Chapter 2
Security Validation in Modern SoC Designs

Sandip Ray, Swarup Bhunia and Prabhat Mishra

2.1 Security Needs in Modern SoC Designs

System-on-Chip (SoC) architecture pervades modern computing devices, being the prevalent design approach for devices in embedded, mobile, wearable, and Internet-of-Things (IoT) applications. Many of these devices have access to highly sensitive information or data (often collectively called "assets"), that must be protected against unauthorized or malicious access. The goal of SoC security *architecture* is to develop mechanisms to ensure this protection. The goal of SoC security *validation* is to ensure that such mechanisms indeed provide the protection needed. Clearly the two activities are closely inter-related in typical SoC security assurance methodologies. This chapter is about the security validation component, but we touch upon architectural issues as necessary.

To motivate the critical role of security validation activities, it is necessary to clarify (1) what kind of assets is being protected, and (2) what kind of attacks we are protecting against. One can get some flavor of the kind (and diversity) of assets by looking at the kind of activities we perform on a typical mobile system. Figure 2.1 tabulates some obvious end user usages of a standard smartphone and the kind of end user information accessed during these usages. Note that it includes such intimate information as our sleep pattern, health information, location, and finances. In addition to private end user information, there are other assets in a smartphone that may have been put by the manufacturers and OEMs, which *they* do not want to be leaked

S. Ray (✉)
Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124, USA
e-mail: sandip.ray@intel.com

S. Bhunia · P. Mishra
Department of ECE, University of Florida, Gainesville, FL 32611, USA
e-mail: swarup@ece.ufl.edu

P. Mishra
e-mail: prabhat@cise.ufl.edu

S. Bhunia et al. (eds.), *Fundamentals of IP and SoC Security*,
DOI 10.1007/978-3-319-50057-7_2

Usages	Assets Exposed
Browsing	Browsing history
Fitness tracking	Health information,sleep pattern
GPS	Location
Phone call	Contacts
Banking,Stock trading	Finances

Fig. 2.1 Some typical smartphone applications and corresponding private end user information

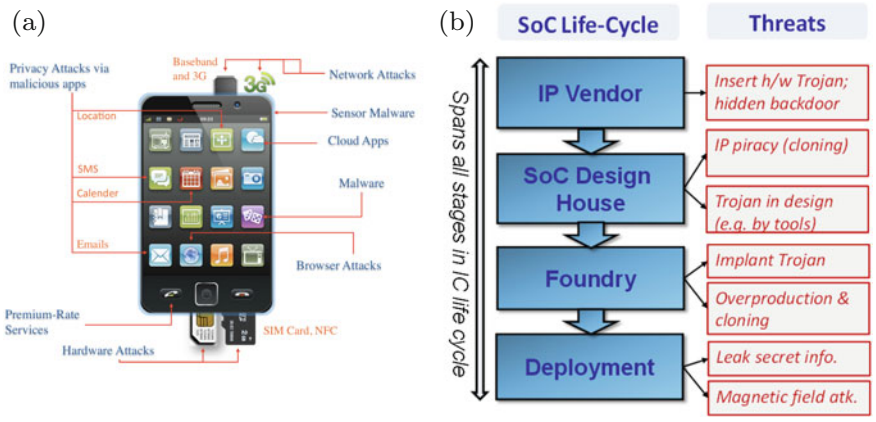

Fig. 2.2 Some potential attacks on a modern SoC design. **a** Potential attack areas for a smartphone after production and deployment. **b** Potential threats from untrusted supply chain during the design life cycle of an SoC design

out to unauthorized sources. This includes cryptographic and DRM keys, premium content locks, firmware execution flows, debug modes, etc. Note that the notion of "unauthorized source" changes based on what asset we are talking about: end user may be an unauthorized source for DRM keys while manufacturer/OEM may be an unauthorized source for end user private information.

In addition to criticality of the assets involved, another factor that makes SoC security both critical and challenging is the high diversity of attacks possible. Figure 2.2 provides a flavor of potential attacks on a modern SoC design. Of particular concern are the following two observations:

- Because of the untrusted nature of the supply chain, there are security threats at most stages of the design development, even before deployment and production.
- A deployed SoC design inside a computing device (e.g., smartphone) in the hand of the end user is prone to a large number of potential attacker entry points, including applications, software, and network, browser, and sensors. Security assurance must permit protection against this large attack surface.

We discuss security validation for the continuum of attacks from design to deployment. Given that the attacks are diverse, protection mechanisms are also varied, and

each induces a significantly different validation challenge. However, validation *technology* is still quite limited. For most of the security requirements, we still very much depend on the perspicuity, talent, and experience of the human validators to identify potential vulnerabilities.

2.2 Supply Chain Security Threats

The life cycle of a SoC from concept to deployment involves number of security threats at all stages involving various parties. Figure 2.2b shows the SoC life cycle and the security threats that span the entire life cycle. These threats are increasing with the rapid globalization of the SoC design, fabrication, validation, and distribution steps, driven by the global economic trend.

