CHAPTER

SECURITY AND TRUST ASSESSMENT, AND DESIGN FOR SECURITY

13

CONTENTS

Hardware Security. https://doi.org/10.1016/B978-0-12-812477-2.00018-6

13.1 INTRODUCTION

With the emergence of information technology and its critical role in our daily lives, the risk of various cyber attacks is larger than ever before. Many security systems or devices have critical assurance requirements, e.g., high assurance electronic systems – military, aerospace, automotive, transportation, financial, and medical. Their failure may endanger human life and environment, cause serious damage to critical infrastructure, hinder personal privacy, and undermine the viability of the whole business sector. Even the perception that a system is more vulnerable than it really is, hindering, for example, paying with a credit card over the Internet, can significantly impede economic development. The defense against intrusion and unauthorized use of resources with software was given significant attention in the past. Security technologies, including antivirus, firewall, virtualization, cryptographic software, and security protocols, have been developed to make systems more secure.

While the battle between software developers and hackers has raged since the 1980's, the underlying hardware was generally considered safe and secure. However, in the last decade or so, the battlefield has expanded to hardware domain, since—in some aspects—emerging attacks on hardware are shown to be more effective and efficient than traditional software attacks. For example, while the cryptographic algorithms have been improved and have become extremely difficult (if not impossible) to break mathematically, their implementations are often not. It has been demonstrated that the security of cryptosystems, system on chips (SoCs), and microprocessor circuits can be compromised using timing analysis attacks [1], power analysis attacks [2], exploitation of design-for-test (DFT) structures [3] [4], and fault-injection attacks [5]. These attacks can effectively bypass the security mechanisms built in the software level, and put devices or systems at risk. These hardware-based attacks aim to exploit the vulnerabilities, within the hardware design, which are introduced either unintentionally or intentionally during the IC design flow.

Many security vulnerabilities in ICs can be unintentionally created by design mistakes and designers' lack of understanding of security vulnerabilities. Further, today's CAD tools are not equipped with understanding security vulnerabilities in integrated circuits. Therefore, a tool can introduce additional vulnerabilities in the circuit [6,7]. These vulnerabilities can facilitate attacks, such as fault-injection or side-channel-based attacks. Also, these vulnerabilities can cause sensitive information to be leaked through observable points, which are accessible to an attacker or give unauthorized access to an attacker to control, or affect a secure system.

Vulnerabilities can also be intentionally introduced in ICs in form of malicious modifications, generally referred to as hardware Trojans, or backdoors [8]. Due to short time-to-market constraints, design houses are increasingly being dependent on external entities (third parties) to procure IPs. Also, due to the ever-increasing cost of manufacturing ICs, design houses rely on untrusted foundries and assemblies for fabricating, testing, and packaging ICs. These untrusted third-party IP owners or foundries can insert hardware Trojans to create backdoors in the design, through which sensitive information can be leaked and other possible attacks (for example, denial of service and reduction in reliability) can be performed.

It is of paramount importance to identify security vulnerabilities during hardware design and validation process, and address them as early as possible due to the following reasons: 1) there is little or no flexibility in changing or updating post-fabricated integrated circuits; 2) the cost of fixing a vulnerability found at later stages during the design and fabrication processes is significantly higher, following the well-known rule-of-ten (the cost of detecting a faulty IC increases by an order of magnitude as one

advances through each stage of design flow). Moreover, if a vulnerability is discovered after manufacturing while the IC is in the field, it may cost a company millions of dollars in lost revenues and replacement costs.

Knowledge of the vulnerabilities and security assessment of a hardware design is not sufficient to protect it. A series of countermeasures/techniques is also required for each vulnerability to prevent adversaries from exploiting it. The development of countermeasures is a challenging task, as they must meet cost, performance, and time-to-market constraints, while ensuring a certain level of protection.

This chapter presents pre-silicon and post-silicon security and trust assessment techniques that can identify potential vulnerabilities in hardware design. Design for security techniques to address the potential vulnerabilities in hardware designs are also discussed.

13.2 SECURITY ASSETS AND ATTACK MODELS

To build a secure integrated circuit, a designer must decide what "assets" to protect, and the possible attacks to be investigated. Further, IC designers must also understand the players (attackers and defenders) and their role in the IC design supply chain. Three fundamental security factors (security assets, potential adversaries, and potential attacks) are associated with security assessment and countermeasures in integrated circuits; they are discussed below.

13.2.1 ASSETS

As defined in [11], an asset is a resource of value, which is worth protecting from the adversary. An asset may be a tangible object, such as a signal in a circuit design, or may be an intangible asset, such as controllability of a signal. Sample assets that must be protected in a SoC are listed below [12] (see Fig. 13.1):

- **On-device key,** namely, private key of an encryption algorithm. These assets are stored on-chip in some form of nonvolatile memory. If these are breached, the confidentiality requirement of the device is compromised.
- **Manufacturer firmware,** low-level program instructions and proprietary firmware. These assets have intellectual property values to the original manufactures. Compromising these assets allows an attacker to counterfeit the device or use pirated firmware on a different device with similar functionality.
- **On-device protected data,** such as user's personal information and meter reading. An attacker can invade someone's privacy by stealing these assets, or can tamper with these assets as in meter reading.
- **Device configuration,** such as, configuration data, determines which resources and services are available to particular users. An attacker may tamper with these assets to gain unauthorized access to these resources.
- **Entropy,** including random numbers generated for cryptographic primitives, for example, initializing vector or cryptographic key generation. Successful attacks on these assets weaken cryptographic strength of a device.

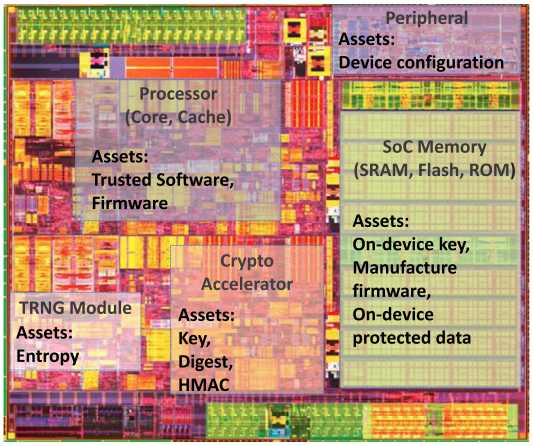

FIGURE 13.1

Example assets in a SoC.

The security assets are known to the hardware designers based on the target specifications of a design. For example, a designer knows the private encryption key used by the cryptomodule and its location in the SoC. Different types of assets and their locations in a SoC are shown in Fig. 13.1.

13.2.2 POTENTIAL ACCESS TO ASSETS

Usually, the aim of an attack is to obtain unauthorized access to assets. Typically, there are four types of such attacks, depending on attackers' abilities: remote attacks, noninvasive physical attacks, semi-invasive physical attacks, and invasive physical attacks.

Remote Attacks: In this case, an attacker has no physical access to the device. The attacker can still perform timing [1] and electromagnetic [13] side-channel attacks to remotely extract private key from devices, such as smartcards or microprocessors used in the cloud system. It has also been demonstrated that an attacker can remotely access the JTAG port and compromise the secret key stored in smartcard of a set-top box [14].

It is also possible to remotely access the scan structure of a chip. For example, in automotive applications, the SoC controlling different critical functions, such as brakes, power-train, and air-bags, goes into "test-mode" every time the car is turned off or on. This key-off/on tests ensure that the critical systems are tested and working correctly before every drive. However, modern cars can be remotely turned on or off, either by trusted parties, such as roadside assistance operators, or by malicious parties, as shown in the recent news [15]. Remotely turning the car on or off allows access to the SoC's test mode, which can be used to obtain information from the on-chip memory or impose unwanted functions.

Remote attacks also include the ones that exploit the weakness of hardware without requiring physical access, such as buffer overflow, integer overflow, heap corruption, format string, and globbing [16].

Noninvasive physical attacks: Basic noninvasive physical attacks consist of using the primary inputs and outputs to take advantage of security weaknesses in the design to obtain sensitive information. Additionally, more advanced attacks use JTAG debug, boundary scan I/O, and DFT structures to monitor and/or control system intermediate states, or snoop bus lines, and system signals [4]. Other noninvasive physical attacks consist of injecting faults to cause an error during the computation of cipher algorithms and exploit the faulty results to extract the asset, for example, private key for AES encryption. Finally, side-channel attacks (SCAs), such as power SCA or EM SCA fall into the category of noninvasive physical attacks. Noninvasive attacks usually require low budget, and do not cause the destruction of the device under attack.

Invasive physical attacks: These attacks are the most sophisticated and expensive attacks and require advanced technical skills and equipment. In a typical invasive physical attack, chemical processes or precision equipments can be used to physically remove micrometer-thin layers of the die of an electronic chip. Microprobes can then be used to read values on data buses, or inject faults into internal nets in the device to activate specific parts, and extract information. Such attacks are usually invasive result in the destruction of the device.

Semi-invasive physical attacks: Semi-invasive attacks fall between noninvasive and invasive physical attacks. These attacks pose a greater threat, because they are more effective than noninvasive attacks, but can be performed at a much lower cost than that of invasive physical attacks. Semi-invasive physical attacks usually require partial depackaging of the chip, or backside thinning to get access to its surface. Unlike invasive attacks, semi-invasive attacks do not require complete removal of the internal layers of the chip. Such attacks include injecting faults to modify SRAM cells content, and changing the state of a CMOS transistor to gain control of a chip's operation, or bypass its protection mechanisms [17].

