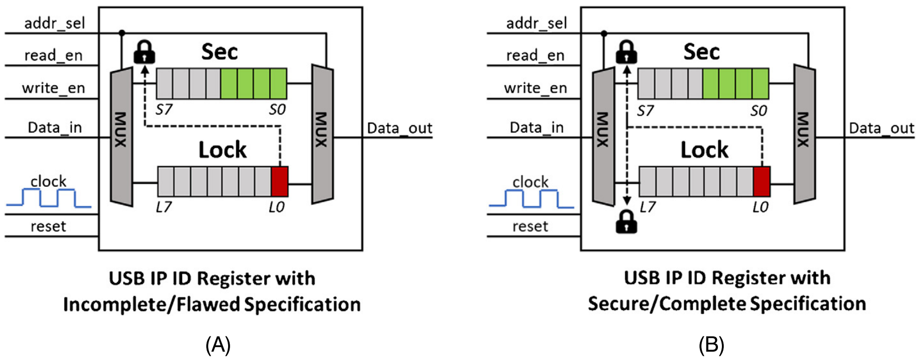

**FIGURE 16.8**

Microarchitectural illustration of the USB IP ID register (A) Sec and Lock registers with incomplete/flawed security specification, (B) Sec and Lock registers with secure/complete security specification.

of memory, and can be accessed by other IPs for operational purposes. The crypto IP is designed to read the plaintext from the trusted part of the memory, and encrypt the text using the keys stored in the Asset IP. Once the encryption is completed, it stores the cipher text to the untrusted memory region that can be accessed by external entry points, such as the USB IP. A firewall is placed on the memory to prevent the untrusted IPs, for instance, the USB IP, to prevent the leakage of the plain text. The firewalls determine the access privilege of the USB IP by registering its ID. Figure 16.8 [33] depicts the microarchitecture of USB IP ID register located at the firewall. The USB ID register is modeled as a security register with lock bit. The registers of the SoC model are basically flip-flops, triggered on the positive edge of the clock. The ID register, *Sec*, only uses the lower 4 (S[3:0]) bits of data-in, and bit 0 of data-in is used for the lock mechanism. Data-out always reads 8 bits from Read, or Lock. As it works as a secure gateway to access the trusted memory regions, it is crucial to perform thorough security analysis of the registers, and fully comprehend the vulnerabilities that might arise from poor design constraints. The ID of the IPs are generated using the 4 lower significant bits of an 8-bit register named *Sec*. To prevent unauthorized writes on the *Sec* register, another register named *Lock* is added to the design. The LSB of *Lock* is used to enable or disable write operations on the *Sec* register.

A threat model for the illustrative SoC model described above would have several crucial aspects that needs to be considered. For instance, the objective of the threat model is prevention of illegal write operations on the *Sec* register, when the lock bit of the *Lock* register is set to 1. This is a security policy that ensures data integrity property on the Sec register. The asset for this particular threat model is the ID and the *Sec* register. An attack scenario for the given threat model would be, any attempt made by untrusted software to modify the *Sec* register once the lock bit is set. Other properties like confidentiality and availability of *Sec* and *Lock* registers are trivial for this case study.

Once a well-defined threat model is structured, the next task of security analysis is the identification of vulnerabilities. For instance, a closer look at the code snippet shown below will reveals that it is possible to circumvent the data integrity of *Sec* register by exploiting poorly written specifications for accessing the *Sec* register. Consequently, the bad design and incomplete specification can aid the attacker to modify the *Lock* bit to disable the locking mechanism. Hence, the untrusted software becomes capable of modifying the *Sec* register (shown in Fig. 16.8).

*RTL Code with incomplete specification:*

```
if
  Addr_sel == 0 AND Lock==1
          Write_En_in==0
Else
          Write_En_in == Write_En
```

On the other hand, a complete set of specification for the design under consideration would protect the write operation on the *Sec* register by the lock mechanism. Also, it should be specified that the *Lock* register is self-locking. So, the corrected implementation would be gating the Write-en by the lock mechanism for both the registers (depicted in Fig. 16.8).

*RTL Code with complete specification:*

```
if
  (Addr_sel == 0 OR Addr_sel == 1) AND Lock==1
          Write_En_in==0
Else
          Write_En_in == Write_En
```

Apart from complete specifications, another critical aspect of designing security features is the definition of right size mitigations. For instance, the security analyst must be able to answer if it is required to design security mechanism for all the registers of the USB IP. If the security features are not adequate, then there is a possibility that some registers of the untrusted IP might contain malicious software. On the other hand, overprotection of the assets can be detrimental to the functional flow of the SoC, and can lead to the obsoleteness of security mechanisms.

## 16.6 THREAT MODELING

Threat modeling is the activity for optimizing SoC security by identifying objectives and vulnerabilities, and defining countermeasures to prevent, or mitigate the effects of, threats to the system. As noted above, it is a vital part of the security architecture definition. It is also a key part of the security validation, particularly in negative testing and white-box hacking activities. Threat modeling roughly involves the following five steps, which are iterated until completion:

**Asset definition.** Identify the system assets governing protection. This requires identification of IPs and the point of system execution, where the assets originate. As discussed above, this includes statically defined assets, and those generated during system execution.

**Policy specification.** For each asset, identify the policies that involve it. Note that a policy may "involve" an asset without specifying direct access control for it. For example, a policy may specify how a secure key $\mathcal{K}$ can be accessed by a specific IP. This in turn may imply how the controller of the fuse, where $\mathcal{K}$ is programmed can communicate with other IPs during boot process for key distribution.

**Attack surface identification.** For each asset, identify potential adversarial actions that can subvert policies governing the asset. This requires identification, analysis, and documentation of each potential "entry point", that is, any interface that transfers data relevant to the asset to an untrusted region.

The entry point depends on the category of the potential adversary considered in the attack, for example, a covert-channel adversary can make use of nonfunctional design characteristics, such as power consumption or temperature to infer the ongoing computation.

**Risk assessment.** The potential for an adversary to subvert a security objective does not, in and of itself, warrant mitigation strategies. The risk assessment and analysis are defined in terms of the so-called DREAD paradigm, composed of the following five components: (a) Damage potential; (b) Reproducibility; (c) Exploitability, that is, the skill and resource required by the adversary to perform the attack; (d) Affected systems, for instance, whether the attack can affect a single system or tens or millions; and (e) Discoverability. In addition to the attack itself, one needs to analyze factors, such as the likelihood that the attack can occur on-field, and the motives of the adversary.

**Threat mitigation:** Once the risk is considered substantial, given the likelihood of the attack, protection mechanisms are defined, and the analysis must be performed again on the modified system.