This growing reliance on reusable pre-verified hardware IPs during SoC design, often gathered from untrusted third-party vendors, severely affects the security and trustworthiness of SoC computing platforms. Statistics show that the global market for third-party semiconductor IPs grew by more than 10 % to reach more than 2.1 billion in late 2012 [1]. The design, fabrication, and supply chain for these IP cores is generally distributed across the globe involving USA, Europe, and Asia. Figure 2.3 illustrates the scenario for an example SoC that includes processor, memory controllers, security, graphics, and analog core. Due to growing complexity of the IPs as well as the SoC integration process, SoC designers increasingly tend to treat these IPs as black box and rely on the IP vendors on the structural/functional integrity of these IPs. However, such design practices greatly increase the number of untrusted components in a SoC design and make the overall system security a pressing concern.

Hardware IPs acquired from untrusted third-party vendors can have diverse security and integrity issues. An adversary inside an IP design house involved in the IP design process can deliberately insert a malicious implant or design modification to incorporate hidden/undesired functionality. In addition, since many of the IP providers are small vendors working under highly aggressive schedules, it is difficult to ensure a stringent IP validation requirement in this ecosystem. Design features may also introduce unintentional vulnerabilities, e.g., intentional information leakage through hidden test/debug interfaces or side-channels through power/performance profiles. Similarly, IPs can have uncharacterized parametric behavior (e.g., power/thermal) which can be exploited by an attacker to cause irrecoverable damage to an electronic system. There are documented instances of such attacks. For example, in 2012, a study by a group of researchers in Cambridge revealed an undocumented silicon level backdoor in a highly secure military-grade ProAsic3 FPGA device from MicroSemi (formerly Actel) [2], which was later described as a vulnerability induced unintentionally by on-chip debug infrastructure. In a recent report, researchers have demonstrated such an attack where a malicious upgrade of a firmware destroys the processor it is controlling by affecting the power management system [3]. It manifests a new attack mode for IPs, where

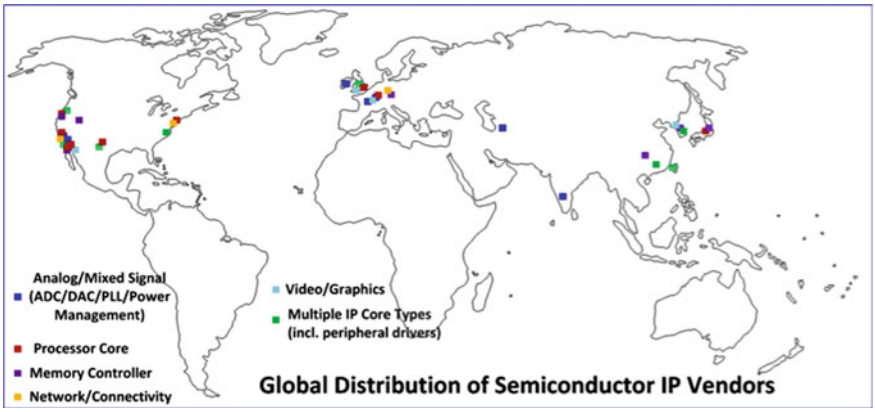

Fig. 2.3 An SoC would often contain hardware IP blocks obtained from entities distributed across the globe

firmware/software update can maliciously affect the power/performance/temperature profile of a chip to either destroy a system or reveal secret information through appropriate side-channel attack, e.g., a fault or timing attack.

Trusted and untrusted CAD tools pose similar trust issues to the SoC designers. Such tools are designed to optimize a design for power, performance, and area. Security optimization is not an option in today's tools, hence sometimes during the optimization new vulnerabilities are introduced [4]. Rogue designers in an untrusted design facility, e.g., in case of a design outsourced to a facility for Design-for-Test (DFT) or Design-for-Debug (DFD) insertion, can compromise the integrity of a SoC design through insertion of stealthy hardware Trojan. These Trojans can act as backdoor or compromise the functional/parametric properties of a SoC in various ways.

Finally, many SoC manufacturers today are fabless and hence must rely upon external untrusted foundries for fabrication service. An untrusted foundry would have access to the entire SoC design and thus brings in several serious security concerns, which include reverse-engineering and piracy of the entire SoC design or the IP blocks as well as tampering in the form of malicious design alterations or Trojan attacks. During distribution of fabricated SoCs through a typically long globally distributed supply chain, consisting of multiple layers of distributors, wholesalers, and retailers, the threat of counterfeits is a growing one. These counterfeits can be low-quality clones, overproduced chips in untrusted foundry, or recycled ones [5]. Even after deployment, the systems are vulnerable to physical attacks, e.g., side-channel attacks which target information leakage, and magnetic field attacks that aim at corrupting memory content to cause denial-of-service (DoS) attacks.

2.3 Security Policies: Requirements from Design

In addition to supply-chain threats, the design itself may have exploitable vulnerabilities. Vulnerabilities in system design, in fact, forms the quintessential objective of security study, and has been the focus of research for over three decades. At a high level, the definition of security requirement for assets in a SoC design follows the well-known "CIA" paradigm, developed as part of information security research [6]. In this paradigm, accesses and updates to secure assets are subject to the following three requirements:

- **Confidentiality**: An asset cannot be accessed by an agent unless authorized to do so.
- **Integrity**: An asset can be mutated (e.g., the data in a secure memory location can be modified) only by an agent authorized to do so.
- **Availability**: An asset must be accessible to an agent that requires such access as part of correct system functionality.

Of course, mapping these high-level requirements to constraints on individual assets in a system is nontrivial. This is achieved by defining a collection of security policies that specify which agent can access a specific asset and under what conditions. Following are two examples of representative security policies. Note that while illustrative, these examples are made up and do not represent security policy of a specific company or system.