13.2.3 POTENTIAL ADVERSARY

It is important to understand the potential adversaries who may utilize the security vulnerabilities to perform attacks. This can help designers in comprehending adversaries' capabilities and choosing right countermeasures depending on the targeted adversary and operation. The adversary might be an individual or an organized party who intends to acquire, damage, or disrupt an asset for which he/she does not have permission to access. Considering an integrated circuit design process and entities involved in it, adversaries can be categorized into insiders and outsiders. Figure 13.2 shows the potential adversaries in different stages of a SoC design process.

Insiders: The design and manufacturing of integrated circuits have become more sophisticated and globally distributed at present time. It creates a higher possibility for launching attacks by insiders who understand the details of the design. An insider could be a rogue employee who works for the design house and the system integrator, or could be an untrusted 3PIP or a foundry. Typically, an insider:

- Has direct access to the SoC design, either as an RTL or gate-level netlist, or as a GDSII layout file.
- Has high technical knowledge in the IC design and supply chain.
- Has the capability to make modifications to the design, for example, inserting hardware Trojans [8, 18]. These hardware Trojans can cause denial of service, or create backdoors in the design through which sensitive information can be leaked. Other possible insider attacks include reducing circuit reliability by manipulating circuit parameters and asset leakage.

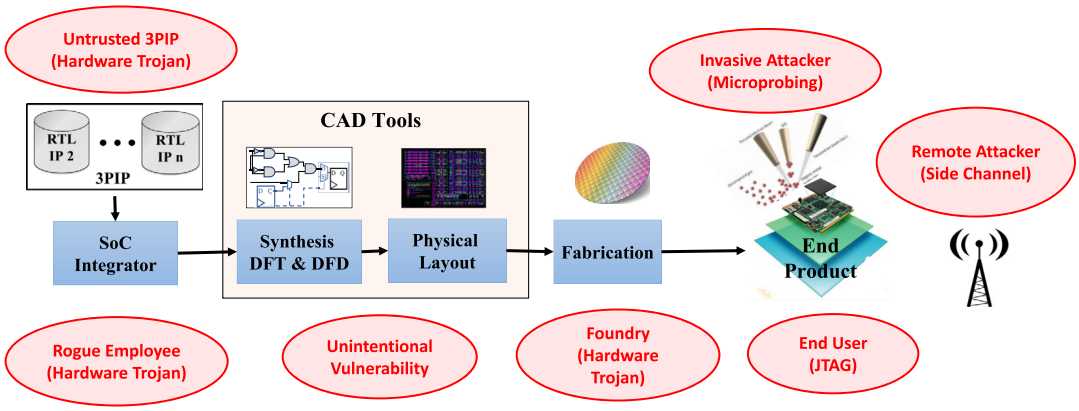

FIGURE 13.2

Potential adversaries in different stages of SoC design process.

Outsiders: This class of attackers is assumed to have access to end products in the market, for example, a packaged IC. Outsider attackers can be divided into three groups based on their capabilities:

- **Remote hacker:** These attackers have no physical access to the device. They must employ remote attacks described in Section 13.2.2, although they may have physical access to a similar device to develop their attack strategy. These attackers generally rely on exploiting software/hardware vulnerabilities, user errors, and design bugs to gain access to assets. Remote hackers include a wide spectrum of attackers, including hobbyist to state-sponsored attackers.
- **End-user:** This group of attackers typically aim to gain free access to contents and services. In this case, the curious end-users may rely on techniques already developed by professional attackers (sometimes available as exploit kit) to carry out their attack. For example, some hobbyists may find a way to jailbreak iPhone or Xbox gaming consoles, and post the procedure on social media, allowing end-users with much less expertise to duplicate the process. Jailbreaking allows users to install jailbreak programs and make companies, such as Apple or Microsoft, lose profits [19].
- **Invasive attacker:** These attackers are security experts and are usually sponsored by nation or industry competitors. Their motives are driven by financial, or political reasons. These groups are capable of executing the more expensive invasive and semiinvasive attacks described in Section 13.2.2.

An insider can introduce or exploit the vulnerabilities in a design more easily compared to an outsider. The main challenge for an outsider to perform an attack is that the internal functionality of the design may not always be known to the attacker. An outsider can reverse engineer the functionality of a chip, but this technique would require extensive resource and time.

The following sections describe the pre-silicon and post-silicon security and trust validation techniques to identify potential vulnerabilities in a hardware design.

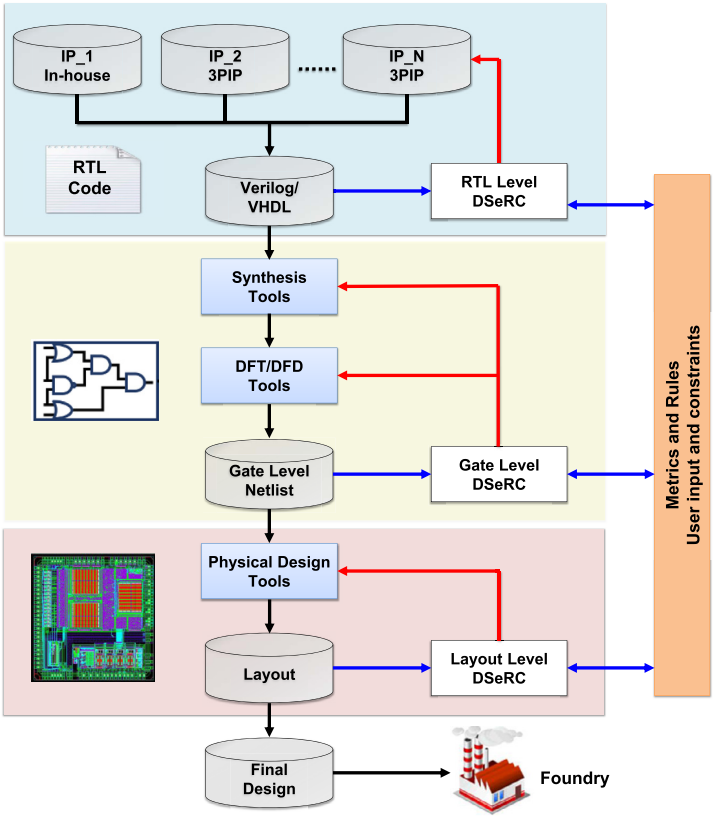

FIGURE 13.3

DSeRC framework.

13.3 PRE-SILICON SECURITY AND TRUST ASSESSMENT FOR SoCs

Different security and trust assessment techniques are employed at the design stage before the chip is fabricated to assess the security and trust issues in a hardware design. This section focuses on the Design Security Rule Check (DSeRC) framework to analyze different vulnerabilities in a hardware design, and consequently assess its security issues at design stage.

13.3.1 DSeRC: DESIGN SECURITY RULE CHECK

In order to identify and evaluate vulnerabilities associated with ICs, DSeRC framework can be integrated into the conventional digital IC design flow, as shown in Fig. 13.3. DSeRC framework reads the design files, constraints, and user input data, and checks for vulnerabilities at all levels of abstraction (RTL, gate-level, and physical layout level). Each of the vulnerabilities is to be tied with a set of rules and metrics, so that each design's security can be quantitatively measured. At RTL abstraction-level,

the DSeRC framework assesses the security of IPs, which are either developed in-house or procured from a third party, and provides feedback to design engineers, so that the identified security issues can be addressed. After resolving the security vulnerabilities at RTL, the design is synthesized into gate-level with design-for-test (DFT) and design-for-debug (DFD) structures being inserted. Then, DSeRC framework analyzes the gate-level netlist for security vulnerabilities. The same process is applied to the physical layout design. Through this process, the DSeRC framework allows the designers to identify and address security vulnerabilities at the earliest possible design steps. This significantly improves the security of ICs, and considerably reduces the development time and cost by lowering the time-to-market constraint. Also, the DSeRC framework allows the designers to quantitatively compare different implementations of the same design and, thereby, allows them to optimize performance without compromising the security. However, the DSeRC framework requires some defined inputs from the designer. For example, the security assets need be specified by hardware designers, based on the target specifications of a given design. Note that any technique that performs security assessment for SoCs at any abstraction level falls under the DSeRC concept discussed above.

The DSeRC framework is comprised of three major components: (i) list of vulnerabilities, (ii) metrics and rules to quantitatively assess the vulnerabilities, and (iii) CAD tools for automated security assessment.

13.3.1.1 Vulnerabilities

The vulnerability in a SoC means a weakness, which allows an adversary to exploit and gain access to assets by carrying out an attack. For the development of DSeRC framework, each vulnerability needs to be assigned to one or multiple proper abstraction levels, where it can be identified efficiently. Generally, an IC design goes through specification, RTL design, gate-level design, and, ultimately, physical layout design. DSeRC framework aims at identifying vulnerabilities as early as possible during the design flow, because late evaluation can lead to a long development cycle, and high design cost. Also, vulnerabilities in one stage, if not addressed, may introduce additional vulnerabilities during transition from one level to the next. This section categorizes vulnerabilities based on the abstraction levels (see Table 13.1).