**Implementation example:** Consider protecting a system against code injection attacks by malicious or rogue IPs by overwriting code segments through direct memory access (DMA). The assets being considered here are appropriate regions of memory hierarchy (including cache, SRAM, secondary storage), and the governing policy may be to define DMA-protected regions, where DMA access is disallowed. The security architect needs to go through all memory access points in the system execution, identify memory access requests to DMA-protected regions, and set up mechanisms, so that DMA requests to all protected accesses will fail. Once this is done, the enhanced system must be evaluated for additional potential attacks, including attacks that can potentially exploit the newly set-up protection mechanisms themselves. Such checks are performed typically via negative testing, that is, looking beyond what is specified to identify if the underlying security requirements can be subverted. For example, such testing may involve looking for ways to access the DMA-protected memory regions, other than directly performing a DMA access. The process is iterative and highly creative, resulting in a collection of increasingly complex line-up of protection mechanisms, until the mitigation is considered sufficient with respect to the risk assessment.

In the following subsections, we will describe some practical attacks on SoC that exploit either functional or side-channel bugs, and describe possible countermeasures.

## 16.6.1 SOFTWARE-INDUCED HARDWARE FAULTS

Many attacks performed in recent times have shown faults in hardware can be induced through software leading to security issues. Next, we provide some examples of such attacks.

### 16.6.1.1 CLKSCREW

CLKSCREW is a prime example of how security-oblivious performance tweaks can lead to major security breaches. This particular fault can be introduced in the hardware directly from software, and can lead to privilege escalation, and even the stealing of encryption keys from the TEE of the device [6]. Dynamic voltage and frequency scaling (DVFS) [7] is a widely used approach to improve energy efficiency of a processor. In this approach, voltage and frequency of a processor are dynamically scaled to save power, and reduce heating effect. There can, however, be bugs in a DVFS system that can allow an attack to happen. But to understand the attack, let us first look at how DVFS is implemented.

### DVFS Implementation

**Hardware-level support:** When a complex SoC is designed, different IPs coming from different vendors may provide widely varying functionality and performance. They also typically have their own voltage and current requirements. For example, the voltage requirement for a processor core is likely to be different from the memory IP, or the communication IP. Hence, in order to properly integrate these components, designers include several voltage regulators [6], and embed them into the power management integrated circuit (PMIC) [8]. Moreover, to regulate different frequencies, a frequency synthesizer is typically integrated into the processor. This frequency synthesizer/phase-locked loop (PLL) circuit can output frequencies within a specified range, with the step function depending on the implementation. For example, in a Nexus 6 device, a standard PLL circuit provides a base frequency of 300 MHz. A high-frequency PLL (HFPLL) is responsible for the dynamic modulation of the output frequency. For fine-tuning, half the signal from the HFPLL is channeled through a frequency divider [6].

**Software-level support:** PMIC drivers [9,10] are provided by the vendor to control the hardware level regulators. Linux CPUfreq can perform OS-level power management by assessing the requirements of the system, and indirectly instructing the hardware regulators to make changes to the frequencies and voltages of different components. It is important to note that an application software cannot directly regulate the voltage or frequency, but can make changes to certain registers, which are later read by the hardware to perform the actual voltage/frequency scaling [6].

### CLKSCREW Fault:

CLKSCREW fault may occur if we “overclock” (i.e., apply higher than rated maximum clock frequency) or undervolt (meaning, apply lower than rated minimum voltage) the system. Let us first describe a few basic concepts before we move on to the fault. In a standard delay flip-flop, a change in the output (Q) happens if the value at the input (D) is switched, and if the flip flop detects a rising clock edge. Typically, between two flip-flops, we have combinational logic. Let us assume $T_{\text{clk}}$ is the clock cycle period; $T_{\text{FF}}$ is the time for which the input to the flip-flop must be kept stable; $T_{\text{setup}}$ is time for which the input signal must be stable before the clock edge appears; $T_{\text{max\_path}}$ is the delay of the combinational circuit, and $K$ is a constant for the assumed microarchitecture [6]. In Fig. 16.9 we see the above-mentioned variables and their role during the digital circuit operation. Therefore, the condition below must hold to ensure no faults triggers in the circuit.

$$T_{\text{clk}} \geq T_{\text{FF}} + T_{\text{max\_path}} + T_{\text{setup}} + K$$

If the constraint above is violated by overclocking, and thus reducing $T_{\text{clk},}$ or by undervolting, and thus increasing the $T_{\text{max\_path}}$, then we will be able to introduce a hardware fault [6]. Due to the constraint violation, the output of the second flip-flop fails to switch state because of the input delay, as shown in Fig. 16.10. Note that this fault can be induced into the hardware by a rogue software.

### Attack Based on CLKSCREW Fault:

Let us take a practical example of how CLKSCREW fault can leak an encryption key. We have noticed in Chapter 8 that differential fault attack, or DFA [11], can guess the AES keys if we can obtain a pair of ciphertext from a plaintext, such that one of the ciphertexts was a victim of single corrupted computation. DFA can reduce the key search space from $2^{128}$ (for a 128-bit AES Key) to $2^{12}$, if we can introduce a random single-byte data corruption at the 7th AES round, and the corrupted data feeds