Example 1 During boot time, data transmitted by the cryptographic engine 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 an instance of confidentiality, while Example 2 is an instance of integrity policy; however, the policies are at a lower level of abstraction since they are intended to be translated to "actionable" information, e.g., architectural or design features. The above examples, albeit hypothetical, illustrate an important characteristic of security policies: the same agent may or may not be authorized access (or update) of the same security asset depending on (1) the phase of the execution (i.e., boot or normal), or (2) the phase of the design life cycle (i.e., manufacturing or production). These factors make security policies difficult to implement. Exacerbating the problem is the fact that there is typically no central documentation for security policies; documentation of policies can range from microarchitectural and system integration documents to informal presentations and conversations among architects, designers, and implementors. Finally, the implementation of a policy is an exercise in concurrency, with different components of the policy implemented in different IPs (in hardware, software, or firmware), that coordinate together to ensure adherence to the policy.

Unfortunately, security policies in a modern SoC design are themselves significantly complex, and developed in ad hoc manner based on customer requirements

and product needs. Following are some representative policy classes. They are not complete, 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 represent 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 of 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 Versus 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 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, e.g., fuse configurations, access control priorities, cryptographic keys, firmware updates, debug and post-silicon observability information, etc. Consequently, boot imposes more stringent security requirements on IP internals and communications than normal execution. 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.

2.4 Adversaries in SoC Security

To discuss security validation, one of the first steps is to identify how a security policy can be subverted. Doing so is tantamount to identifying potential adversaries and charactertizing the power of the adversaries. Indeed, effectiveness of virtually all security mechanisms in SoC designs today are critically dependent on how realistic the model of the adversary is, against which the protection schemes are considered. Conversely, most security attacks rely on breaking some of the assumptions made regarding constraints on the adversary while defining protection mechanisms. When discussing adversary and threat models, it is worth noting that the notion of adversary can vary depending on the asset being considered: in the context of protecting DRM keys, the end user would be considered an adversary, while the content provider (and even the system manufacturer) may be included among adversaries in the context of protecting private information of the end user. Consequently, rather than focusing on a specific class of users as adversaries, it is more convenient to model adversaries corresponding to each policy and define protection and mitigation strategies with respect to that model.

Defining and classifying the potential adversary is a highly creative process. It needs considerations such as whether the adversary has physical access to the system, which components they can observe, control, modify, or reverse-engineer, etc. Recently, there have been some attempts at developing a disciplined, clean categorization of adversarial powers. One potential categorization, based on the interfaces through which the adversary can gain access to the system assets, can be used to classify them into the following six broad categories (in order of increasing sophistication). Note that there has been significant research into specific attacks in different categories, and a comprehensive treatment of different attacks is beyond the scope of this chapter; the interested reader is encouraged to look up some of the references for a thorough description of specific details.

Unprivileged Software Adversary: This form of adversary models the most common type of attack on SoC designs. Here the adversary is assumed to not have access to any privileged information about the design or architecture beyond what is available for the end user, but is assumed to be smart enough to identify or "reverse-engineer" possible hardware and software bugs from observed anomalies. The underlying hardware is also assumed to be trustworthy, and the user is assumed to have no physical access to the underlying IPs. The importance of this naïve adversarial model is that any attack possible by such an adversary can be potentially executed by any user, and can therefore be easily and quickly replicated on-field on a large number of system instances. For these types of attacks, the common "entry point" of the attack is assumed to be user-level application software, which can be installed or run on the system without additional privileges. The attacks then rely on design errors (both in hardware and software) to bypass protection mechanisms and typically get a higher privilege access to the system. Examples of these attacks include buffer overflow, code injection, BIOS infection, return-oriented programming attacks, etc. [7, 8].

System Software Adversary: This provides the next level of sophistication to the adversarial model. Here we assume that in addition to the applications, potentially the operating system itself may be malicious. Note that the difference between the system software adversary and unprivileged software adversary can be blurred, in the presence of bugs in the operating system implementation leading to security vulnerabilities: such vulnerabilities can be seen as unprivileged software adversaries exploiting an operating system bug, or a malicious operating system itself. Nevertheless, the distinction facilitates defining the root of trust for protecting system assets. If the operating system is assumed untrusted, then protection and mitigation mechanisms must rely on lower level (typically hardware) primitives to ensure policy adherence. Note that system software adversary model can have a highly subtle and complex impact on how a policy can be implemented, e.g., recall from the masquerade prevention example above that it can affect the definition of communication fabric architecture, communication protocol among IPs, etc.

Software Covert-Channel Adversary: In this model, in addition to system and application software, a side-channel or covert-channel adversary is assumed to have access to nonfunctional characteristics of the system, e.g., power consumption, wall-clock time taken to service a specific user request, processor performance counters, etc., which can be used in subtle ways to identify how assets are stored, accessed, and communicated by IPs (and consequently subvert protection mechanisms) [9, 10].

Naïve Hardware Adversary: Naive hardware adversary refers to the attackers who may gain the access to the hardware devices. While the attackers may not have advanced reverse-engineering tools, they may be equipped with basic testing tools. Common targets for these types of attacks include exposed debug interfaces and glitching of control or data lines [11]. Embedded systems are often equipped with multiple debugging ports for quick prototype validation and these ports often lack proper protection mechanisms, mainly because of the limited on-board resources. These ports are often left on purpose to facilitate the firmware patching or bug-fixing for errors and malfunctions detected on-field. Consequently, these ports also provide potential weakness which can be exploited for violating security policies. Indeed, some of the "celebrated" attacks in recent times make use of available hardware interfaces including the XBOX 360 Hack [12], Nest Thermostat Hack [13], and several smartphone jailbreaking techniques.

