INVITED PAPER

System-on-Chip Platform Security Assurance: Architecture and Validation

By Sandip Ray, *Senior Member IEEE*, Eric Peeters, Mark M. Tehranipoor, *Senior Member IEEE*, and Swarup Bhunia, *Senior Member IEEE*

ABSTRACT | Modern system-on-chip (SoC) designs include a wide variety of highly sensitive assets which must be protected from unauthorized access. A significant aspect of SoC design involves exploration, analysis, and evaluation of resiliency mechanisms against attacks to such assets. These attacks may arise from a number of sources, including malicious intellectual property blocks (IPs) in the hardware, malicious or vulnerable firmware and software, insecure communication of the system with other devices, and side-channel vulnerabilities through power and performance profiles. Countermeasures for these attacks are equally diverse, which include architecture, design, implementation, and validation-based protection. In this paper, we provide a comprehensive overview of the security infrastructure in modern SoC designs, including both resiliency techniques and their validation paradigms at presilicon and postsilicon stages. We identify gaps in current resiliency and analysis architectures and propose design and validation solutions to address them. Finally, we provide industry perspectives on the role and impact of current practices on SoC security, and discuss some emerging trends in this important area.

KEYWORDS | Security architecture; security policy; system-on-chip (SoC) security; trusted SoC; untrusted IPs

I. INTRODUCTION

We are living in a world surrounded by billions of computing systems, identifying, tracking, and analyzing some of our intimate personal information, including health, sleep, location, and network of friends. The trend is toward even higher proliferation of such devices, with an estimated 50 billion smart, connected devices by 2020, according to a recent report by Cisco. These devices generate, process, and exchange a large amount of sensitive information and data (often collectively referred to as "security assets" or simply "assets"). In addition to private end-user information, assets include security-critical parameters introduced during the system architecture definition, e.g., fuses, cryptographic, and digital rights management (DRM) keys, firmware execution flows, and on-chip debug modes. Malicious access to these assets can result in leakage of company trade secrets for device manufacturers or content providers, identity theft or privacy breach for end users, and even destruction of human life.

Security assurance of a modern computing device involves a number of challenges. One key challenge is the sheer complexity of the design. Most modern computing systems are architected via a system-on-chip (SoC) paradigm, viz., through a composition of predesigned hardware or software blocks [referred to as intellectual properties (IPs)] that interact through a network of on-chip communication fabrics. The IPs themselves are highly complex artifacts optimized for performance, power, and silicon overhead. Adding to the complexity are the communication protocols used in implementing complex system-level use cases. Finally, security assets are sprinkled at different IPs across the design, and access to the assets is governed by complex security policies. The policies are defined by system architects as well as different IP and SoC integration teams, and undergo refinement and modification throughout the system development. This makes it challenging to validate a system, develop architectures to provide built-in resilience against unauthorized access, or update security requirements, e.g., in response to changing customer needs.

Another source of challenge is the supply chain involved in the development of a modern computing device. There is a large number of players involved, including IP providers, SoC design house, and foundry. With the increasing globalization of the semiconductor design and fabrication process, each of these players

Manuscript received October 3, 2016; revised May 12, 2017; accepted June 1, 2017. The work presented here is supported in part by National Science Foundation (NSF) and Semiconductor Research Corporation (SRC). *(Corresponding author: Sandip Ray.)*

S. Ray is with NXP Semiconductors, Austin, TX 78735 USA (e-mail: sandip.r.ray@gmail.com).
E. Peeters is with Texas Instruments Inc., Dallas, TX, USA.
M. M. Tehranipoor and **S. Bhunia** are with the University of Florida, Gainesville, FL 32611 USA.

Digital Object Identifier: 10.1109/JPROC.2017.2714641

often involves large number of organizations—often across geography—coordinating to create a complex supply-chain pipeline. Every component of the pipeline is vulnerable to malicious design alterations, subversions, piracy, and other security threats. Even in cases where a component is designed without intended malice, aggressive time-to-market requirements and high optimization needs often result in errors and vulnerabilities inadvertently left in the design, which can be exploited by a malicious adversary in the field.