Register-transfer level (RTL): The design specification is first described in a hardware description language (HDL) (for example, Verilog) to create the RTL abstraction of the design. Several attacks performed at the RTL have been discussed in the literature. For example, Fern et al. [20] demonstrated that "Don't-care" assignments in the RTL code can be leveraged as the source of vulnerability to implement hardware Trojans that leak assets. Additionally, in the RTL, hardware Trojans are most likely to be inserted at hard-to-control and hard-to-observe parts of the code [21]. Identifying hard-to-control and hard-to-observe parts of the code can help designers assess the susceptibility of the design to Trojan insertion at the RTL.

In general, vulnerabilities identified at the RTL are comparatively easier to address. However, some vulnerabilities, for instance, the susceptibility of the design to fault-injection or side-channel attacks, are much more challenging, if not impossible, to identify at this level.

Gate-level: The RTL specification is synthesized into gate-level netlist using commercial synthesis tools, such as design compiler. At the gate-level, a design is usually represented by a flattened netlist and, therefore, loses its abstraction. However, more accurate information regarding the design in terms of gates or transistors is available to designers. At gate-level, hard-to-control and hard-to-observe nets can be used to design hard-to-detect hardware Trojans [22]. Also, transition from RTL to gate-level

Table 13.1 Vulnerabilities, metrics, and rules included in DSeRC

	Vulnerability	Metric	Rule	Attack (Attacker)
RTL Level	Dangerous Don't Cares	Identify all 'X' assignments and check if 'X' can propagate to observable nodes	'X' assignments should not be propagated to observable nodes	Hardware Trojan Insertion (Insider)
	Hard-to-control & hard-to-observe signal	Statement hardness and signal observability [21]	Statement hardness (signal observability) should be lower (higher) than a threshold value	Hardware Trojan (Insider)
	Asset leakage	Structure checking and information flow tracking	YES/NO: access assets or observe assets	Asset hacking (End user)
	...	...	...	...
Gate Level	Hard-to-control & hard-to-observe net	Net controllability and observability [22]	Controllability and observability should be higher than a threshold value	Hardware Trojan (Insider)
	Vulnerable Finite State Machine (FSM)	Vulnerability factor of fault-injection (VF_{FI}) & vulnerability factor of Trojan insertion (VF_{Tro}) [7]	VF_{FI} and VF_{Tro} should be zero	Fault-injection, Hardware Trojan (Insider, end user)
	Asset leakage	Confidentiality and integrity assessment [24,25]	YES/NO: access assets or observe assets	Asset hacking (End user)
	Design-for-Test (DFT)	Confidentiality and integrity assessment [24,25]	YES/NO: access assets or observe assets	Asset hacking (End user)
	Design-for-Debug (DFD)	Confidentiality and integrity assessment [24,25]	YES/NO: access assets or observe assets	Asset hacking (End user)
	...	...	...	...
Layout Level	Side-channel signal	Side-channel vulnerability factor (SVF) [26]	The SVF should be lower than a threshold value	Side-channel attack (End user)
	Micro-probing	Exposed area of the security-critical nets which are vulnerable to micro-probing attack [23]	The exposed area should be lower than a threshold value	Micro-probing attack (Professional attacker)
	Injectable Fault/Error	Timing violation vulnerability factor (TVVF) [27]	TVVF higher than a threshold means the implementation is insecure	Timing-based fault-injection attack (End user)
	...	...	...	...

can introduce additional vulnerabilities by the CAD tools. Examples of vulnerabilities introduced by the tools have been discussed in Chapter 6, Section 6.4.2. These vulnerabilities need to be analyzed at the gate-level.

DFT and DFD structures are generally incorporated into the ICs at the gate-level. Therefore, the vulnerabilities introduced by the test and debug structure need to be analyzed at this level.

Layout level: Physical layout design is the last design stage before sending the GDSII design files to fabrication facility, so all the remaining vulnerabilities should be addressed in this level. During the layout design, the placement and routing phase gives information about the spatial arrangements of the cells and metal connections in the circuit. In the layout level, power consumption, electromagnetic emanations, and execution time can be accurately modeled. Therefore, vulnerability analysis of side-channel- and fault-injection-based attacks can be done very accurately at this level. Additionally, some vulnerability analyses, for example, vulnerability to probing attack [23], can only be done at the layout level. However, any analysis done at this level is very time-consuming compared to RTL and gate-level analyses.

13.3.1.2 Metrics and Rules

The vulnerabilities discussed so far are tied with metrics and rules, so that each design's security can be quantitatively measured (see Table 13.1). These rules and metrics of the DSeRC framework can be compared with the well-known design rule check (DRC). In DRC, semiconductor manufacturers convert manufacturing specifications into a series of metrics that enable the designer to quantitatively measure a mask's manufacturability. For the DSeRC framework, each vulnerability needs to be mathematically modeled, and the corresponding rules and metrics must be developed, so that the vulnerability of a design can be quantitatively evaluated. As for the rules, there can be two types; one type is based on quantitative metric, and the other is based on a binary classification (yes/no). A brief description of some of the rules and metrics corresponding to the vulnerabilities are shown in Table 13.1.

Asset leakage: Vulnerabilities associated with asset leakage can be unintentionally created by design-for-test (DFT) and design-for-debug (DFD) structures, CAD tools, and/or by designer's mistakes. These vulnerabilities cause violation of information security policies, that is, confidentiality and integrity policies. Therefore, the metric for identifying these vulnerabilities is confidentiality and integrity assessment. Contreras et al. [24] and Nahiyan et al. [25] presented a framework that validates whether the confidentiality and integrity policies are being upheld in the SoC. The rule for this vulnerability can be stated as follows:

Rule: An asset signal should never propagate to an observe point or be influenced by a control point that is accessible by an attacker.

Vulnerable FSM: The synthesis process of a finite state machine (FSM) can introduce additional security risks in the implemented circuit by inserting additional don't-care states and transitions. An attacker can utilize these don't-care states and transitions to facilitate fault-injection and Trojan attacks. Nahiyan et al. [7] developed two metrics—named vulnerability factor for fault-injection (VF_{FI}) and vulnerability factor for Trojan insertion (VF_{Tro})—to quantitatively analyze how susceptible an FSM is against fault-injection and Trojan attacks, respectively. The higher the values of these two metrics, the more vulnerable the FSM is to fault and Trojan attacks. The rule for this vulnerability can be stated as follows:

Rule: For an FSM design to be secured against fault-injection and Trojan insertion attacks, the values of VF_{FI} and VF_{Tro} should be zero.

Microprobing attack: Microprobing is a type of physical attack that directly probes the signal wires inside the chip in order to extract sensitive information. This attack has raised serious concerns for security-critical applications. Shi et al. [23] developed a layout-driven framework to quantitatively evaluate a post place-and-route design in terms of exposed area of the security-critical nets, which are vulnerable to microprobing attack. The larger the exposed area, the more vulnerable the net is to probing attack. Therefore, the rule for microprobing vulnerability can be stated as follows:

Rule: The exposed area to microprobing should be lower than a threshold value.

Susceptibility to Trojan insertion: Salmani et al. [21] developed a metric named "Statement Hardness" to evaluate the difficulty of executing a statement in the RTL code. Areas in HDL code with large value of Statement Hardness are more vulnerable to Trojan insertion. Therefore, the metric Statement Hardness gives the quantitative measure of a design's susceptibility to Trojan insertion. Next is to define rule(s) to evaluate if the design is secure. For this vulnerability, the rule can be stated as follows:

Rule: For a design to be secured against Trojan-insertion attack, statement hardness of each statement in the design should be lower than SH_{thr}. Here, SH_{thr} is a threshold value that needs to be derived from the area and performance budget.

At the gate-level, a design is vulnerable to Trojan insertion, which can be implemented by adding and deleting gates. To hide the effect of an inserted Trojan, an adversary targets hard-to-detect areas of the gate-level netlist. Hard-to-detect nets are defined as nets which have low transition probability and are not testable through well-known fault-testing techniques (for example, stuck-at, transition delay, path delay, and bridging faults) [22]. Inserting a Trojan in hard-to-detect areas would reduce the probability to trigger the Trojan and, thereby, reduce the probability of being detected during verification and validation testing. Tehranipoor et al. [28] developed metrics to evaluate hard-to-detect areas in the gate-level netlist.

Fault-injection and side-channel attacks: Yuce et al. [27] introduced timing violation vulnerability factor (TVVF) metric to evaluate the vulnerability of a hardware structure to setup-time violation attacks, which are one subset of fault-injection attacks. Huss et al. [29] developed a framework named AMASIVE (adaptable modular autonomous side-channel vulnerability evaluator) to automatically identify side-channel vulnerabilities of a design. Also, a metric named side-channel vulnerability factor (SVF) has been developed to evaluate an IC's vulnerability to power side-channel attacks [26].

Note that the development of these metrics is a challenging task, because ideally the metrics must be independent of attack models and targeted applications, or functionality of a design. For example, an attacker can apply voltage-starving- or clock-glitching-based fault-injection attacks to obtain the private key of AES, or RSA encryption modules. The metric for fault-injection needs to provide a quantitative measure of vulnerability for any design (AES or RSA) against any of such attacks (voltage starving or clock glitching). One strategy would be to first identify the root vulnerabilities that these attacks try to exploit. For this particular example, both voltage starving and clock glitching aim to effect a setup-time violation. The framework must, therefore, evaluate the difficulty of violating setup-time for a given design, for gaining access to the targeted security assets.