Hardware Reverse-Engineering Adversary: In this model, the adversary is assumed to be able to reverse-engineer the silicon implementation for on-chip secrets identification. In practice, such reverse-engineering may depend on sniffing interfaces as discussed for naïve hardware adversaries. In addition, they can depend on advanced techniques such as laser-assisted device alteration [14] and advanced chip-probing techniques [15]. Hardware reverse engineering can be further divided into two categories: (1) chip-level and (2) IP core functionality reconstruction. Both attack vectors bring security threats into the hardware systems, and permit extraction of secret information (e.g., cryptographic and DRM keys coded into hardware), which cannot be otherwise accessed through software or debugging interfaces.

Malicious Hardware Intrusion Adversary: A hardware intrusion adversary (or hardware Trojan adversary) is a malicious piece of hardware inside the SoC design. It is different from a hardware reverse-engineering adversary in that instead of "passively" observing and reverse-engineering functionality of the rest of the design components, it has the ability to communicate with them (and "fool" them into violating requisite policies). Note that as with the difference between system software and unprivileged software adversaries above, many attacks possible by an intrusion adversary can, in principle, be implemented by a reverse-engineering adversary in the presence of hardware bugs. Nevertheless, the root of trust and protection mechanisms required are different. Furthermore, in practice, hardware Trojan attacks have become a matter of concern specifically in the context of SoC designs that include untrusted third-party IPs as well as those integrated in an untrusted design house. Protection policies against such adversaries are complex, since it is unclear a priori which IPs or communication fabric to trust under this model. The typical approach taken for security in the presence of intrusion adversaries (and in some cases, reverse-engineering adversaries) is to ensure that a rogue IP $\mathscr{A}$ cannot subvert a non-rogue IP $\mathscr{B}$ into deviating from a policy.

2.5 IP-Level Trust Validation

One may wonder, why is it not possible to reuse traditional functional verification techniques to this problem? This is due to the fact that IP trust validation focuses on identifying malicious modifications such as hardware Trojans. Hardware Trojans typically require two parts: (1) a trigger, and (2) a payload. The trigger is a set of conditions that their activation deviates the desired functionality from the specification and their effects are propagated through the payload. An adversary designs trigger conditions such that they are satisfied in very rare situations and usually after long hours of operation [16]. Consequently, it is extremely hard for a naïve functional validation technique to activate the trigger condition. Below we discuss a few approaches based on simulation-based validation as well as formal methods. A detailed description of various IP trust validation techniques is available in [17, 18].

Simulation-Based Validation: There are significant research efforts on hardware Trojan detection using random and constrained-random test vectors. The goal of logic testing is to generate efficient tests to activate a Trojan and to propagate its effects to the primary output. These approaches are beneficial in detecting the presence of a Trojan. Recent approaches based on structural/functional analysis [19–21] are useful to identify/localize the malicious logic. Unused Circuit Identification (UCI) [19] approaches look for unused portions in the circuit and flag them as malicious. The FANCI approach [21] was proposed to flag suspicious nodes based on the concept of control values. Oya et al. [20] utilized well-crafted templates to identify Trojans in TrustHUB benchmarks [22]. These methods assume that the attacker uses rarely occurring events as Trojan triggers. Using "less-rare" events as trigger

will void these approaches. This was demonstrated in [23], where Hardware Trojans were designed to defeat UCI [19].

Side-Channel Analysis: Based on the fact that a trigger condition usually has extremely low probability, the traditional ATPG-based method for functional testing cannot fulfill the task of Trojan activation and detection. Bhunia et al. [16] proposed the multiple excitation of rare occurrence (MERO) approach to generate more effective tests to increase the probability to trigger the Trojan. A more recent work by Saha et al. [24] can improve MERO to get higher detection coverage by identifying possible payload nodes. Side-channel analysis focuses on the side channel signatures (e.g., delay, transient, and leakage power) of the circuit [25], which avoids the limitations (low trigger probability and propagation of payload) of logic testing. Narasimhan et al. [26] proposed the Temporal Self-Referencing approach on large sequential circuits, which compares the current signature of a chip at two different time windows. This approach can completely eliminate the effect of process noise, and it takes optimized logic test sets to maximize the activity of the Trojan.

Equivalence Checking: In order to trust an IP block, it is necessary to make sure that the IP is performing the expected functionality—nothing more and nothing less. From security point of view, verification of correct functionality is not enough. The verification engineer has to confirm that there are no other activities besides the desired functionality. Equivalence checking ensures that the specification and implementation are equivalent. Traditional equivalence checking techniques can lead to state space explosion when large IP blocks are involved with significantly different specification and implementation. One promising direction is to use Gröbner basis theory to verify arithmetic circuits [27]. Similar to [28], the reduction of specification polynomial with respect to Gröbner basis polynomials is performed by Gaussian elimination to reduce verification time. In all of these methods, when the remainder is nonzero, it shows that the specification is not exactly equivalent with the implementation. Thus, the nonzero remainder can be analyzed to identify the hidden malfunctions or Trojans in the system.