Given the broad spectrum of vulnerabilities and corresponding mitigation strategies, the subject of SoC security today is highly fragmented. Different research groups focus on different aspects of the problem, without full understanding of the tradeoffs and synergies. For example, there has been little work on integrating techniques for supply-chain security with architectural resiliency initiatives for design-level security implementation. Consequently, security research in different communities runs into the danger of reinventing the "wheel" that already exists in another context, or creating a solution for one problem that breaks fundamental requirements of another.

The goal of this paper is to provide a comprehensive overview of security assurance requirements and practices in modern SoC designs. Existing literature notably lacks such a coverage on SoC security, specifically in materials related to industrial practices. We discuss the SoC design lifecycle, identify the security concerns tackled at each stage, and the challenges involved in addressing them, which include technical obstacles (e.g., scalability of analysis), as well as gaps in methodology and supply chain (e.g., unavailability of specification, interface, and adversary models, untrusted or buggy third-party IP blocks). We discuss current industrial practices, point out their inadequacies, and present results from some emergent research that provide promising directions.

The remainder of the paper is organized as follows. Section II provides a basic overview of SoC security challenges, identifying the design, architecture, and supply-chain roots. Section III discusses the overall spectrum of security solutions employed in different phases of the SoC design lifecycle. Sections IV–VIII go into more detail in the different components of secure SoC design, viz., assessment, specification, architecture, and validation. For each of the activities discussed, we describe the current practices and point out their limitations. Section X discusses security challenges coming from other aspects of design, in particular, interoperability with validation. We discuss a few emergent approaches in Section XI, and conclude in Section XII.

II. OVERVIEW OF SOC SECURITY CHALLENGES

Before getting into the current practice of security assurance, it is sobering to understand the scale and complexity of security threats to which our computing systems are exposed. The security literature over the years is replete with instances of security attacks, and the number of attack instances has been growing over the years. As an example, *Forbes Magazine* recently reported results from Cansecwest 2015 [1], where four different attacks were presented [2]–[5] exploiting security vulnerabilities related to the system management mode on the Intel processor architecture running BIOS. Each attack could "hijack" millions of BIOS from diverse system vendors. Perhaps more disturbingly, these attacks represent only a very small segment of the attack surface of a computing device, e.g., exploiting vulnerabilities of a specific feature present in the architecture of the CPU, which is only one IP of a modern SoC design. To give an idea of the scale of the attack surface, Fig. 1 illustrates some of the potential security attacks in a smartphone. Note that each category of attack represents a rich body of literature, with several documented instances.

Unfortunately, the situation is exacerbated with increasing proliferation of smart computing devices and platforms in the IoT regime. First, the diversity of these devices provides newer unanticipated avenues for attacks (see below). Second, the devices do not operate in isolation, but are in continuous communication with billions of other smart devices through the network of cloud and data centers. Consequently, it has become possible for an adversary to exploit the vulnerability of one (or a few) systems to infect a large number of connected devices. Furthermore, the ramifications of the attacks are staggering. A decade back, smartphones represented the limits of the imagination of many people in sophistication of computing applications; now, we have realized applications in the scale of smart cities, homes, and multiplexes. A security vulnerability in a single device in this ecosystem can have a ripple effect affecting the entire application, with potentially catastrophic consequences to national security, economy, and human life.

Computer security is, of course, a mature area of research with significant results going back to at least four decades, resulting in a large body or results on adversary

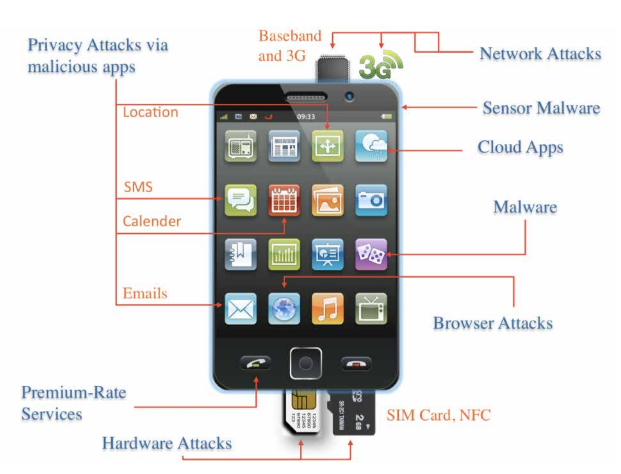

Fig. 1. ***Some potential attacks on a modern smartphone.***

models, resilience techniques, and validation approaches for various computing models. To understand the challenge of security assurance on modern computing platforms, it is worth understanding the inadequacy of these technologies. In the rest of the section, we explain this issue and discuss how the changing computing paradigm from desktops to handheld affects and constrains the applicable security solutions. We then discuss the scope of SoC security assurance in greater detail.