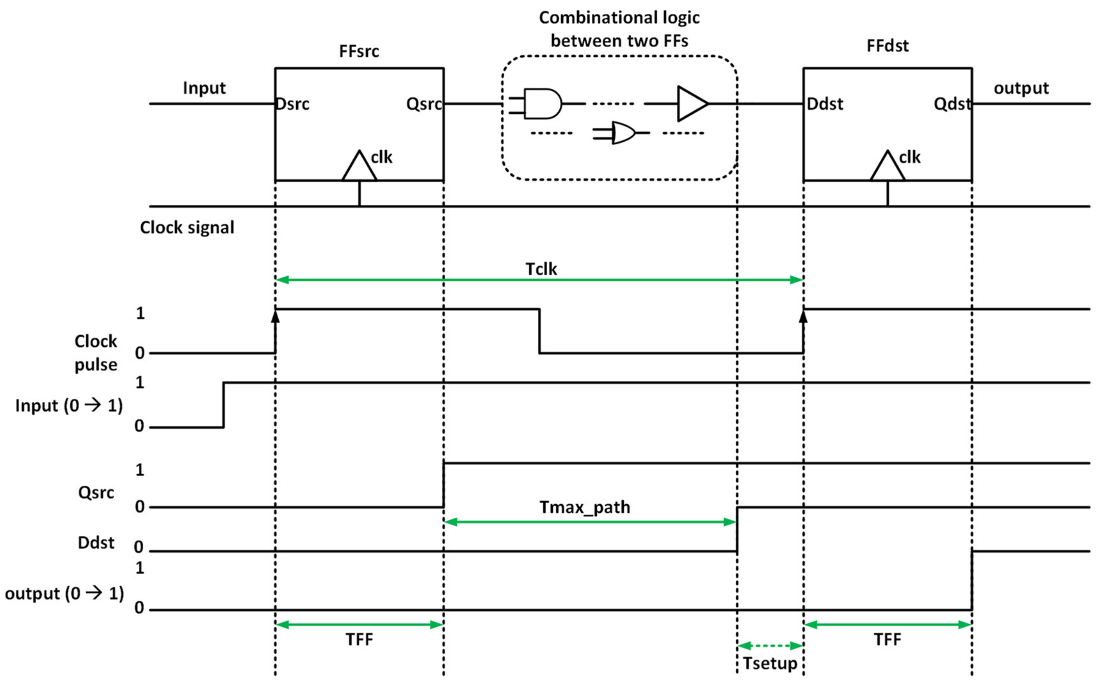

**FIGURE 16.9**

Different timing variables during the operation of a standard digital circuit [6].

into the next round. Once the search space is reduced to $2^{12}$, one can perform brute-force search to find the correct key [6]. The data corruption can be achieved through the CLKSCREW Method. This is an example of a small exploit of the fault from a much bigger possible fault space.

### Defense Against CLKSCREW Fault:

One easy countermeasure against such attack is to impose a hard limit to the upper and lower values of voltage and frequency, using extra limit-checking components, or by using e-fuses [6]. This mechanism imposes a constraint on the device during design phase, but to find the true operational limits, we must run intensive electrical testing after the device is manufactured. Process variation during fabrication can also impose extra variance making this solution very hard to uniformly implement across different devices and designs.

### *16.6.1.2 Rowhammer Attack*

Rowhammer is a recently reported system level attack that introduces a bit flip in the DRAM memory and can lead to privilege escalation [12], or other malicious effects. Bitflip errors can be randomly introduced in the memory due to background radiations and neutrons from cosmic ray secondaries [13]. We will discuss remedies in later sections. Researchers have reported reliable solutions to this

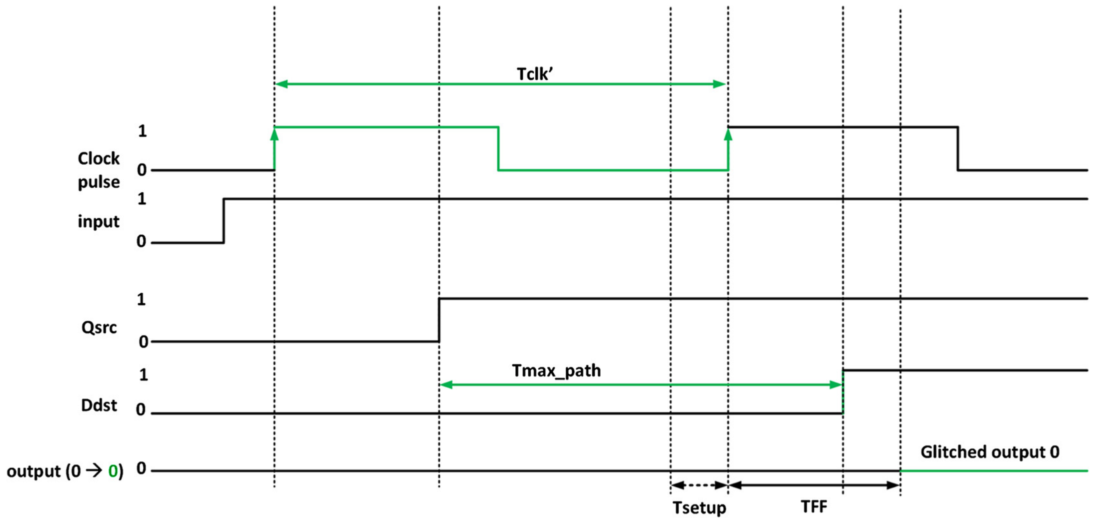

**FIGURE 16.10**

Glitch introduced in the output of the second flip-flop due to violation of the timing constraint [6].

problem with certain noncritical drawbacks. These solutions—although designed for normal bitflip error correction—works well towards mitigating a Rowhammer attack.

Bit errors that can be controlled to an extent, and are repeatable, impose a real threat to the security of the system. Rowhammer is one such technique that allows an attacker to cause a targeted bitflip in a certain memory location, allowing read access to restricted memory, and can even cause a privilege escalation. Eighty-five percent (85%) of the DDR3 modules tested in [14] were found to be susceptible to Rowhammer attack [15]. Even DDR4 modules can be attacked, using the Rowhammer exploit as pointed out in [16].

## Rowhammer Attack Model

As can be seen in Fig. 16.11, we can visualize DRAM [12] as a rectangular block, with each row representing a word of a certain length, and number of rows determining the overall capacity of the DRAM. To access a row or word in the memory we perform the following steps:

1. Allow the Row-buffer access to the selected row. This makes the Row-buffer hold the word from the selected row.
2. Read the information from the row buffer.
3. Disconnect the Row-buffer from the previously selected row, so that the next word can be accessed.

To carry out a Rowhammer attack, an attacker performs the following: [14]

1. Select a target row in DRAM to introduce a bitflip.
2. Rapidly access the adjacent row of the target row to cause the bitflip.
3. Exploit the bitflip to gain access to the system.

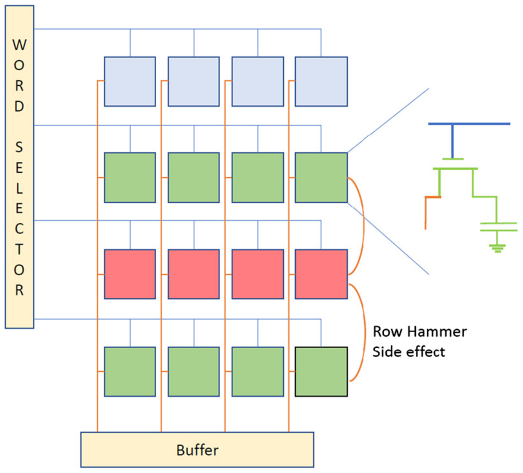

**FIGURE 16.11**

Standard DRAM structure from high-level perspective. Green (medium gray in print version) rows are being constantly accessed leading to a bitflip in the red (dark gray in print version) row. Note that this figure shows the double-sided hammering. The standard variant of the attack only makes repeated access to one of the adjacent rows of the target row.