13.3.1.3 CAD Tools for Security Validation

The DSeRC framework is intended to be integrated with the conventional IC design flow, so that security evaluation can be made an inherent part of the design process. This requires the development of CAD tools, which can automatically evaluate the security of a design, based on DSeRC rules and metrics. The tools' evaluation times need to be scalable with the design size. Also, the tools should

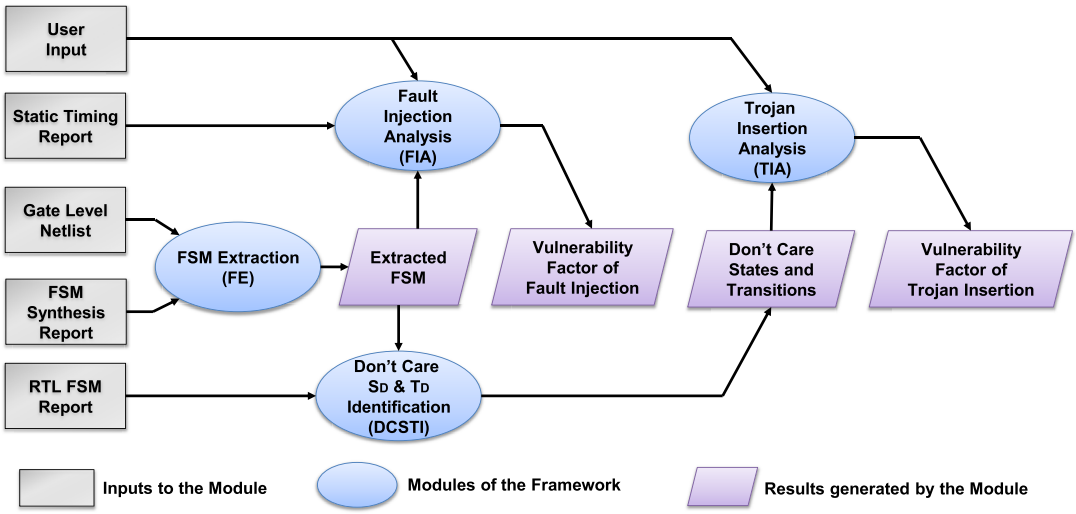

FIGURE 13.4

Overall workflow of the AVFSM framework.

be easy to use, and the outputs generated by the tools need to be understandable by the design engineer. Following is a brief description of some CAD tools, which can be incorporated in the DSeRC framework for security assessment:

CAD tool for analyzing vulnerabilities in FSM: Nahiyan et al. [7] developed a comprehensive framework, called AVFSM, to automatically analyze the vulnerabilities of FSMs against fault-injection and Trojan attacks. AVFSM takes the followings as inputs: (i) gate-level netlist of the design; (ii) FSM synthesis report; and (iii) user given inputs. The framework outputs the list of vulnerabilities found in the given FSM. Here, the user needs to specify which are the protected states and the authorized states. Protected states are those states that can compromise the security of an FSM if these states are either bypassed or accessed from any states apart from the authorized states. Authorized states are those states which are allowed to access a protected state.

The overall workflow of the AVFSM framework is shown in Fig. 13.4. The AVFSM framework is composed of four modules:

- FSM Extraction (FE): To analyze various vulnerabilities in a given FSM, one first needs to extract the state transition graph (STG) from the synthesized gate-level netlist. An automatic-test-pattern-generation-based (ATPG-based) FSM extraction technique can be developed to produce the STG with the don't-care states and transitions from the synthesized netlist. This FSM extraction technique takes the gate-level netlist and the FSM synthesis report as inputs, and automatically generates the STG. The detailed algorithm for this technique can be found in [7].
- Don't-care states and transitions identification (DCSTI): It reports the don't-care states and transitions introduced by the synthesis process of the given FSM. The don't-care states and transitions can create vulnerabilities in the FSM by allowing a protected state to be illegally accessed through

the don't-care states and transitions. These don't-care states, which can access the protected states are defined as Dangerous Don't-Care States (DDCS).

- Fault-injection analysis (FIA): Module FIA uses the vulnerability factor for fault-injection (VF_{FI}) metric to measure the overall vulnerability of the FSM to fault-injection attack. VF_{FI} is defined as follows:

$$VF_{\text{FI}} = \{PVT(\%), ASF\} \tag{13.1}$$

  The metric VF_{FI} is composed of two parameters {PVT(%), ASF}. PVT(%) indicates the percentage of vulnerable transitions. Vulnerable transition is defined as a set of transitions during which a fault can be injected to gain access to a protected state. ASF provides a probabilistic measure of successful fault-injection attacks during the vulnerable transitions. The detailed algorithm to calculate these metrics is provided in [7]. The greater the values of these two parameters are, the more susceptible the FSM is to fault attacks. A VF_{FI} of (0,0) means that a protected state cannot be accessed by an unauthorized state, and the respective FSM is not vulnerable to fault attacks.
- Trojan-insertion analysis (TIA): Module TIA uses the vulnerability factor for Trojan insertion (VF_{Tro}) metric to evaluate the vulnerability of the FSM to Trojan insertion as follows:

$$VF_{\text{Tro}} = \frac{Total\ number\ of\ s'}{Total_{\text{Transition}}}, \tag{13.2}$$

  where $s' \in DDCS$. When a don't-care state that has direct access to a protected state is introduced, it can create a vulnerability in the FSM by allowing the attacker to utilize this don't-care state to insert a Trojan to facilitate access to the protected state. These states are represented as DDCS. For a secure design, this metric's value should be zero.

Information flow tracking: Contreras et al. [24] and Nahiyan et al. [25] developed an information flow tracking (IFT) framework that detects the violation of confidentiality and integrity policies. This framework is based on modeling an asset (for example, a net carrying a secret) as stuck-at-0 and stuck-at-1 faults, and it leverages the automatic test pattern generation (ATPG) algorithm to detect those faults. A successful detection of faults means that the logical value of the asset-carrying net can be observed through the observe points, or logical value of the asset can be controlled by the control points. In other words, there exists information flow from the asset to observe points, or from control points to the asset. Here, the observe points refer to any primary or pseudo-primary (scan FFs) outputs that can be used to observe internal signals. On the other hand, the control points refer to the primary or pseudoprimary (scan FFs) inputs that can be used to control internal circuit signals.

Figure 13.5 shows the overall flow of the IFT framework and its four main steps: initialize, analysis, propagation, and recursive.

(i) Initialize: This first step takes the name of the asset nets to which IFT will be applied, the gate-level netlist of the design, and the technology library (required for ATPG analysis) as inputs. Then, the framework adds scan capability to all the registers/flip-flops (FFs) in the design to make them controllable and observable. Here, the "What If" analysis feature is used to virtually add and/or remove FFs from scan chain. This feature allows performing partial-scan analysis dynamically, without requiring to re-synthesize the netlist. Also, masks are applied to all FFs, so that asset propagation to each FF can be independently tracked. Applying mask is an important step as it allows controlling fault propagation to one FF at a time.

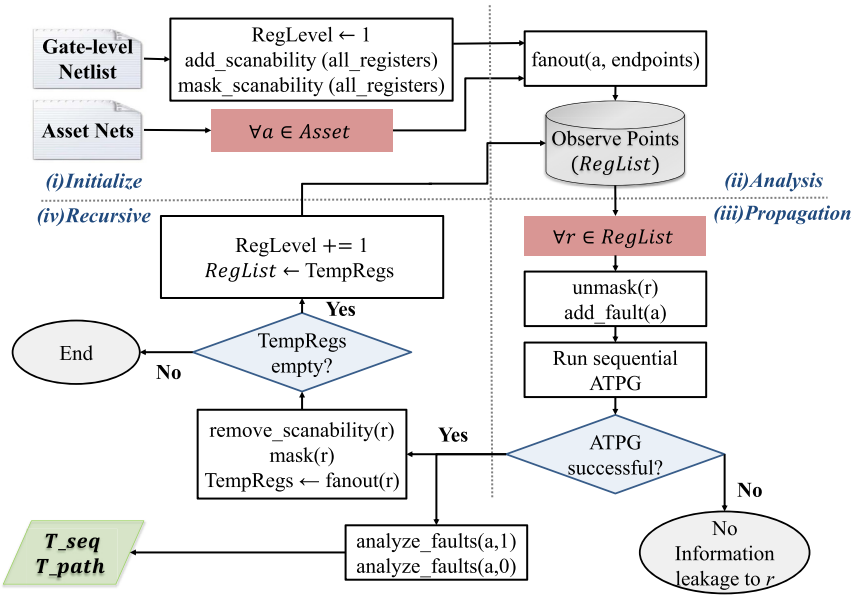

FIGURE 13.5

Information flow tracking (IFT) framework to identify the observe points, where an asset bit propagates to.

(ii) Analysis: This step utilizes fanout analysis to identify which FFs are located in fanout of a particular asset bit. For each asset bit $a \in asset$ (shown in Fig. 13.5), the asset analysis step finds the FFs that are in the fanout cone of a.