A. Design Challenges to Security

There are four key factors that contribute to design challenges in ensuring the security of modern computing platforms: 1) high complexity of devices; 2) aggressive time-to-market requirements that do not provide adequate validation time; 3) high diversity; and 4) continuous connectivity, particularly of devices that were not originally meant to be connected [6], [7].

To understand the critical role of complexity, recall that even a decade back one could clearly demarcate computing systems in two categories: general-purpose systems (e.g., desktops, laptops, etc.) and embedded systems (e.g., medical equipment, personal organizer, automotive infotainment, etc.). The general-purpose systems were characterized by high programmability to support diverse use-case scenarios, resulting in a complex hardware architecture; nevertheless, they also provided a reasonably clean interface (e.g., at the instruction-set architecture) to enable software development at a level of abstraction without significant concern on hardware or power/performance constraints for most application or even system-level software development. On the other end, embedded systems were targeted for unique use cases. Each use case induced unique constraints on form factor, power, performance, security, reliability, etc., and drove the design, architecture, and optimization of the whole system. Consequently, the systems were typically characterized by tight coupling of hardware and software modules optimized for the metrics of interest as dictated by the target use case. Research in security assurance and verification consequently looked at 1) embedded systems security [8] where potential vulnerabilities were limited by the narrowness of target use-cases; or 2) general-purpose computing system security, where the decoupling of hardware and software permitted exploration of the two components separately. Furthermore, since for general-purpose systems the hardware architecture was fairly standard and the supply-chain reasonably trustworthy, one could trust them to be free of malicious instrumentation. Consequently, the hardware was taken as the root of trust and the primary security focus was on software components. However, with the advent of modern "embedded devices" like smartphones, tablets, smart watches, and wearables, the demarcation between embedded and general-purpose systems has become murky; these devices inherit the complexity of embedded systems, including the tight hardware/software integration and aggressive optimizations to address form factor, power/performance, and usage-specific constraints. However, they also inherit the complexity of general-purpose systems, including a diversity and complexity of use cases (e.g., the number of use cases of a smartphone or tablet is compatible with those in a laptop or desktop). Consequently, one must rethink architecture and validation from the ground up to ensure that we can encompass systems of such complexity.

Furthermore, these devices must conform to aggressive time-to-market requirements. The system lifecycle from conception to production ranges from three to four years for a desktop or a laptop; this is shrunk to less than a year for a mobile device. The shrinking is driven by market economics, e.g., the consumer device refresh cycle. Furthermore, the launch schedule is governed by seasonal demand cycles, e.g., back-to-school and holiday seasons; the success in launching a product within a specific short time-window may mean the difference between high market share and complete failure [9], [10]. This severely constrains the amount of system-level functional validation performed, resulting in vulnerability escapes to silicon or in-field.

Finally, exacerbating the situation are two additional factors in the modern computing environment, large system diversity and their continuous connectivity. The high diversity of consumer computing devices, coupled with aggressive time-to-market requirements, implies an urgent need for reuse of design blocks. However, security requirements vary significantly depending on the product, and cannot be "preverified" at the IP level, e.g., security requirements from the display IP may vary significantly depending on whether it is used for a mobile phone or a gaming system. On the other hand, the lack of system-level hardware/software security validation technology, together with the inadequate system-level validation time and lack of sufficient documentation of system-level security objectives (see below), implies that requirements for security validation—and sometimes even security architecture—are not thought through and often reused from an earlier (sometimes different) product. This can lead to significant security holes which are only identified in-field. Furthermore, since many of these devices target relatively naïve consumers, it is difficult to ensure software/firmware patching or other measure to mitigate vulnerabilities discovered in-field are applied.