Model Checking: Model checking is the process of analyzing a design for the validity of properties stated in temporal logic. A model checker takes the Register Transfer Level (RTL) (e.g., Verilog) code along with the property written as a Verilog assertion and derives a Boolean satisfiability (SAT) formulation for validating/invalidating the property. This SAT formulation is fed to a SAT engine, which then searches for an input assignment that violates the property [29]. In practice, designers know the bounds on the number of steps (clock cycles) within which a property should hold. In Bounded Model Checking (BMC), a property is determined to hold for at least a finite sequence of state transitions. The Boolean formula for validating/ invalidating the target property is given to a SAT engine, and if a satisfying assignment is observed within specific clock cycles, that assignment is a witness against the target property [30]. The properties can be developed to detect Trojans that corrupt critical data and verify the target design for satisfaction of these properties using a bounded model checker.

Theorem Proving: Theorem provers are used to prove or disprove properties of systems expressed as logical statements. However, verifying large and complex systems using theorem provers require excessive effort and time. Despite these limitations, theorem provers have currently drawn a lot of interest in verification of security properties on hardware. In [31–33], the Proof-Carrying Hardware (PCH) framework was used to verify security properties on soft IP cores. Supported by the Coq proof assistant [34], formal security properties can be formalized and proved to ensure the trustworthiness of IP cores. The PCH method is inspired from the proof-carrying code (PCC), which was proposed by Necula [35]. The central idea is that untrusted developers/vendors certify their IP. During the certification process, the vendor develops *safety proof* for the safety policies provided by IP customers. The vendor then provides the user with the IP design, which includes the formal proof of the safety properties. The customer becomes assured of the safety of the IP by validating the design using a proof checker. A recent approach presented a scalable trust validation framework using a combination of theorem proving and model checking [36].

2.6 Security Along SoC Design Life Cycle

We now turn to the problem of system-level security validation for the SoC designs. This process takes place in the SoC design house and continues across the system design life cycle. When performing system-level validation, the constituent IPs are assumed to have undergone a level of standalone trust validation before integration.

Figure 2.4 provides a high-level overview of the SoC design life cycle. Each component of the life cycle, of course, involves a large number of design, development, and validation activities. Here, we summarize the key activities involved along the life cycle, that pertain to security. Subsequent sections will elaborate on the individual activities.

Risk Assessment. Security requirements definition is a key part of product planning, and happens concurrently with (and in close collaboration with) the definition of architectural features of the product. This process involves identifying the security assets in the system, their ownership, and protection requirements, collectively defined as *security policies* (see below). 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.

Security Architecture. The goal of a security architecture is to design mechanisms for protection of system assets as specified by the PSS. It includes several components, as follows: (1) identifying and classifying potential adversary for each asset; (1) determining attacker entry points, also referred to as threat modeling; and (3) developing protection and mitigation strategies. The process can identify additional security policies—typically at a lower level than those identified during risk assessment (see below)—which are added to the PSS. The security definition typi-

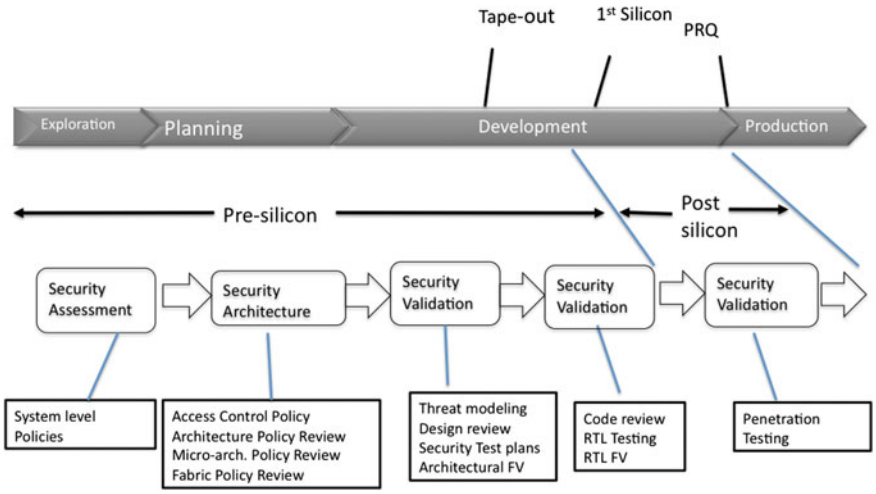

Fig. 2.4 A typical SoC life cycle from exploration to production

cally proceeds in collaboration with architecture and design of other system features, including speed, power management, thermal characteristics, etc., with each component potentially influencing the others.

Security Validation. Security validation represents one of the longest and most critical part of security assurance for industrial SoC designs, spanning the architecture, design, and post-silicon components of the system life cycle. The actual validation target and properties validated at any phase, of course, depends on the collateral available in that phase. For example, we target, respectively, architecture, design, implementation, and silicon artifacts as the system development matures. Below we will discuss some of the key validation activities and associated technologies. One key component of security validation is to develop techniques to subvert the advertised security requirements of the system, and identify mitigation measures. Mitigation measures for early-stage validation targeting architecture and early system design often include significant refinement of the security architecture itself. At later stages of the system life cycle, when architectural changes are no longer feasible due to product maturity, mitigation measures can include software or firmware patches, product defeature, etc.