(iii) Propagation: This step analyzes the propagation of each asset bit a to each individual FF. To perform a comprehensive analysis of potential points of the asset bit propagation, each FF must be analyzed separately. For each $r \in RegList$ (shown in Fig. 13.5), the applied mask is removed, so the key- bit propagation to r can be tracked. The next step adds the key bit a as the only stuck-at fault in the design, and runs ATPG algorithm in the sequential mode to find paths to propagate $a = 0$, and $a = 1$ to FF r. If both, $a = 0$ and $a = 1$ can be detected from r, then there exists an information flow from a to r and the algorithm marks r as an observe point. The asset-propagation step also stores the propagation path (T_{path}) and the control sequence (T_{seq}) required for the asset-bit propagation for further analysis. Note that, T_{seq} contains the list of input ports and control registers, which controls the information propagation from a to r.

(iv) Recursive: This step leverages the partial-scan technique along with sequential ATPG to find propagation paths through all sequential levels until the output, or the last-level FFs, are reached. Here, the function remove_scanability (shown in Fig. 13.5) makes the ATPG tool treat r as a nonscan FF for simulation purposes, without redoing scan insertion. The FF's output ports Q and QN are used to get a new fanout emerging from r to the next level of registers. To find information flow through multiple levels of registers, the scanability of all identified registers in RegList is removed incrementally, and

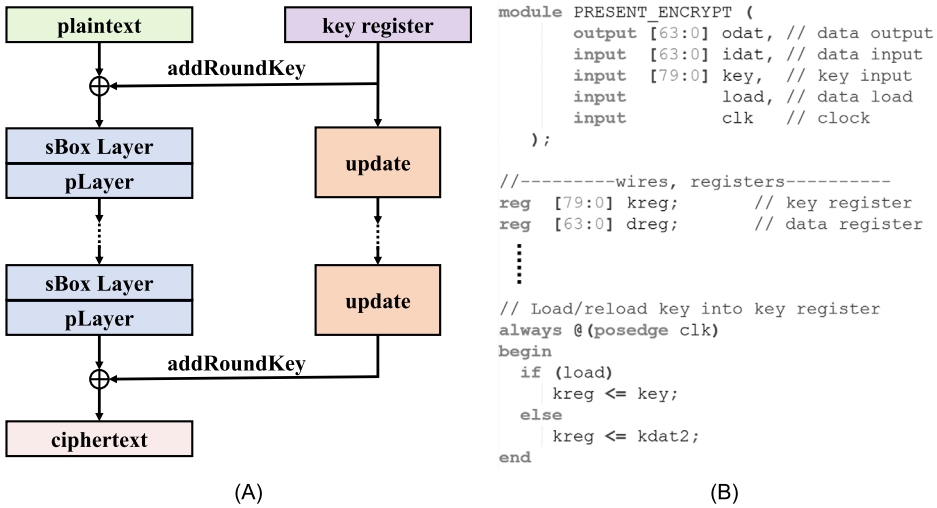

FIGURE 13.6

Unintentional vulnerabilities created by design mistakes. (A) Top-level description of PRESENT, (B) Verilog implementation of PRESENT.

sequential ATPG is used to create propagation paths from asset bit a to subsequent-level registers. This process continues until the last level of registers.

The output of the IFT framework is a list of observe points (registers/FFs), where the asset bit propagates to, and the propagation path (T_{path}), along with the stimulus vector (T_{seq}) for asset propagation for each FF, r.

13.3.2 WORKFLOW OF DSeRC FRAMEWORK

This section describes how the rules and metrics are used to identify a vulnerability under DSeRC framework. Table 13.1 shows the list of vulnerabilities and their corresponding metrics and rules covered by the DSeRC framework. An example of hardware implementation of PRESENT encryption algorithm is used to illustrate the workflow of DSeRC framework. Figure 13.6A shows the top-level description of PRESENT encryption algorithm [30]. A segment of its Verilog implementation is shown in Fig. 13.6B [31]. One can see that the key is directly being assigned to the register, defined as "kreg" in the module. Although the encryption algorithm itself is secure, a vulnerability is unintentionally created in its hardware implementation. When this design is implemented, the "kreg" register will be included in the scan-chain, and an attacker can gain access to key through scan-chain-based attack [4].

The designer first gives the RTL design files and the name of the asset (the "key") as input to DSeRC framework. DSeRC framework uses the information flow tracking to analyze if the key can be leaked through any observation points, for example, registers. As for this design, the key can be observed through the kreg register. Therefore, the framework gives a preemptive warning to the designer that if the register kreg is included in the DFT structure then, the key can be leaked through scan-chain. It will

be up to the designer to apply a countermeasure to address this vulnerability before moving to next level of abstraction. One possible countermeasure would be to exclude kreg from the scan-chain.

After addressing this vulnerability, the design is synthesized and DFT is inserted. The designer then gives the synthesized gate-level netlist to DSeRC and the framework uses the confidentiality assessment [24,25] technique to analyze if the key can be leaked through any observable point. If the key is secured from leaking, then the DSeRC rule is satisfied and the design is considered to be secured against asset leakage. On the other hand, if the DSeRC framework identifies that the key is still being leaked to scan flip-flops (observable points), then the framework raises a flag and points the scan flip-flops carrying the information about the key. The designer needs to address this vulnerability before moving to physical layout stage. One possible approach would be to apply secure scan structure [4] to counter the vulnerability introduced by the DFT structure.

Note that manually tracking the asset in a SoC to evaluate whether it is being leaked through an observable point is an extremely difficult, if not impossible, task for the designer. DSeRC framework can pinpoint the asset leakage paths, allowing the designers to concentrate the analysis effort on those paths and, therefore, make informed decision.

While DSeRC may be a logical and necessary step in designing secure ICs, it would not necessarily eliminate the need for security subject matter experts. The DSeRC framework is intended to be an automated framework and, therefore, may not take the application of an IC into consideration. For example, the Trojan observability metric reports the difficulty of observing each signal in the design. For Trojan detection, a high observability metric is desired. However, for an encryption module, a high observability metric for the private key poses a serious threat. Therefore, it should be on the designer to interpret the results generated by the DSeRC framework.

13.4 POST-SILICON SECURITY AND TRUST ASSESSMENT FOR ICs

This section presents some of the commonly used post-silicon validation techniques, for example, fuzzing, negative testing, and white-box hacking. These activities inherently depend on human creativity, where tools and infrastructures primarily act as assistants, filling up gaps in human reasoning and providing recommendations.

13.4.1 FUZZING

Fuzzing, or fuzz testing [32], is a testing technique that involves providing invalid, unexpected, or random inputs for hardware or software and monitoring the result for exceptions, such as crashes, failing built-in code assertions, or memory leaks. It was developed as a software testing approach and has since been adapted to hardware/software systems. In the context of security, it is effective for exposing a number of potential attacker entry points, including through buffer or integer overflows, unhandled exceptions, race conditions, access violations, and denial of service. Traditionally, fuzzing uses either random inputs or random mutations of valid inputs. A key attraction to this approach is its high automation compared to other validation technologies, such as penetration testing and formal analysis. Nevertheless, since it relies on randomness, fuzzing may miss security violations that rely on unique corner-case scenarios. To address that deficiency, there has been recent work on “smart” input generation for fuzzing, based on domain-specific knowledge of the target system [33]. Smart fuzzing

may provide a greater coverage of security attack entry points, at the cost of more upfront investment in design understanding.

13.4.2 NEGATIVE TESTING

Negative testing looks beyond the functional specification to identify if security objectives are underspecified, or can be subverted. As an example, in case of direct memory attack (DMA), negative testing may extend the deterministic security requirement (that is, abortion of DMA-access for the protected memory ranges) to identify if there are any other paths to the protected memory, in addition to the address translation activated by a DMA access request, and potential input stimulus to activate such paths.

13.4.3 HACKATHONS

Hackathons, also referred to as white-box hacking, fall in the "black magic" end of the security validation spectrum. The idea is for expert hackers to perform goal-oriented attempts at breaking security objectives. This activity depends primarily on human creativity, although some guidelines exist on how to approach them (see discussion on penetration testing in the next section). Because of their cost and the need for high human expertise, such approaches are performed when attacking complex security objectives, typically at hardware/firmware/software interfaces.

13.4.4 PENETRATION TESTING

A penetration test, or intrusion test, is an attack on a system with the intention to find security weakness. It is often performed by expert hackers with deep knowledge of the system architecture, design, and implementation. Roughly, the penetration testing involves iterative applications of the following three phases: attack surface enumeration, vulnerability exploitation, and result analysis.

13.4.4.1 Attack Surface Enumeration

The first task is to identify the features or aspects of the system that are vulnerable to attacks. This is typically a creative process involving a number of activities, including documentation review, network service scanning, and even fuzzing, or random testing.

13.4.4.2 Vulnerability Exploitation

Once the potential attacker entry points are discovered, applicable attacks and exploits are attempted against target areas. This may require research into known vulnerabilities, looking up applicable vulnerability class attacks, engaging in vulnerability research specific to the target, and writing/creating the necessary exploits.

13.4.4.3 Result Analysis

In this phase, the resulting state of the target after a successful attack is compared against security objectives and policy definitions to determine whether the system is indeed compromised. Note that even if a security objective is not directly compromised, a successful attack may identify additional attack surface, which must then be accounted for with further penetration testing.

While there are commonalities between penetration testing and testing for functional validation, there are important differences. In particular, the goal of functional testing is to simulate benign user behavior and (perhaps) accidental failures under normal environmental conditions of design operation, as defined by its specification. On the other hand, the penetration testing goes outside the specification or the limits set by the security objective, and simulates deliberate attacker behavior.