B. Supply Chain Challenges

The security picture is further complicated for SoC designs due to increasing reliance on hardware IPs gathered from untrusted third party vendors. Fig. 2 shows the SoC lifecycle and the security threats that span the entire lifecycle. These threats are increasing with the rapid globalization of the SoC design, fabrication, validation, and distribution steps. Statistics show that the global market for third

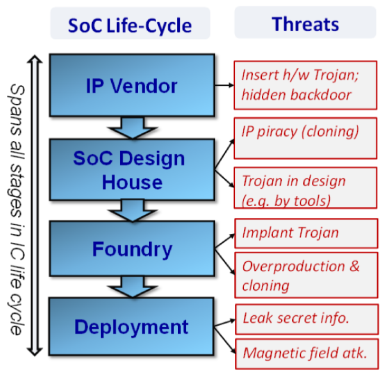

Fig. 2. ***Effects of globally distributed supply chain on SoC security.***

party semiconductor IPs reached more than 2.1 billion in late 2012 [11]. The design, fabrication, and supply chain for these IP cores are generally distributed across the globe (cf., Fig. 3). 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 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 vulnerabilities, e.g., information leakage through hidden test/debug interfaces or side channels through power/timing profiles [12].

Computer-aided design (CAD) tools pose similar trust issues to the SoC designers. Such tools are designed to optimize a design for power, performance, and area. These optimizations can introduce new vulnerabilities [13]. Rogue designers in an untrusted design facility, e.g., in case of a design outsourced to an untrusted 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 a backdoor or compromise the functional/parametric properties of a system in various ways.

Finally, many SoC manufacturers today are fabless and rely upon external untrusted foundries for fabrication service. An untrusted foundry has access to the entire 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 alterations or Trojan attacks. During distribution of fabricated SoC designs through a typically long distribution 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. 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 attacks.

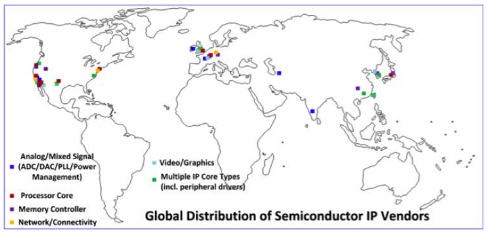
■ Analog/Mixed Signal (ADC/DAC/PLL/Power Management)
■ Processor Core
■ Memory Controller
■ Network/Connectivity
■ Video/Graphics
■ Multiple IP Core Types (incl. peripheral drivers)
Global Distribution of Semiconductor IP Vendors

Fig. 3. ***SoCs would typically integrate IP blocks from entities distributed across the globe.***

C. Scope of SoC Security

Given the complex roots of security assurance in modern computing devices, it is unsurprising that security assurance itself is a problem of very broad scope. The problem is classified into the following three groups.

1) *Hardware security:* This refers to security issues arising from problems in the underlying hardware. Current approaches to hardware security primarily focus on hardware supply-chain vulnerabilities, e.g., Trojan attacks, and counterfeit IPs.

2) *System or platform security:* This refers to vulnerabilities resulting from functional or performance bugs in the system that can be exploited by a malicious third party during execution. Examples of such vulnerabilities include functional bugs in security-critical IPs (e.g., cryptographic engine), information leakage due to unanticipated behavior when the system encounters inputs of unexpected types, information leakage from system power profile, etc.

3) *Cloud security or cybersecurity:* This refers to vulnerabilities arising from the communication of an embedded computing system or IoT either with other embedded systems in the Internet or with servers and data centers in the cloud. It includes eavesdropping or "man-in-the-middle" attacks, breach of confidentiality of the stored data in the cloud, corruption of the integrity of collected data, etc.

The remainder of the paper focuses primarily on platform security assurance, although we refer to the other components in the context of their influence on platform security. Note that while both hardware security and cybersecurity have received significant attention with several mature research programs [14], [15], there has been a dearth of a comprehensive treatment of platform security. On the other hand, platform security in fact is a large and complicated topic. The notion of "platform" itself includes all hardware, firmware, and software components of the system, typically also including the essential system and application services. To achieve security assurance at that level, one must identify all potential vulnerabilities (e.g., software, hardware, physical access, etc.), and motivations for an attack (e.g., data theft, jailbreaking, DRM bypass, etc.). Even identifying the security objectives is non-trivial; they include design features, architecture parameters, security requirements of the operating system and applications, and even the user's security expectations (even if undocumented). Finally, given aggressive time-to-market constraints, security assurance must only cover architecture and validation components not covered already by other activities; thus the security architect and validator is faced with the daunting task of understanding different designs, architectures, and validation flows, and identifying gaps in them which can undermine security objectives of the system.