2.7 Security Validation Activities

Unfortunately, the role of security validation is different from most other kinds of validation (such as functional or power-performance or timing) since the requirements are typically less precise. In particular, the goal of security validation is to "validate conditions related to security and privacy of the system that are not covered by other

validation activities." The requirement that security validation focuses on targets not covered by other validation is important given the strict time-to-market constraints, which preclude duplication of resources for the same (or similar) validation tasks; however, it puts onus on the security validation organization to understand activities performed across the spectrum of the SoC design validation and identify holes that pertain to security. To exacerbate the problem, a significant amount of security objectives are not clearly specified, making it difficult to (1) identify validation tasks to be performed, and (2) develop clear coverage/success criteria for the validation. Consequently, the validation plan includes a large number of diverse activities that range from the science to the art and sometimes even "black magic."

At a high level, security validation activities can be divided roughly among the following four categories.

Functional Validation of Security-sensitive Design Features. This is essentially extension to functional validation, but pertain to design elements involved in critical security feature implementations. An example is the cryptographic engine IP. A critical functional requirement for the crypographic 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, security validation planning may determine that correctness of cryptographic functionality to be crucial enough to justify further validation beyond the coverage provided by vanilla functional validation activities. Consequently, such an IP may undergo more rigorous testing, or even formal analysis in some cases. Other such critical IPs may include IPs involved in secure boot, on-field firmware patching, etc.

Validation of Deterministic Security Requirements. Deterministic security requirements are validation objectives that can be directly derived from security policies. Such objectives typically encompass access control restrictions, address translations, etc. Consider an access control restriction that specifies a certain range of memory to be protected from Direct Memory Access (DMA) access; this may be done to ensure protection against code-injection attacks, or protect a key that is stored in such location, etc. An obvious derived validation objective is to ensure that all DMA calls for access to a memory whose address translates to an address in the protected range must be aborted. Note that validation of such properties may not be included as part of functional validation, since DMA access requests for DMA-protected addresses are unlikely to arise for "normal" test cases or usage scenarios.

Negative Testing. Negative testing looks beyond the functional specification of designs to identify if security objectives can be subverted or are underspecified. Continuing with the DMA-protection example above, negative testing may extend the deterministic security requirement (i.e., abortion of DMA access for protected memory ranges) to identify if there are any other paths to protected memory in addition to address translation activated by a DMA access request, and if so, potential input stimulus to activate such paths.

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, they are performed for attacking complex security objectives, typically at hardware/firmware/software interfaces or at the chip boundary.

2.8 Validation Technologies

Recall from above that focused functional validation of security-critical design components form a key constituent of security validation. From that perspective, security validation includes and supersedes all functional validation tools, flows, and methodologies. Functional validation of SoC designs is a mature and established area, with a number of comprehensive surveys covering different aspects [37, 38]. In this section, we instead consider validation technologies to support other validation activities, e.g., negative testing, white-box hacking, etc. As discussed above, these activities inherently depend on human creativity; tools, methodologies, and infrastructures around them primarily act as assistants, filling in gaps in human reasoning and providing recommendations.

Security validation today primarily uses three key technologies: fuzzing, penetration testing, and formal or static analysis. Here we provide a brief description of these technologies. Note that fuzzing and static analysis are very generic techniques with applications beyond security validation; our description will be confined to their applications only on security.

Fuzzing. Fuzzing, or fuzz testing [39], is a testing technique for hardware or software that involves providing invalid, unexpected, or random inputs and monitoring the result for exceptions such as crashes, or failing built-in code assertions or memory leaks. Figure 2.5 demonstrates a standard fuzzing framework. It was developed as a software testing approach, and has since been adapted to hardware/software systems. It is currently a common practice in industry for system-level validation. 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. Smart fuzzing

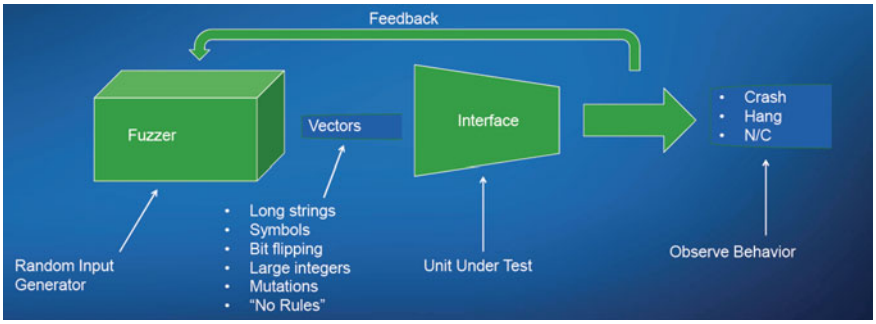

Fig. 2.5 A pictorial representation of fuzzing framework used in post-silicon SoC security validation

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

Penetration Testing. A penetration test is an attack on a computer system with the intention to find security weakness, potentially gaining access to it, its functionality, and data. It is typically performed by expert hackers often with deep knowledge of system architecture, design, and implementation characteristics. Note that while there are commonalities between penetration testing and testing done on functional validation, there are several important differences. In particular, roughly, penetration testing involves iterative application of the following three phases:

1. **Attack Surface Enumeration**. The first task is to identify the features or aspects of the system that are vulnerable to attack. This is typically a creative process involving a smorgasbord of activities, including documentation review, network service scanning, and even fuzzing or random testing (see below).
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.
3. **Result Analysis**. If the attack is successful, then in this phase the resulting state of the target is compared against security objectives and policy definitions to determine if the system was 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.