The efficacy of penetration testing critically depends on the ability to identify the attack surface in the first phase previously discussed. Unfortunately, rigorous methodologies for achieving this are lacking. Following are some of the typical activities in current industrial practice to identify attacks and vulnerabilities. They are classified as "easy," "medium," and "hard", depending on the creativity necessary. Note that there are tools to assist the individual in many of the activities below [34,35]. However, determining the relevance of the activity, identifying the degree to which each activity should be explored, and inferring a potential attack from the result of the activity involve significant creativity.

- Easy approaches: These include review of available documentation (for example, specification and architectural materials), known vulnerabilities or misconfigurations of IPs, software, or integration tools, missing patches, and use of obsolete or out-of-date software versions.
- Medium-complexity approaches: These include inferring potential vulnerabilities in the target of interest from information about misconfigurations, vulnerabilities, and attacks in related or analogous products, for example, a competitor product and a previous software version. Other activities of similar complexity involve executing relevant public security tools, or published attack scenarios against the target.
- Hard approaches: These include full security evaluation of any utilized third-party components, integration testing of the whole platform, and identification of vulnerabilities involving communications among multiple IPs, or design components. Finally, the vulnerability research involves identifying new classes of vulnerabilities for the target, which have never been seen before. The latter is particularly relevant for new IPs, or SoC designs for completely new market segments.

13.4.5 FUNCTIONAL VALIDATION OF SECURITY-SENSITIVE DESIGN FEATURES

This is essentially an extension to functional validation but it pertains to design elements involved in critical security feature implementations. An example is the cryptographic engine IP. A critical functional requirement for the cryptographic engine is that it encrypts and decrypts data correctly for all modes. As with any other design block, the cryptographic engine is also a target of functional validation. However, given that it is a critical component of a number of security-critical design features, cryptographic functionality may be crucial enough to justify further validation, beyond the coverage provided by functional validation activities. Consequently, such an IP may undergo more rigorous testing, or even formal analysis. Other such critical IPs may include IPs involved in secure boot, and in-field firmware patching.

13.4.6 VALIDATION OF DETERMINISTIC SECURITY REQUIREMENTS

Deterministic security requirements are validation objectives that can be directly derived from security policies. They include access control restrictions and address translations. Let us consider an access control restriction that specifies a certain range of memory to be protected from DMA access. This may be done to ensure protection against code-injection attacks, or protect a key that is stored in

such location. An obvious derived validation objective is to ensure that all DMA calls for access to a protected memory must be aborted. Note that validation of such properties may not be included in the functional validation, since DMA access requests for DMA-protected addresses are unlikely to arise for "normal" test cases, or usage scenarios.

The following sections discuss some countermeasures, which can be adopted at design stage to address the security issues in the hardware design.

13.5 DESIGN FOR SECURITY

This section presents design strategies that make the hardware design inherently resilient to different security issues discussed earlier.

13.5.1 SECURITY ARCHITECTURE

The typical approach for developing a baseline secure architecture depends on the following two steps:

- Use threat modeling to identify potential threats to the current architecture definition.
- Refine the architecture with mitigation strategies covering the threats identified.

The baseline architecture is typically derived from legacy architectures for previous products, adapted to account for the policies defined for the system under exploration. In particular, for each asset, the architect must identify: 1) who can access the asset; 2) what kind of access is permitted by the policies; and 3) at what points in the system execution or product development lifecycle such access requests can be granted or denied. The process can be complex and tedious for several reasons. A SoC design may have a significant number of assets, often in the order of thousands, if not more. Furthermore, not all assets are statically defined; many assets are created at different IPs during the system execution. For example, a fuse or an e-wallet may have a statically defined asset, such as key configuration modes. During system execution, these modes are passed to the cryptographic engine, which generates the cryptographic keys for different IPs, and transmits them through the system network-on-chip (NoC) to the respective IPs. Each participant in this process has sensitive assets (either static or created) during different phases of the system execution. The security architecture must account for any potential access to these assets at any point of execution, possibly under the relevant adversary model.

There has been a significant amount of work toward standardizing architecture to implement access control for different assets. Most of the relevant work has taken the form of developing a trusted execution environment (TEE), viz., a mechanism for guaranteeing isolation between code and sensitive data at different points of the system execution. TEEs, of course, have been a part of computer security for a long time. One of the most common TEE architectures is the trusted platform module (TPM), which is an international standard for a secure cryptoprocessor. It is designed to secure the hardware by integrating cryptographic keys into devices [36]. It covers methods to securely generate cryptographic keys and limit their use, random number generator requirements, and capabilities, such as remote attestation, and sealed storage. In addition to TPM, there has been significant work on architecting other TEEs, both in the industrial platform and in academic research [37,38]. Below, three TEE frameworks specifically developed for SoC designs are presented: Samsung KNOX [39], Intel

Software Guard Extension (SGX) [40], and ARM TrustZone [41]. Note that in spite of differences motivated by the isolation and separation targets, the underlying architectural plans for these TEEs are similar, particularly as it relates to the combination of hardware support (for example, secure operating modes and virtualization), and software mechanisms (such as, context switch agents and integrity check).

13.5.1.1 Samsung KNOX

This architecture is specifically targeted toward smartphones, and provides secure separation features to enable information partition between business and personal content to coexist on the same system. In particular, it permits hot swap between these two content worlds, for example without requiring system restart. The key ingredient of this technology is a separation kernel that implements the information isolation. This architecture permits several system-level services, including the following:

- Trusted boot, that is, preventing unauthorized OS and software from being loaded onto the device at startup.
- Trust-zone-based integrity measurement architecture (TIMA), which continually monitors kernel integrity.
- Security enhancement (SE) for Android, an enforcement mechanism providing protection of system/user data based on confidentiality and integrity requirements through separation.
- KNOX container, which offers a secure environment in which protected business applications can run with guaranteed information separation from the rest of the device.

13.5.1.2 ARM TrustZone

TrustZone technology is a system-wide approach to provide security on high-performance computing platforms. The TrustZone implementation relies on partitioning the SoC's hardware and software resources, so that they exist in two worlds: secure and nonsecure. The hardware supports access control and permissions for the handling of secure/nonsecure applications, and the interaction and communication among them. The software supports secure system calls and interrupts for secure runtime execution in a multitasking environment. These two aspects ensure that no secure world resources can be accessed by the nonsecure world components, except through secure channels, enabling an effective wall-of-security to be built between the two domains. This protection extends to input/output (I/O), connected to the system bus via the TrustZone enabled AMBA3 AXI bus fabric, which also manages memory compartmentalization.

13.5.1.3 Intel SGX

SGX is an architecture for providing a trusted execution environment provided by the underlying hardware to protect sensitive application and user programs or data against potentially malicious, or tampered operating systems. SGX permits applications to initiate secure enclaves or containers, which serve as so-called "islands of trust". It is implemented as a set of new CPU instructions that can be used by applications to set aside such secure enclaves of code and data. This enables 1) applications to preserve the confidentiality and integrity of sensitive data without disrupting the ability of legitimate system software to manage the platform resources; and 2) end users to retain control of their platforms, applications, and services even in the presence of malicious system software.

The TEEs provide a foundation (that is, a mechanism of isolation) for implementing security policies. However, they are a far cry from a standardized approach for implementing policies themselves.

To provide such approaches, it is necessary to 1) develop a language for succinctly and formally expressing security policies; 2) generate a parameterized "skeleton" design that can be easily instantiated to diverse policy implementations; and 3) develop techniques for synthesizing policy implementation from high-level descriptions. Recent academic and industrial research has attempted to address some of these issues. [42] provides a language and synthesis framework for certain security policies. [43] provides a microcontroller-based flexible framework for implementing diverse security policies. There have been optimized architectural support for specific classes of policies, for example, control-flow integrity [44], and Trojan resistance [45].

13.5.2 SECURITY POLICY ENFORCER

This module is responsible for enforcing security policies that are imperative for ensuring security at the hardware level. The readers are referred to Chapter 16 of this book for more details.

13.5.3 SIDE-CHANNEL RESISTANT DESIGN

Different countermeasure techniques have been proposed to counter the power and electromagnetic side-channel attack (details of these attacks have been discussed in Chapter 8). These countermeasure can be broadly categorized as hiding mechanism and masking mechanism. A brief description of these countermeasures are discussed in the following section.

13.5.3.1 Hiding Mechanism

Hiding mechanisms attempt to eliminate the relationship between the leaked information and the secret data, that is, make the leaked information uncorrelated to the secret data. A side-channel attack typically depends on the signal-to-noise ratio (SNR) which is defined as follows:

$$SNR = \frac{var(signal)}{var(noise)}. \tag{13.3}$$

Here, the signal refers to the leaked power signal, which is correlated to the secret data and exploited by the adversaries to perform the side-channel attack. The noise refers to the power signal, which has no correlation to the secret data. The hiding mechanisms decrease the SNR either by increasing the noise or by decreasing the signal to counter the side-channel attack. These mechanisms mainly utilize randomization and equalization techniques to decrease the SNR.

Randomization: These techniques attempt to increase the noise of the circuit to reduce the SNR by constantly changing the execution order, or by generating noise directly [49]. One possible approach for applying randomization is by injecting white noise on the channel (additive white Gaussian noise) [50]. The main idea here is to incorporate noise generation sources in the design, which randomly modifies the current on the power line.