III. SECURITY ALONG SOC DESIGN LIFECYCLE

Fig. 4 provides a high-level overview of the SoC design lifecycle. Each component of the lifecycle, of course, involves a large number of design, development, and validation activities. Here we summarize the key activities involved along the lifecycle that pertain to security. Subsequent sections will elaborate on the individual activities.

A. 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.

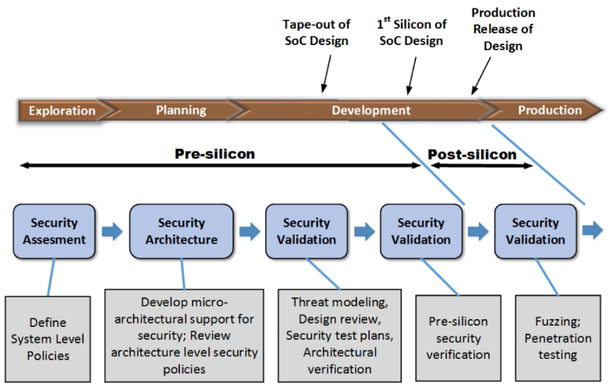
Tape-out of SoC Design
1st Silicon of SoC Design
Production Release of Design
Exploration
Planning
Development
Production
Pre-silicon
Post-silicon
Security Assesment
Security Architecture
Security Validation
Security Validation
Security Validation
Define System Level Policies
Develop micro-architectural support for security; Review architecture level security policies
Threat modeling, Design review, Security test plans, Architectural verification
Pre-silicon security verification
Fuzzing; Penetration testing

Fig. 4. ***Lifecycle of a typical SoC from exploration to production.***

B. Security Architecture

The goal of a security architecture is to design mechanisms for protection of system assets. It includes several components: 1) identifying and classifying potential adversary for each asset; 2) 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 typically proceeds in collaboration with architecture and design of other system features, including speed, power management, thermal characteristics, etc.

C. Security Validation

Security validation represents one of the most critical parts of security assurance, spanning the architecture, design, and postsilicon components of the system lifecycle. The actual validation target and properties validated at any phase depend on the collateral available in that phase, e.g., the validators target, respectively, architecture, design, implementation, and silicon artifacts as the design matures. One key activity is to subvert the advertised security requirements in PSS, and identify mitigation measures. Mitigation measures for architecture and early system design often include significant refinement of the security architecture itself. At later stages of the system lifecycle, when architectural changes are no longer feasible, mitigation measures can include software or firmware patches, product defeature, etc.

IV. INTRODUCTION TO SECURITY POLICIES

SoC security is driven by the requirement to protect system assets against unauthorized access. Such access control can be defined by confidentiality, integrity, and availability requirements [16]. The goal of a security policy is to map the requirements to "actionable" design constraints that can be used by IP implementers or SoC integrators to develop protection mechanisms.

- *Example 1:* During boot time, data transmitted by the crypto 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 a confidentiality requirement while Example 2 is an integrity constraint; however, the policies provide concrete conditions to be checked by the design for accessing an asset. Furthermore, access to an asset may vary depending on the state of execution (e.g., boot time, normal execution, etc.), or position in the development lifecycle. Following are some representative policy classes. They are not exhaustive, but illustrate the diversity of policies employed.

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

2) Information flow: Values of secure assets can sometimes be inferred without direct access, through indirect observation or "snooping" of intermediate computation or communications of IPs. Information flow policies restrict such indirect inference. An example information flow policy is given below.

- *Key obliviousness:* A low-security IP cannot infer the cryptographic keys by snooping 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.

3) 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.

4) 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 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.

5) Secure boot: Booting a system entails communication of significant security assets, e.g., fuse configurations, access control priorities, cryptographic keys, firmware updates, postsilicon observability information, etc. Consequently, boot imposes stringent security requirements on IPs and communications. Individual policies during boot can be access control, information flow, and TOCTOU requirements; however, it is often convenient to coalesce them into a unified set of boot policies.