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

Clearly, the efficacy of penetration testing critically depends on the ability to identify the attack surface in the first phase above. Unfortunately, rigorous methodologies for achieving this are lacking. Following are some of the typical activities in current industrial practice to identify attacks and vulnerabilities. We classify them below as "easy," "medium," and "hard" depending on the creativity necessary. Note that there are tools to assist the human in many of the activities below [40, 41]. However, determining the relevancy 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 (e.g., specification, architectural materials, etc.), known vulnerabilities or misconfigurations of IPs, software, or integration tools, missing patches, use of obsolete or out-of-date software versions, etc.
- **Medium Approaches**. These include inferring potential vulnerabilities in the target of interest from information about misconfigurations, vulnerabilities, and attacks in related or analogous products, e.g., a competitor product, a previous software version, etc. Other activities of similar complexity involve executing relevant public security tools or published attack scenarios against the target.
- **Hard Approaches**. This includes 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, *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.

Static or Formal Reasoning. This involves making use of mathematical logic to either derive a security assurance requirement formally, or identifying flaws in the target system (architecture, design, or implementation). Application of formal methods typically involve significant effort, either in the manual exercise of performing deductive reasoning or in developing abstractions of the security objective which are amenable to analysis by automated formal tools [38, 42]. In spite of the cost, however, the effort is justified for highly critical security objectives, e.g., cryptographic algorithm implementation. Furthermore, for some critical properties, automated formal methods can be used in a light-weight manner as effective state exploration tools. For example, TOCTOU property violations often involve scenarios of overlapping execution of different instances of the same protocol, which are effectively exposed by formal methods tools [43]. Finally, formal proofs have also been used as certification mechanisms for third party IP vendors to convey security assurance to SoC system integration teams [33].

2.9 Summary

We have provided a tutorial overview of the industrial practices in security assurance and validation of modern SoC designs. The goal has been to give the reader an overall big picture, provide an understanding of the current state of the practice, and describe the different pieces of a highly complex ecosystem that must interact and cooperate to ensure trustworthiness of our computing devices. The picture of the current practice is scary. On the one hand, the complexity involved is staggering and increasing at an alarming rate. On the other hand, the state of the art in current practice is to depend on human creativity and experience to identify innovative attacks within a small time window before the system goes on field (and is exposed to attacks from the "bad guys")—an approach that we know cannot scale over the complexity we are encountering. While there are promising emergent approaches, we are very far from solving the problem of creating trustworthy computing devices. The need is to develop a disciplined approach to security assurance, from the ground up. Perhaps more importantly, it may require a highly cooperative research initiative involving the different participants, viz., architects, designers, validators, and even cross-cutting stake-holders such as power/performance architects, physical design engineers, etc. Our objective for this chapter has been to serve as the starting point for researchers to understand the overall complexity and contribute to development of trustworthy and secure systems.

Although we covered a broad spectrum of activities on security, we only scratched the surface. There are more complexities involved, including trade-offs with power management, physical design, testing, etc., as well as complex supply chain issues, which we only touched peripherally. The readers interested in deeper exploration are encouraged to explore into some of the references, which include challenges and surveys of specific components, and use the discussions in this paper as a glue for connecting the different pieces.

References

1. Ramamoorthy, G.: Market share analysis: semiconductor design intellectual property, worldwide (2012). https://www.gartner.com/doc/2403015/market-share-analysis-semiconductor-design
2. Skorobogatov, S., Woods, C.: Breakthrough silicon scanning discovers backdoor in military chip. In: CHES, pp. 23–40 (2012)
3. Messmer, E.: RSA security attack demo deep-fries Apple Mac components (2014). http://www.networkworld.com/news/2014/022614-rsa-apple-attack-279212.html
4. Nahiyan, A., Xiao, K., Forte, D., Jin, Y., Tehranipoor, M.: AVFSM: a framework for identifying and mitigating vulnerabilities in FSMs. In: Design Automation Conference (DAC) (2016)
5. Tehranipoor, M., Guin, U., Forte, D.: Counterfeit Integrated Circuits: Detection and Avoidance. Springer (2014)
6. Greenwald, S.J.: Discussion topic: what is the old security paradigm. In: Workshop on New Security Paradigms, pp. 107–118 (1998)