For a DDR3 Ram [14], it is observed that 139,000 or more subsequent memory access can exhibit a memory error.

A variant of this attack is shown in Fig. 16.11. This is termed as double-sided hammering and involves high-frequency access to both of the adjacent memory rows of the target row [14]. This version of the attack has a higher chance of success and requires less access frequency than the original.

The following is a sample code [12,14,17] that can lead to a Rowhammer attack.

*Rowhammer attack code*

```
codeXYZ:
  mov (X), %eax   // read from address X
  mov (Y), %ebx   // read from address Y
  clflush (X)     // flush cache for address X
  clflush (Y)     // flush cache for address Y
  mfence
  jmp CodeXYZ
```

In the code above, the memory locations X and Y are repeatedly accessed to trigger a Rowhammer attack. Every time X and Y are fetched from the main memory, a copy is stored in the cache. Next time the code tries to access X and Y memory locations, the data will be fetched from the cache instead of main memory, provided that dirty bit is not set. This does not allow the attacker to trigger the Rowhammer attack, and that is why the attacker uses clflush() to release the content of the memory locations X and Y from the cache after each fetch, so that every access to X and Y is an access to the main memory

DRAM. On March 9, 2015, [14], Google's Project Zero exposed two working exploits for Rowhammer attack that caused privilege escalation. The first attack was on the Google Native Client (NaCl) [12]. The attacker was able to escape from within a sandbox, and directly execute system calls. In the other attack, page table entries [12] were modified by coupling memory, spraying with Rowhammer disturbance error. Both of these attacks rely on the clflush() call, which in x86-64 architecture cannot be converted into a privileged function. There are also ways [18] to perform a Rowhammer attack without the use of clflush(). The idea is to cause cache misses, so that the same main memory location is repeatedly accessed, triggering the Rowhammer effect. To achieve this effect, a particular memory access pattern is formulated, based on the cache replacement policy used by the target OS [14]. Although this method appears to be unscalable across different OS and cache replacement policies, there are certain adaptive cache eviction policies proposed to address the issue [12]. This particular flavor of Row-hammer attack is called the memory eviction attack.

#### Cause of Rowhammer Attack

Hammering a specific location of a DRAM memory electrically interferes with neighboring rows. It causes voltage fluctuation, which in turn causes the adjacent row to discharge faster than usual, and if the memory module is unable to refresh the cell in time, we see a bitflip [14]. Increase in DRAM density has led to memory cells holding smaller charges, while also being closely packed together. This fact makes the cells vulnerable to electromagnetic interactions from neighboring cells, which lead to the introduction of memory error.

#### Countermeasures Against Rowhammer Attack

**Error-correction code (ECC):** Although not intended as a countermeasure to Rowhammer attack, ECC works relatively well in dealing with this problem [12]. Single-error correction and double-error detection (SECDED) Hamming code [13] is an extremely popular ECC mechanism. Chipkill ECC ensures correction of multiple bit errors, up to full-chip data recovery, and can be used, but with high overhead. Even Non-ECC chips with parity support can prevent a Rowhammer attack by detecting it.

**Avoid clflush:** A few of the Rowhammer attacks—as discussed above—can be mitigated by modifying systems, such that they do not allow the clflush statement to execute [17].

**Shared libraries:** Shared libraries being available across processes allow the attacker to perform Rowhammer attack on these codes to escalate privilege. If libraries are not shared among processes with different privilege, then clflush, and memory eviction attack may be prevented [15].

### 16.6.2 SOFTWARE-INDUCED HARDWARE TROJAN ATTACKS

Recall from Chapter 5 that hardware Trojans are malicious logic in a design that can be triggered under rare circumstances, such as by an external signal, or at certain internal circuit condition, so as to avoid detection. The trigger condition initiates the Trojan functionality, and once triggered, the payload of the Trojan makes some modification in the data stream, or can send out some information from the data stream to the attacker. In this section, we look at hardware Trojans that can be activated by a signal sent from the software layer [21].

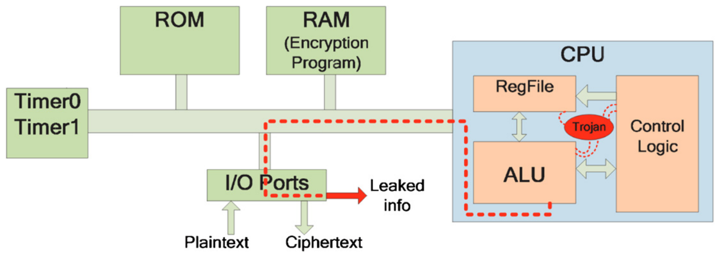

**FIGURE 16.12**

Example of a software controlled hardware Trojan.

#### *16.6.2.1 Hardware Trojan Trigger on Microprocessor*

Hardware Trojans if triggered based on inputs to a processor, or based on its output, can be exploited from the software layer. Figure 16.12 shows a Trojan that is inserted in a CPU, and based on the data stream inside the control logic, ALU and registers, leak information outside the system via the I/O ports [21]. Software exploitable hardware Trojans could be designed to support general attacks with variable payload effects, defined by the malicious software. However, such Trojans are more suitable for general-purpose processors or complex embedded processors that already have hardware supported security features, where various attacks could be performed, based on corrupting the security features through Trojan-induced backdoors [21].

##### Trojan Trigger Condition

The simplest Trojan is an always-on Trojan, which does not require any trigger condition to start malfunctioning. Though generally easier to implement, it is likely to get detected during post-manufacturing testing, due to the evident side-channel footprint. To avoid this, it has been proposed to make the Trojan trigger condition either controllable externally, by an attacker, or to use rare conditions in the internal circuitry to activate the Trojan [21].

Three aspects of a processor can be leveraged by a Trojan to act as a trigger condition from the software level [21]. They are:

1. Specific sequence of instructions.
2. Specific sequence of data.
3. Combination of both instruction, and data sequence.

This gives the control of the Trojan entirely to the attacker. For a processor running an operating system that supports multiple users, the trigger condition can be made even more complex to avoid accidental discovery by a defense mechanism, or during standard verification.