Equalization: These techniques attempt to decrease the leaked power signal, which is correlated to the secret data and reduce the SNR. The main idea here is to make equal power consumption for all operations related to processing secret data. The equalization technique can be realized by a specific type of logic style, which consumes a constant amount of power. Examples of this kind of logic style include dual rail [51] and differential logic [52]. Typically, the output bit transition of a CMOS logic gate depends on the input vectors. For example, a transition from $0 \rightarrow 1$ on the output of an OR gate

indicates that either one or two input of the OR gate has a transition from $0 \rightarrow 1$. In other words, the power consumption, which depends on the logic transition of CMOS gates becomes a function of input vectors, and this power consumption is exploited by the adversary for side-channel attack. A solution would be to make the logic circuit consume the same amount of power for every kind of bit transition (i.e. $0 \rightarrow 0$, $0 \rightarrow 1$, $1 \rightarrow 0$, $1 \rightarrow 1$). The dual rail and the differential logic achieve this property by precharging the output in the first half of every clock cycle, and evaluating the correct output value in the second half [49].

13.5.3.2 Masking Mechanism

These mechanisms attempt to randomize the intermediate values (function of the sensitive information) of a cryptographic operation to break the dependencies between these values and the power consumption [50]. Unlike the hiding mechanisms, masking mechanisms are applied at the algorithmic level and can be implemented by standard CMOS logic gates. The main idea here is to conceal each intermediate value by a random mask that is different for every execution. It ensures that the sensitive data is masked with a random value, which eliminates dependencies between the intermediate values and the power consumption [53]. Masking technique can be implemented by Boolean secret sharing technique, which is discussed below.

Let us consider that X and K denote two intermediate values associated with the plaintext and the sub-key of a cryptographic operation. Also, let us consider that another variable Z which is expressed as $Z = X \oplus K$. Z variable is a function of the intermediate value K and any operation on Z variable could leak some information about K. Boolean masking technique attempts to secure the operation of Z by randomly splitting into two shares M_0 and M_1, expressed by the following equation:

$$Z = M_0 \oplus M_1. \tag{13.4}$$

M_1 is referred to as the mask, and M_0 is referred to as the masked variable. M_0 is derived such that $M_0 = Z \oplus M_1$. Any operation on Z leads to the processing of two new shares M_0' and M_1' such that

$$S(Z) = M_0' \oplus M_1', \tag{13.5}$$

where M_1' share is usually generated at random, and M_0' share is derived as $M_0' = S(Z) \oplus M_1'$. The main challenge here is to deduce M_0' from M_0 and M_1 and M_1' without compromising the security of the scheme. When S is linear function, deducing M_0' is relatively an easy task as compared to when S is nonlinear. The detailed derivation of these function is discussed in [53].

13.5.4 PREVENT TROJAN INSERTION

These techniques consist of preventive mechanisms that attempt to thwart hardware Trojan insertion by attackers. The readers are referred to Chapter 5 of this book for more details.

13.6 EXERCISES

13.6.1 TRUE/FALSE QUESTIONS

1. Random number can be an asset.

2. Remote attackers cannot perform attacks exploiting scan structure.
3. Injecting faults to modify SRAM contents is a semi-invasive attack.
4. Side-channel attacks fall into invasive attack category.
5. Vulnerabilities associated with don't-care states are introduced in the RTL stage.
6. Vulnerabilities associated with DFT structure are introduced in the gate-level stage.

13.6.2 LONG-ANSWER TYPE QUESTIONS

1. Describe the "Assets" that can be found in a SoC.
2. How can the entropy asset be exploited?
3. How can a scan-based attack be performed remotely? Show an example.
4. Explain the differences between semi-invasive and invasive attack.
5. Describe the capabilities of an "Insider" attacker. What kind of attacks can an "Insider" perform?
6. How potential vulnerabilities can be introduced by design mistakes? Show an example.
7. How potential vulnerabilities can be introduced by CAD tools? Show an example.
8. Describe the principle of the following tests: (i) Fuzzing, (ii) Negative Testing, (iii) Penetration Testing.
9. How does the "Hiding Mechanisms" protect against side-channel attacks?

REFERENCES

[1] P.C. Kocher, Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems, in: Annual International Cryptology Conference, Springer, pp. 104–113.
[2] P. Kocher, J. Jaffe, B. Jun, Differential power analysis, in: Annual International Cryptology Conference, Springer, pp. 388–397.
[3] D. Hely, M.-L. Flottes, F. Bancel, B. Rouzeyre, N. Berard, M. Renovell, Scan design and secure chip, in: IOLTS, vol. 4, pp. 219–224.
[4] J. Lee, M. Tehranipoor, C. Patel, J. Plusquellic, Securing scan design using lock and key technique, in: Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on, IEEE, pp. 51–62.
[5] E. Biham, A. Shamir, Differential fault analysis of secret key cryptosystems, in: Annual International Cryptology Conference, Springer, pp. 513–525.
[6] C. Dunbar, G. Qu, Designing trusted embedded systems from finite state machines, ACM Transactions on Embedded Computing Systems (TECS) 13 (2014) 153.
[7] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, M. Tehranipoor, AVFSM: a framework for identifying and mitigating vulnerabilities in FSMs, in: Design Automation Conference (DAC), 2016 53rd ACM/EDAC/IEEE, IEEE, pp. 1–6.
[8] M. Tehranipoor, F. Koushanfar, A survey of hardware Trojan taxonomy and detection, IEEE Design & Test of Computers 27 (2010).
[9] K. Xiao, A. Nahiyan, M. Tehranipoor, Security rule checking in IC design, Computer 49 (2016) 54–61.
[10] A. Nahiyan, K. Xiao, D. Forte, M. Tehranipoor, Security rule check, in: Hardware IP Security and Trust, Springer, 2017, pp. 17–36.
[11] ARM Holdings, Building a secure system using trustzone technology, https://developer.arm.com/docs/genc009492/latest/trustzone-software-architecture/the-trustzone-api. (Accessed August 2018), [Online].
[12] E. Peeters, SoC security architecture: current practices and emerging needs, in: Proceedings of the 52nd Annual Design Automation Conference, ACM, p. 144.
[13] T. Korak, T. Plos, Applying remote side-channel analysis attacks on a security-enabled NFC tag, in: Cryptographers' Track at the RSA Conference, Springer, pp. 207–222.
[14] A. Das, J. Da Rolt, S. Ghosh, S. Seys, S. Dupuis, G. Di Natale, M.-L. Flottes, B. Rouzeyre, I. Verbauwhede, Secure JTAG implementation using Schnorr protocol, Journal of Electronic Testing 29 (2013) 193–209.

[15] P. Mishra, S. Bhunia, M. Tehranipoor, Hardware IP Security and Trust, Springer, 2017.
[16] S. Chen, J. Xu, Z. Kalbarczyk, K. Iyer, Security vulnerabilities: from analysis to detection and masking techniques, Proceedings of the IEEE 94 (2006) 407–418.
[17] S.P. Skorobogatov, Semi-invasive attacks: a new approach to hardware security analysis, Ph.D. thesis, University of Cambridge, Computer Laboratory, 2005.
[18] M. Tehranipoor, C. Wang, Introduction to Hardware Security and Trust, Springer Science & Business Media, 2011.
[19] M.A. Harris, K.P. Patten, Mobile device security considerations for small- and medium-sized enterprise business mobility, Information Management & Computer Security 22 (2014) 97–114.
[20] N. Fern, S. Kulkarni, K.-T.T. Cheng, Hardware Trojans hidden in RTL don't cares—automated insertion and prevention methodologies, in: Test Conference (ITC), 2015 IEEE International, IEEE, pp. 1–8.
[21] H. Salmani, M. Tehranipoor, Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level, in: Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2013 IEEE International Symposium on, IEEE, pp. 190–195.
[22] H. Salmani, M. Tehranipoor, R. Karri, On design vulnerability analysis and trust benchmarks development, in: Computer Design (ICCD), 2013 IEEE 31st International Conference on, IEEE, pp. 471–474.
[23] Q. Shi, N. Asadizanjani, D. Forte, M.M. Tehranipoor, A layout-driven framework to assess vulnerability of ICs to microprobing attacks, in: Hardware Oriented Security and Trust (HOST), 2016 IEEE International Symposium on, IEEE, pp. 155–160.
[24] G.K. Contreras, A. Nahiyan, S. Bhunia, D. Forte, M. Tehranipoor, Security vulnerability analysis of design-for-test exploits for asset protection in SoCs, in: Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific, IEEE, pp. 617–622.
[25] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, M. Tehranipoor, Hardware Trojan detection through information flow security verification, in: Test Conference (ITC), 2017 IEEE International, IEEE, pp. 1–10.
[26] J. Demme, R. Martin, A. Waksman, S. Sethumadhavan, Side-channel vulnerability factor: a metric for measuring information leakage, ACM SIGARCH Computer Architecture News 40 (2012) 106–117.
[27] B. Yuce, N.F. Ghalaty, P. Schaumont, TVVF: estimating the vulnerability of hardware cryptosystems against timing violation attacks, in: Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on, IEEE, pp. 72–77.
[28] M. Tehranipoor, H. Salmani, X. Zhang, Integrated Circuit Authentication: Hardware Trojans and Counterfeit Detection, Springer Science & Business Media, 2013.
[29] S.A. Huss, M. Stöttinger, M. Zohner, AMASIVE: an adaptable and modular autonomous side-channel vulnerability evaluation framework, in: Number Theory and Cryptography, Springer, 2013, pp. 151–165.
[30] A. Bogdanov, L.R. Knudsen, G. Leander, C. Paar, A. Poschmann, M.J. Robshaw, Y. Seurin, C. Vikkelsoe, Present: An ultra-lightweight block cipher, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pp. 450–466.
[31] OpenCores, http://opencores.org. (Accessed August 2018).
[32] A. Takanen, J.D. Demott, C. Miller, Fuzzing for Software Security Testing and Quality Assurance, Artech House, 2008.
[33] S. Bhunia, S. Ray, S. Sur-Kolay, Fundamentals of IP and SoC Security: Design, Verification, and Debug, Springer, 2017.
[34] Microsoft Corporation, Microsoft free security tools—Microsoft baseline security analyzer, https://blogs.microsoft.com/cybertrust/2012/10/22/microsoft-free-security-tools-microsoftbaseline-security-analyzer/. (Accessed August 2018), [Online].
[35] Flexera, http://secunia.com. (Accessed August 2018).
[36] Trusted Computing Group, Trusted platform module specification, http://www.trustedcomputinggroup.org/tpm-main-specification/. (Accessed August 2018), [Online].
[37] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, J.M. McCune, Trustworthy execution on mobile devices: what security properties can my mobile platform give me? in: International Conference on Trust and Trustworthy Computing, Springer, pp. 159–178.
[38] J.M. McCune, B.J. Parno, A. Perrig, M.K. Reiter, H. Isozaki, Flicker: an execution infrastructure for TCB minimization, in: ACM SIGOPS Operating Systems Review, vol. 42, ACM, pp. 315–328.
[39] Samsung, Samsung knox, http://www.samsungknox.com. (Accessed August 2018).
[40] Intel, Intel software guard extensions programming reference, https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf. (Accessed August 2018), [Online].
[41] ARM Holdings, Products Security, https://www.arm.com/products/silicon-ip-security. (Accessed August 2018), [Online].