7. Davi, L., Sadeghi, A.R., Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In: Proceedings of the 2009 ACM workshop on Scalable trusted computing, STC'09 (2009)
8. Schuster, F., Tendyck, T., Liebchen, C., Davi, L., Sadeghi, A.R., Holz, T.: Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In: Proceedings of the 36th IEEE Symposium on Security and Privacy (2015)
9. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: 16th Annual International Cryptology Conference, pp. 104–113 (1996)
10. Kocher, P.C., Jaffe, J., Jun, B.: Differential power analysis. In: 19th Annual International Cryptology Conference, pp. 398–412 (1999)
11. Ray, S., Yang, J., Basak, A., Bhunia, S.: Correctness and security at odds: post-silicon validation of modern SoC designs. In: Proceedings of the 52nd Annual Design Automation Conference (2015)
12. Homebrew Development Wiki: JTAG-Hack. http://dev360.wikia.com/wiki/JTAG-Hack
13. Hernandez, G., Arias, O., Buentello, D., Jin, Y.: Smart nest thermostat: a smart spy in your home. In: Black Hat USA (2014)
14. Rowlette, R., Eiles, T.: Critical timing analysis in microprocessors using near-IR laser assisted device alteration (LADA). In: IEEE International Test Conference, pp. 264–273 (2003)
15. http://www.chipworks.com/
16. Chakraborty, R.S., Wolff, F., Paul, S., Papachristou, C., Bhunia, S.: MERO: A statistical approach for hardware trojan detection. In: Workshop on Cryptographic Hardware and Embedded Systems (2009)
17. Mishra, P., Bhunia, S., Tehranipoor, M.: Hardware IP Security and Trust. Springer (2016)
18. Guo, X., Dutta, R.G., Jin, Y., Farahmandi, F., Mishra, P.: Pre-silicon security verification and validation: a formal perspective. In: ACM/IEEE Design Automation Conference (DAC) (2015)
19. Hicks, M., Finnicum, M., King, S., Martin, M., Smith, J.: Overcoming an untrusted computing base: detecting and removing malicious hardware automatically. In: IEEE Symposium on Security and Privacy (SP), pp. 159–172 (2010)
20. Oya, M., Shi, Y., Yanagisawa, M., Togawa, N.: A score-based classification method for identifying hardware-trojans at gate-level netlists. In: Design Automation and Test in Europe (DATE), pp. 465–470 (2015)
21. Waksman, A., Suozzo, M., Sethumadhavan, S.: Fanci: identification of stealthy malicious logic using boolean functional analysis. In: ACM SIGSAC Conference on Computer and Communications Security, pp. 697–708 (2013)
22. Trust-HUB. https://www.trust-hub.org/
23. Sturton, C., Hicks, M., Wagner, D., King, S.: Defeating UCI: building stealthy and malicious hardware. In: 2011 IEEE Symposium on Security and Privacy (SP), pp. 64–77 (2011)
24. Saha, S., Chakraborty, R., Nuthakki, S., Anshul, Mukhopadhyay, D.: Improved test pattern generation for hardware trojan detection using genetic algorithm and boolean satisfiability. In: Cryptographic Hardware and Embedded Systems (CHES), pp. 577–596 (2015)
25. Aarestad, J., Acharyya, D., Rad, R., Plusquellic, J.: Detecting trojans through leakage current analysis using multiple supply pad I_{ddq}s. In: IEEE Transactions on Information Forensics and Security, pp. 893–904 (2010)
26. Narasimhan, S., Wang, X., Du, D., Chakraborty, R., Bhunia, S.: Tesr: a robust temporal self-referencing approach for hardware trojan detection. In: Hardware-Oriented Security and Trust (HOST), pp. 71–74 (2011)
27. Farahmandi, F., Mishra, P.: Automated test generation for debugging arithmetic circuits. In: Design Automation and Test in Europe (DATE) (2016)
28. Lv, J., Kalla, P., Enescu, F.: Efficient groebner basis reductions for formal verification of galois field arithmetic circuits. IEEE Trans. CAD (TCAD) **32**, 1409–1420 (2013)
29. Cadence Berkeley Lab: The cadence SMV model checker. http://www.kenmcmil.com
30. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Tools and Algorithms for the Construction and Analysis of Systems, p. 193207 (1999)

31. Jin, Y.: Design-for-security vs. design-for-testability: A case study on dft chain in cryptographic circuits. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI) (2014)
32. Jin, Y., Yang, B., Makris, Y.: Cycle-accurate information assurance by proof-carrying based signal sensitivity tracing. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 99–106 (2013)
33. Love, E., Jin, Y., Makris, Y.: Proof-carrying hardware intellectual property: a pathway to trusted module acquisition. IEEE Trans. Inf. Forensics Secur. **7**(1), 25–40 (2012)
34. INRIA: The coq proof assistant (2010). http://coq.inria.fr/
35. Necula, G.C.: Proof-carrying code. In: POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 106–119 (1997)
36. Guo, X., Dutta, R., Mishra, P., Jin, Y.: Scalable SoC trust verification using integrated theorem proving and model checking. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) (2016)
37. Bhadra, J., Abadir, M.S., Wang, L., Ray, S.: A survey of hybrid technqiues for functional verification. IEEE Des. Test Comput. **24**(2), 112–122 (2007)
38. Gupta, A.: Formal hardware verification methods: a survey. Formal Methods Syst. Des. **2**(3), 151–238 (1992)
39. Takanen, A., DeMott, J.D., Mille, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House (2008)
40. Corporation, M.: Microsoft free security tools microsoft baseline security analyzer (2015). https://blogs.microsoft.com/cybertrust/2012/10/22/microsoft-free-security-tools-microsoft-baseline-security-analyzer/
41. Software, F.: (2012). http://secunia.com
42. Clarke, E.M., Grumberg, O., Peled, D.A.: Model-Checking. The MIT Press, Cambridge, MA (2000)
43. Krstic, S., Yang, J., Palmer, D.W., Osborne, R.B., Talmor, E.: Security of SoC firmware load protocol. In: IEEE HOST (2014)

http://www.springer.com/978-3-319-50055-3

Fundamentals of IP and SoC Security
Design, Verification, and Debug
Bhunia, S.; Ray, S.; Sur-Kolay, S. (Eds.)
2017, VI, 316 p. 105 illus., 52 illus. in color., Hardcover
ISBN: 978-3-319-50055-3