##### Trojan Payload

Regarding the Trojan payload, many options have been proposed in the literature, starting from simply inverting the data at some internal node, corrupting memory or primary output, or leaking sensitive information stored inside the hardware [21]. Channels available for leaking information can be realized through output ports, modulation of existing outgoing information, or the carrier frequency, phase, and

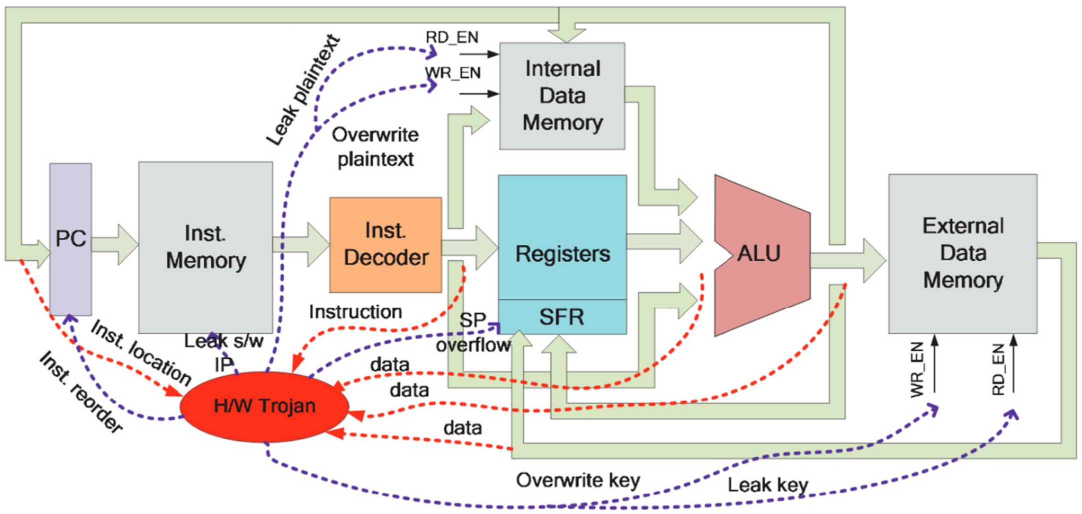

**FIGURE 16.13**

Possible software induced Trojan triggers.

amplitude to piggyback on existing modes of communication, or side-channels, such as power trace and EM radiation. For example, as shown in Fig. 16.13, a Trojan payload can be constructed to leak secret information during operation through output ports. When an instruction is fetched from the instruction memory to the register, it is also passed onto the LED ports for display. In reality, the information-leakage channel could be temporarily unused ports, or other side channels. A payload causing malfunction can perform an illegal memory write, or modify stack pointers, or change the inputs at the branch predictor.

### 16.6.3 SOFTWARE-INDUCED SIDE-CHANNEL ATTACK

Given the little effort spent on the cross-layer security during design time, attackers with a good understanding of both hardware and software components of a system can come up with a combined attack that can cripple the system. In this section, we will look at some well-known exploits that can be carried out, due to hardware vulnerability, but with the attack vector originating from the software layer. Although they rely on hardware bugs, these attacks, in most cases, can be remotely performed, and potentially have a huge cascading impact in the computing world and the world economy. The current solutions to these attacks are also discussed, but none are without major drawbacks [12,22,23].

#### *16.6.3.1 Spectre*

By exploiting the speculative execution of modern processors, Spectre attack can trick the system into executing instructions that are not supposed to be executed given the current system state [22]. This exploit, coupled with a side-channel attack, can lead to the exposure and theft of critical secret information from a victim system. This particular vulnerability can be exploited in a wide range of

AMD, Intel, and ARM processors affecting billions of systems. There are still no conclusive solutions to this problem and, as of writing, this remains an open field of research. In this section, we shall look into some concepts required to understand Spectre attack, the spectre attack model, and some proposed remedies.

### Out-of-Order Execution

In order to enhance performance and parallelize a program, which is by construct sequential, modern processors execute instructions from different parts of the program in parallel. It may happen that one instruction being executed is preceded by a set of instructions, which are yet to execute. This particular nature of executing a code is known as out-of-order execution [22].

### Branch Prediction

When a branch instruction is encountered, the processor makes a prediction on the direction in which the program will flow, and starts the speculative execution. The performance improvement due to this scheme is directly dependent on the number of right guesses the processor makes. For making a guess, certain components are implemented in the system. Typically, a predictor comprises of local and global predictors for improved accuracy. To further increase performance, whenever a branch instruction is executed, the corresponding correct jump location is cached in the Branch Target Buffer (BTB), and can be used as a guess for the same branch instruction when it is again executed [24].

### Speculative Execution

While performing out-of-order execution, it may happen that the processor arrives at a branch for which the condition is dependent on values that are yet to be computed in preceding instructions. At this stage the processor has two choices, either wait for the preceding instructions to complete and incur a huge performance delay, or speculate [22], and guess the result of the conditional branch. After the guess is made, the current register state is stored as a checkpoint, and subsequent instructions start to execute. Once the previous instructions on which the condition of branch dependent on are finished, the guess made is validated, and if the guess was wrong, the program state is reverted back to a checkpoint, and the execution of the correct path is initiated. Also in case of a wrong guess, all pending instructions are abandoned, so that it does not make any visible effect.

### Spectre Exploit on Conditional Branch

The main idea is to use the spectre vulnerability to read unauthorized data during speculative execution, and retrieve the data using a side-channel attack. Let us first look at how to read the unauthorized data. For the code below, let us assume that the code is part of kernel syscall.

*Spectre exploit on conditional branch [22]*

```
if (i < SizeOfArray1)
   A = Array2[Array1[i] * 256];
```

The attack follows the following steps:

1. **Train the predictor:** At first the conditional statement is accessed using values of i that are truly less than SizeOfArray1. With more such accesses, the chance of the processor to guess i < SizeOfArray1 increases.

2. **Craft i:** Maliciously pick a value for i, such that Array1[i] denotes a secret value ‘S’ in the memory that should not be accessible by the current process.
3. **Ensure conditions:** It is important to make sure that SizeOfArray1 and Array2 are not in cache, but the secret S is in the cache (which can be difficult to achieve).
4. **Side-channel probe:** While the processor is busy fetching SizeOfArray1 and Array2 from the main memory, the vulnerable data can be probed through side channel.

The first and third steps can be naturally present, or forced by the attacker. To evict SizeOfArray1 and Array2 from the cache, the attacker can dump a huge amount of random data into the cache indirectly by reading them from the memory. The attacker needs to know the cache replacement policy being used by the OS to properly carry out this step. To fetch the value S in the cache, the attacker can invoke a function that uses S. For example, if S is a cryptographic key, the attacker can trick the kernel into using the encryption function, which is easily accomplished. The final side-channel probe can be accomplished by identifying the change in cache configuration, and inferring information about the asset [22].