[42] X. Li, V. Kashyap, J.K. Oberg, M. Tiwari, V.R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, F.T. Chong, Sapper: a language for hardware-level security policy enforcement, ACM SIGARCH Computer Architecture News 42 (2014) 97–112.
[43] A. Basak, S. Bhunia, S. Ray, A flexible architecture for systematic implementation of SoC security policies, in: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, IEEE Press, pp. 536–543.
[44] L. Davi, M. Hanreich, D. Paul, A.-R. Sadeghi, P. Koeberl, D. Sullivan, O. Arias, Y. Jin, Hafix: hardware-assisted flow integrity extension, in: Proceedings of the 52nd Annual Design Automation Conference, ACM, p. 74.
[45] L. Changlong, Z. Yiqiang, S. Yafeng, G. Xingbo, A system-on-chip bus architecture for hardware Trojan protection in security chips, in: Electron Devices and Solid-State Circuits (EDSSC), 2011 International Conference of, IEEE, pp. 1–2.
[46] 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.
[47] IEEE, IEEE standard test access port and boundary scan architecture, IEEE Standards 11491, 2001.
[48] 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.
[49] E. Peeters, Side-channel cryptanalysis: a brief survey, in: Advanced DPA Theory and Practice, Springer, 2013, pp. 11–19.
[50] A. Moradi, Masking as a side-channel countermeasure in hardware, ISCISC 2016 Tutorial, 2006.
[51] D. May, H.L. Muller, N.P. Smart, Random register renaming to foil DPA, in: International Workshop on Cryptographic Hardware and Embedded Systems, Springer, pp. 28–38.
[52] F. Macé, F.-X. Standaert, I. Hassoune, J.-D. Legat, J.-J. Quisquater, et al., A dynamic current mode logic to counteract power analysis attacks, in: Proc. 19th International Conference on Design of Circuits and Integrated Systems (DCIS), pp. 186–191.
[53] H. Maghrebi, E. Prouff, S. Guilley, J.-L. Danger, A first-order leak-free masking countermeasure, in: Cryptographers' Track at the RSA Conference, Springer, pp. 156–170.
[54] J.A. Roy, F. Koushanfar, I.L. Markov, Ending piracy of integrated circuits, Computer 43 (2010) 30–38.
[55] R.S. Chakraborty, S. Bhunia, Security against hardware Trojan through a novel application of design obfuscation, in: Proceedings of the 2009 International Conference on Computer-Aided Design, ACM, pp. 113–116.
[56] A. Baumgarten, A. Tyagi, J. Zambreno, Preventing IC piracy using reconfigurable logic barriers, IEEE Design & Test of Computers 27 (2010).
[57] J.B. Wendt, M. Potkonjak, Hardware obfuscation using PUF-based logic, in: Proceedings of the 2014 IEEE/ACM International Conference on Computer-Aided Design, IEEE Press, pp. 270–277.
[58] J. Rajendran, M. Sam, O. Sinanoglu, R. Karri, Security analysis of integrated circuit camouflaging, in: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, ACM, pp. 709–720.
[59] R.P. Cocchi, J.P. Baukus, L.W. Chow, B.J. Wang, Circuit camouflage integration for hardware IP protection, in: Proceedings of the 51st Annual Design Automation Conference, ACM, pp. 1–5.
[60] Y. Bi, P.-E. Gaillardon, X.S. Hu, M. Niemier, J.-S. Yuan, Y. Jin, Leveraging emerging technology for hardware security-case study on silicon nanowire FETs and graphene SymFETs, in: Test Symposium (ATS), 2014 IEEE 23rd Asian, IEEE, pp. 342–347.
[61] K. Xiao, M. Tehranipoor, BISA: built-in self-authentication for preventing hardware Trojan insertion, in: Hardware-Oriented Security and Trust (HOST), 2013 IEEE International Symposium on, IEEE, pp. 45–50.
[62] D. McIntyre, F. Wolff, C. Papachristou, S. Bhunia, Trustworthy computing in a multi-core system using distributed scheduling, in: On-Line Testing Symposium (IOLTS), 2010 IEEE 16th International, IEEE, pp. 211–213.
[63] C. Liu, J. Rajendran, C. Yang, R. Karri, Shielding heterogeneous MPSoCs from untrustworthy 3PIPs through security-driven task scheduling, IEEE Transactions on Emerging Topics in Computing 2 (2014) 461–472.
[64] O. Keren, I. Levin, M. Karpovsky, Duplication based one-to-many coding for Trojan HW detection, in: Defect and Fault Tolerance in VLSI Systems (DFT), 2010 IEEE 25th International Symposium on, IEEE, pp. 160–166.
[65] J. Rajendran, H. Zhang, O. Sinanoglu, R. Karri, High-level synthesis for security and trust, in: On-Line Testing Symposium (IOLTS), 2013 IEEE 19th International, IEEE, pp. 232–233.
[66] T. Reece, D.B. Limbrick, W.H. Robinson, Design comparison to identify malicious hardware in external intellectual property, in: Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on, IEEE, pp. 639–646.
[67] Trusted integrated circuits (TIC) program announcement, 2011.
[68] K. Vaidyanathan, B.P. Das, L. Pileggi, Detecting reliability attacks during split fabrication using test-only BEOL stack, in: Proceedings of the 51st Annual Design Automation Conference, ACM, pp. 1–6.

[69] M. Jagasivamani, P. Gadfort, M. Sika, M. Bajura, M. Fritze, Split-fabrication obfuscation: metrics and techniques, in: Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on, IEEE, pp. 7–12.
[70] B. Hill, R. Karmazin, C.T.O. Otero, J. Tse, R. Manohar, A split-foundry asynchronous FPGA, in: Custom Integrated Circuits Conference (CICC), 2013 IEEE, IEEE, pp. 1–4.
[71] Y. Xie, C. Bao, A. Srivastava, Security-aware design flow for 2.5D IC technology, in: Proceedings of the 5th International Workshop on Trustworthy Embedded Devices, ACM, pp. 31–38.
[72] J. Valamehr, T. Sherwood, R. Kastner, D. Marangoni-Simonsen, T. Huffmire, C. Irvine, T. Levin, A 3-D split manufacturing approach to trustworthy system development, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32 (2013) 611–615.
[73] K. Vaidyanathan, B.P. Das, E. Sumbul, R. Liu, L. Pileggi, Building trusted ICs using split fabrication, in: 2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 1–6.
[74] F. Imeson, A. Emtenan, S. Garg, M.V. Tripunitara, Securing computer hardware using 3D integrated circuit (IC) technology and split manufacturing for obfuscation, in: USENIX Security Symposium, pp. 495–510.
[75] K. Xiao, D. Forte, M.M. Tehranipoor, Efficient and secure split manufacturing via obfuscated built-in self-authentication, in: Hardware Oriented Security and Trust (HOST), 2015 IEEE International Symposium on, IEEE, pp. 14–19.
[76] D.B. Roy, S. Bhasin, S. Guilley, J.-L. Danger, D. Mukhopadhyay, From theory to practice of private circuit: a cautionary note, in: Computer Design (ICCD), 2015 33rd IEEE International Conference on, IEEE, pp. 296–303.