### Side-Channel Probe to Complete the Attack

After the sensitive data is accessed, and is stored in the cache, the attacker has about 200 instruction execution time [22] to read the sensitive data from the cache. There are several ways this final step can be carried out. If the attacker has access to Array2, then the value stored can be easily probed by detecting the cache state change. Another alternative method is to use prime-and-probe [25] attack. In this method the attacker accesses known attack data from the main memory, so that the cache can be filled with those data. Once the cache is filled with known attack data, for any new data to be stored in the cache, some portion of the attacker data is evicted, and that can be traced by the attacker to infer information about the asset.

### Evict+Time: A Variation

In this variant of the Spectre exploit [22,25], the attacker first trains the processor to predict wrongly during the speculation. Assume ‘i’ contains a secret, which is otherwise inaccessible to the process. Now during speculative execution, the attacker can perform the read Array1[i], even if i is out of bound for the Array1. Evict + Time attack assumes that the value of i is initially in the cache, then a portion of the cache is evicted through accessing his or her own memory which maps to the same cache set. If during this eviction the i is also evicted, then the subsequent read Array1[i] instruction will take a longer time to execute. If not, it will be fast, as it remains in the cache. The attacker systematically evicts portions of the cache, and notes the time of execution to infer information about the asset.

```
if (Predicted True but false)
  read Array1[i]
read [j]
```

### Spectre Exploit on Branch Target Buffer

The second spectre exploit involves manipulating indirect branches by poisoning the BTB. Indirect calls or branches are common occurrences at the assembly level. They are the result of programming abstractions, such as function pointers and object-oriented class inheritance, both of which resolve to jumps with destination addresses realized at runtime. As you may recall, the BTB maps the address of the source instruction to a destination address. However, the BTB only uses the lower bits of the source instruction. This design choice creates the potential for aliased addresses. Therefore, the attacker can fill the BTB with illegal destinations from source addresses that are aliases of the target branch instruction. Now, when the victim indirect branch is encountered with an uncached destination, the processor will jump to the illegal address provided by the tainted BTB, and begin speculative execution. The attacker must choose addresses of gadget code to be speculatively executed that will access secret memory, and leave evidence to be detected by side-channel attacks.

#### Spectre Prime: A Variation

A variant of spectre called Spectre Prime [26] uses Prime+Probe threat model. Unlike the normal Spectre, this affects a system with multiple cores. In multiple core systems, there are separate caches associated with each core. If one of the cores makes a change to a particular resource in its cache, then it is necessary that the caches of the other cores reflect the change as well, in order to avoid incoherent reads.

#### Countermeasures Against Spectre

There are no attractive solution for Spectre attack as of yet, but certain solutions have been proposed. If the processor does not perform speculative execution in a sensitive section of the code, then that can perhaps stop the Spectre attack [22]. Serializing instructions [27] can sometimes perform exactly that and halt some of the attacks for Intel x85 processors, but this is not a complete solution and not applicable for all processors. In addition to disabling, hyperthreading, and forcing a branch prediction state, refreshing at each context switch is another proposed solution, but is probably not possible to implement in current architectures [22,28].

### *16.6.3.2 Meltdown*

Meltdown [23] is a side-channel attack made possible due to out-of-order execution in a majority of the processors we use. Intel microarchitectures released since 2010 are vulnerable to this attack, and more likely than not, processors released even before are also vulnerable. The attack can be carried out in any OS, and is purely a hardware vulnerability. Memory isolation is an important security feature of any modern system, which essentially partitions the memory, and allows access to each partition, based on the privilege level of the process making the memory access request. The Meltdown exploit, however, can completely undermine memory isolation, and lead to the access of kernel-memory by user processes. This allows an attacker to steal critical information stored in kernel memory, such as passwords and encryption keys.

#### Out-of-Order Execution

The main reason for the success of Meltdown exploit is out-of-order execution [23]. As previously discussed, out-of-order execution is a performance tweak, that allows a processor to predict jump addresses and branch directions, so that it can compute ahead while the real address and branch condition are being evaluated [22]. A more detailed discussion is present in the Spectre Section. Any instruction that is executed out-of-order and vulnerable to potential side-channel attacks is termed as a transient instruction [22].

### The Attack Environment

For the Meltdown attack certain assumptions are made on the attack environment [23]. They are as follows:

1. **Target:** The targets of this attack are any personal computers and different virtual machines that are hosted on the cloud [23].
2. **Attacker initial access:** To initiate the attack, the attacker must have full underprivileged access to the personal computer, or the virtual machine [23].
3. **Physical access:** No physical access to the systems is required by the attacker for using Meltdown Exploit [23].
4. **Defenses assumed:** For this attack, the systems can be protected by address space layout randomization (ASLR) and Kernel ASLR. Also, CPU features, such as SMAP, NX, SMEP, and PXN can be present [23]. All these will fail to defend the system against Meltdown.
5. **OS bugs:** The attacker assumes that the operating system has no bugs, and does not rely on the exploitation of any to gain access to secret information, or to attain kernel privilege.

### The Attack Model

For carrying out a successful Meltdown attack, the steps are as follows [23]:

1. **Choose target memory:** First the attacker needs to decide, which directly inaccessible memory location contains valuable information that is worth extracting by this exploit [23].
2. **Load:** Load the target memory address into a register [23].
3. **Transient instruction exploit:** A transient instruction originated due to out-of-order execution is used to access a particular line of the cache, based on what is stored in the register [23].
4. **Flush+Reload:** Using Flush+Reload [29] side-channel attack, the attacker can perform a fine-grained version of Evict+Time to infer information about the asset by using the *cache flush* command [23].
5. **Repeat if needed:** These steps can be repeated to dump as much of the kernel memory as required. Theoretically, the attacker can dump the entire physical memory if he or she wishes so [23].

In any modern OS, the kernel memory can be addressed from any user process, but the processor triggers an exception if the user process does not have the permission to access the kernel memory as shown in Fig. 16.14. But due to speculation, while the exception is being processed, instructions are executed beforehand, and cause a leak in the kernel memory. Once the data is leaked, Meltdown exploits the Flush+Reload [29] technique to retrieve the stolen data from the cache [23]. Flush+Reload [29] is similar to Evict+Time [25], and uses cache-manipulation techniques. In the flush phase, the memory portion being monitored is evicted from the cache. In the wait phase of the attack, if the victim access that same memory location, the content comes back to the cache. In the reload phase, the attacker tries to access the same memory location, and if the content is in the cache, then the reload operation will take a much shorter time as compared to when the content is not in the cache. This allows the attacker to know if the right target location of the cache is flushed, and—subsequently—can infer the content of the target memory location [23].

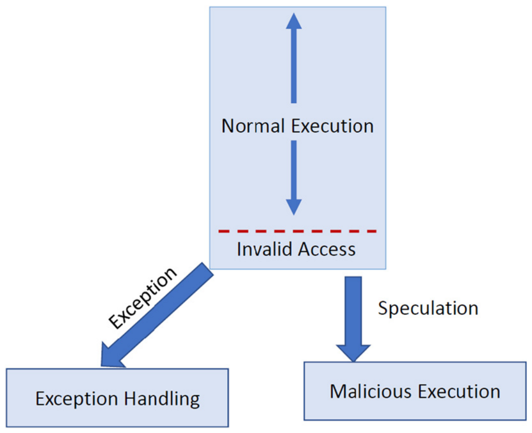

**FIGURE 16.14**

Meltdown vulnerability [23].

### Meltdown Prime: A Variant

This is a multicore variant of the Meltdown exploit and is very similar to Spectre Prime and the original Meltdown itself [23,26]. The exploit preys on the vulnerability of the cache coherence protocols that ensure coherence among multiple cache instances of different cores. The vulnerable cache-line invalidation scheme of these protocols, along with speculation, allows the exploit to go through.

### Countermeasures

Completely removing speculation and out-of-order execution is the trivial solution to the problem, but not a practical one, as speculation and out-of-order execution leads to a huge performance gain. This exploit is kind of a race between the attacker trying to retrieve sensitive information, and the processor trying to determine if the speculation was valid or not. The processor, unfortunately, takes a very long time to determine the fate of the speculative executions and it may happen that by the time speculation is validated or invalidated, around 200 instructions are already executed along the predicted path [22, 23]. If the position of the kernel space and user space in the virtual memory space can be hard fixed, then using as few as a 1-bit comparison, the processor can determine if an invalid access is going to take place. This leaves far less room for the attacker to carry out the attack, though, in fact, it may be impossible to entirely solve this class of problems. But it is important to note that these solutions are not applicable for Spectre [22,23].

### Comparison Between Spectre and Meltdown

Meltdown can be only exploited currently on specific architectures, but Spectre is pervasive and affects all processors that employ speculative execution. For both Meltdown and Spectre, the attacker must be allowed to execute code in the system as an unprivileged user. Both the exploits use the inherent vulnerability that exists due to speculation and out-of-order execution. Fixing Meltdown at the software level is possible, but with high overhead. As for Spectre, even software solution is not easy to implement. Both of these vulnerabilities can greatly benefit from efficient hardware-based solutions.

## 16.7 HANDS-ON EXPERIMENT: SoC SECURITY POLICY

### 16.7.1 OBJECTIVE

*This experiment is designed to help students understand components of SoC architecture, common security-critical assets in an SoC, which are accessible to constituent IP blocks, and protection mechanisms based on security policies.*

### 16.7.2 METHOD

*First, students will map a given Verilog description of a simple SoC consisting of 4-10 IP blocks, including processor core, memory, cryptomodule, and communication module. The IPs will be connected using a point-to-point interconnect framework. Students will perform functional simulation of the SoC to understand how the IPs work, and interact with each other by way of a SoC to appreciate how a key used by cryptomodule is vulnerable to unauthorized access. Next, students will design a few simple access-control security policies to safeguard the cryptokey against unauthorized access.*

### 16.7.3 LEARNING OUTCOME

*Through this experiment, students are expected to be familiar with several important notions in system level security. They are expected to understand the security issues arising from possible CIA violations on on-chip assets, and policy-based solutions to protect these assets. Finally, students can use the experience to achieve better understanding of software attacks that can lead to CIA violations, and how security policy can help to achieve robust protection.*

### 16.7.4 ADVANCED OPTIONS

*Additional exploration on this topic can be done through investigation on more complex interconnect fabric, such as bus-based or NoC-based architecture. Students may also consider adding more IP blocks, including IPs that use firmware, and mount firmware-based attacks on security assets.*

*More details about the experiment are available in the supplementary document. Please visit: http://hwsecuritybook.org.*

## 16.8 EXERCISES

### 16.8.1 TRUE/FALSE QUESTIONS

1. Meltdown does not rely on speculative execution; it exploits only out-of-order execution.
2. Internet-of-things ecosystem cannot be attacked using any hardware-software exploit because it is too diverse for the attacker.
3. Software Trigged Hardware Trojans can be used to cause denial-of-service (DoS) attack.
4. Preventing the execution of the clflush (cache flush) command can deter all types of Rowhammer attacks.
5. During a Spectre exploit, once the processor starts to perform speculative execution, it gives the attacker theoretically unlimited amount of time to finish the attack.

6. Trusted Execution Environment can prevent DoS attacks.
7. Evict+Time requires knowledge of cache structure to properly trigger cache contention in a system.
8. Security policies no longer need to be defined when using Security Policy Enforcement Architectures.
9. Modifications to DfD are needed to maintain both security and controllability of internal signals.
10. For pre-silicon verification, formal verification can be applied on large designs.

### 16.8.2 SHORT-ANSWER TYPE QUESTIONS

1. What is the difference between Meltdown and Meltdown Prime?
2. What is the potential overhead if we disable out-of-order execution in an attempt to address the Spectre exploit?
3. In a 32-bit RISC architecture with 5-bit instruction opcode, what is the probability of activating a software-induced hardware Trojan with a trigger condition of 5 ADD instructions in a row?
4. Explain how ECC can sometimes deal with Rowhammer attack.
5. How can overclocking and undervolting lead to a CLKSCREW attack?
6. Which phase of SoC Verification can be used to detect side-channel attacks. Why?
7. Instead of the centralized policy engine described earlier, a designer decides to implement distributed policy enforcement (relevant security policies are described in the corresponding IP instead of a centralized engine). Describe two advantages of a centralized policy engine over a distributed one, and vice versa.
8. Which access control policies, if any, can address software induced hardware Trojans?
9. Describe an encryption method that is ideal for securing data in a DfD structure with cost-effective area overhead, and key management.
10. An attacker uses the same trigger condition from the Trojan in Question 3 Short Answer, to toggle the secure mode flag in a microprocessor with a trusted execution environment (TEE). Assess this Trojan using the DREAD method.

### 16.8.3 LONG-ANSWER TYPE QUESTIONS

1. Why is the Meltdown bug easier to mitigate than Spectre?
2. Describe the preventive measures that can be taken to address CLKSCREW exploit.
3. How is the Evict+Time variant of Spectre different from the standard version? Explain both methods and compare.
4. For pre-silicon verification, compare and contrast simulation and formal verification.
5. Where are the security loopholes in an IoT ecosystem? Point them out from a very high level perspective.
6. Using Fig. 16.7, what other vulnerabilities can exist in this scenario?
7. Describe the potential overhead of the Security Policy Architecture without access to the DfD infrastructure.
8. Even with thorough security policy implementation and analysis, vulnerabilities like Spectre and Meltdown still surface. Describe the potential drawbacks and complexities that exist in defining system wide security policies throughout the design process.
9. Briefly describe the security policies needed to protect the lock bit in Fig. 16.8. Explain each chosen policy with a violating scenario.

## REFERENCES

[1] IBM Microelectronic, Coreconnect bus architecture, IBM White Paper Google Scholar, 1999.
[2] D. Flynn, AMBA: enabling reusable on-chip designs, IEEE Micro 17 (4) (1997) 20–27.
[3] K. Rosenfeld, R. Karri, Security-aware SoC test access mechanisms, in: VLSI Test Symposium (VTS), 2011 IEEE 29th, IEEE, 2011, pp. 100–104.
[4] S.J. Greenwald, Discussion topic: what is the old security paradigm, in: Workshop on New Security Paradigms, 1998, pp. 107–118.
[5] A. Basak, S. Bhunia, S. Ray, A flexible architecture for systematic implementation of SoC security policies, in: Proceedings of the 34th International Conference on Computer-Aided Design, 2015.
[6] A. Tang, S. Sethumadhavan, S. Stolfo, CLKSCREW: exposing the perils of security-oblivious energy management, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 1057–1074.
[7] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, 5th ed., Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.
[8] F. Shearer, Power Management in Mobile Devices, Newnes, 2011.
[9] Qualcomm krait pmic frequency driver source code, [Online]. Available: https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/drivers/clk/qcom/clock-krait.c.
[10] Qualcomm krait pmic voltage regulator driver source code, [Online]. Available: https://android.googlesource.com/kernel/msm/+/android-msm-shamu-3.10-lollipop-mr1/arch/arm/mach-msm/krait-regulator.c.
[11] M. Tunstall, D. Mukhopadhyay, S. Ali, Differential fault analysis of the advanced encryption standard using a single fault, in: C.A. Ardagna, J. Zhou (Eds.), Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication, Springer Berlin Heidelberg, Berlin, Heidelberg, 2011, pp. 224–233.
[12] M. Seaborn, T. Dullien, Exploiting the DRAM rowhammer bug to gain kernel privileges: how to cause and exploit single bit errors, BlackHat, 2015.
[13] P.K. Lala, A Single Error Correcting and Double Error Detecting Coding Scheme for Computer Memory Systems, in: Proceedings. 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2003.
[14] Y. Kim, R. Daly, J. Kim, C. Fallin, J.H. Lee, D. Lee, C. Wilkerson, K. Lai, O. Mutlu, Flipping bits in memory without accessing them: an experimental study of dram disturbance errors, in: 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), June 2014, pp. 361–372.
[15] D. Gruss, C. Maurice, S. Mangard, Rowhammer.js: a remote software-induced fault attack in JavaScript, CoRR, arXiv: 1507.06955, 2015, [Online]. Available: http://arxiv.org/abs/1507.06955.
[16] P. Pessl, D. Gruss, C. Maurice, S. Mangard, Reverse engineering Intel DRAM addressing and exploitation, CoRR, arXiv: 1511.08756, 2015, [Online]. Available: http://arxiv.org/abs/1511.08756.
[17] M. Seaborn, T. Dullien, Exploiting the DRAM rowhammer bug to gain kernel privileges, Project Zero team at Google, [Online]. Available: https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.
[18] D. Gruss, C. Maurice, Rowhammer.js: a remote software-induced fault attack in JavaScript, GitHub.
[19] Semiconductor industry association (SIA), Global billings report history (3-month moving average) 1976-March 2009, [Online]. Available: http://www.sia-online.org/galleries/statistics/GSR1976-march09.xls, 2008.
[20] R.S. Chakraborty, S. Narasimhan, S. Bhunia, Hardware Trojan: threats and emerging solutions, in: 2009 IEEE International High Level Design Validation and Test Workshop, Nov 2009, pp. 166–171.
[21] X. Wang, T. Mal-Sarkar, A. Krishna, S. Narasimhan, S. Bhunia, Software exploitable hardware Trojans in embedded processor, in: 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Oct 2012, pp. 55–58.
[22] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: exploiting speculative execution, ArXiv e-prints, Jan. 2018.
[23] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown, ArXiv e-prints, Jan. 2018.
[24] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, M. Peinado, Inferring fine-grained control flow inside SGX enclaves with branch shadowing, in: 26th USENIX Security Symposium (USENIX Security 17), USENIX Association, Vancouver, BC, 2017, pp. 557–574.
[25] D.A. Osvik, A. Shamir, E. Tromer, Cache attacks and countermeasures: the case of AES, in: D. Pointcheval (Ed.), Topics in Cryptology – CT-RSA 2006, CT-RSA 2006, in: Lecture Notes in Computer Science, vol. 3860, Springer, Berlin, Heidelberg, 2006.

[26] C. Trippel, D. Lustig, M. Martonosi, MeltdownPrime and SpectrePrime: automatically-synthesized attacks exploiting invalidation-based coherence protocols, ArXiv e-prints, Feb. 2018.
[27] Intel 64 and IAa-32 architectures software developer manual, vol 3: system programmer's guide, section 8.3, 2016.
[28] Q. Ge, Y. Yarom, G. Heiser, Do hardware cache flushing operations actually meet our expectations?, CoRR, arXiv:1612.04474, 2016, [Online]. Available: http://arxiv.org/abs/1612.04474.
[29] Y. Yarom, K. Falkner, FLUSH+RELOAD: a high resolution, low noise, l3 cache side-channel attack, in: 23rd USENIX Security Symposium (USENIX Security 14), USENIX Association, San Diego, CA, 2014, pp. 719–732.
[30] A. Basak, S. Bhunia, S. Ray, Exploiting design-for-debug for flexible SoC security architecture, in: Proceedings of the 53rd Annual Design Automation Conference, ACM, p. 167.
[31] E. Ashfield, I. Field, P. Harrod, S. Houlihane, W. Orme, S. Woodhouse, Serial Wire Debug and the Coresight Debug and Trace Architecture, ARM Ltd., Cambridge, UK, 2006.
[32] IEEE, IEEE standard test access port and boundary scan architecture, IEEE Standards 11491, 2001.
[33] J. Portillo, E. John, S. Narasimhan, Building trust in 3PIP using asset-based security property verification, in: VLSI Test Symposium (VTS), 2016, pp. 1–6.