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

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

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

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

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

The above descriptions perhaps belie the complexity involved in implementing policies. Consider the SoC configuration shown in Fig. 5. Suppose that *IP0* needs to send a message to the DRAM. Ordinarily, the message would be routed through *Router3*, *Router0*, *Router1*, and *Router2*. However, such a route permits message redirection via software. Each router includes a base address register (BAR) which is used to route messages for specific destinations. One of the routers in the proposed path, *Router0* is connected to the CPU; the BARs in this router are subject to potential overwrite by the host operating system, which can redirect a message passing through *Router0* to a different destination. Consequently, a secure message cannot be sent through this route unless the host operating system is trusted. Note that understanding the potential of redirection requires knowledge of fabric

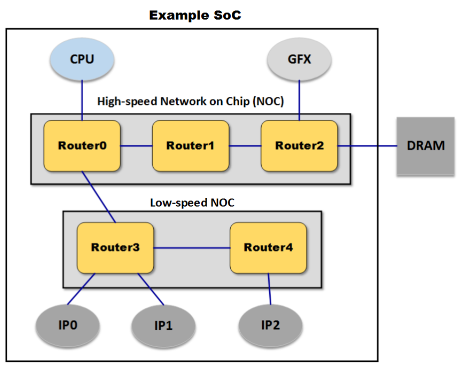

Fig. 5. ***An illustrative simple SoC configuration. SoC designs include several on-chip communication fabrics with differing speed and power profiles. This configuration has a high-speed fabric with three routers connected linearly, and a low-speed fabric with two routers also connected linearly.***

operation, routers design (e.g., the use of BARs), as well as the capabilities of the software in an adversarial role.

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

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

V. A TAXONOMY OF ADVERSARIES

To ensure that an asset is protected, the designer needs, in addition to the security policy, a comprehension of the power of the adversary. Effectiveness of virtually all protection mechanisms is critically dependent on how realistic the model of the adversary is. Conversely, most security attacks rely on breaking some of the assumptions made regarding constraints on the adversary. The notion of adversary can vary depending on the asset being considered: for protecting DRM keys, the end user would be an adversary, while the content provider (and even the system manufacturer) may be included among adversaries in the context of protecting the private information of the end user. Rather than focusing on a specific class of users as adversaries, it is more convenient to model adversaries corresponding to each policy and define protection and mitigation strategies with respect to that model.

Defining and classifying the potential adversary is a creative process. It needs considerations such as whether the adversary has physical access, which components they can observe, control, modify, or reverse engineer, etc. Recently, there has been some attempts at developing a disciplined 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 this classification is one of the several potential ones, e.g., another orthogonal classification could be done by categorizing the different potential attacks on system or application features as shown in Fig. 1. However, we discuss the design-based characterization since this is particularly useful in the context of SoC security architecture and validation.

1) *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 can 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 in the field on a large number of system instances. For this type of attacks, the common "entry point" of the attack is user-level application software which can be installed or run on the system without additional privilege. The attacks 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. [17], [18].

2) *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 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 subtle and complex impact on policy implementation, e.g., recall from the masquerade prevention example above that it can affect the definition of communication fabric architecture, communication protocol among IPs, etc.

3) *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) [19], [20].

4) Naïve hardware adversary: This refers to the attackers who may gain access to the hardware devices. While the attackers may not have advanced reverse engineering tools, they may be equipped with basic testing tools. Targets for this type of attacks include exposed debug interfaces and glitching of control or data lines [21]. Embedded systems are often equipped with multiple debugging ports for quick prototype validation. These ports also provide potential weakness which can be exploited for violating security policies.

5) Hardware reverse-engineering adversary: This adversary can reverse engineer the silicon implementation for on-chip secrets identification. In addition to sniffing interfaces, they can depend on advanced techniques such as laser-assisted device alteration and chip-probing. Hardware reverse engineering can be further divided into two categories: 1) chip-level reverse engineering; and 2) IP functionality reconstruction. Both attack vectors bring in security threats to 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.

6) Malicious hardware intrusion adversary: A hardware intrusion adversary (or hardware Trojan adversary) is a malicious hardware inside the design. This category of adversaries encapsulates potential threats arising from the SoC supply chain. It is different from a hardware reverse-engineering adversary in that instead of "passively" observing and reverse-engineering functionality, it has the ability to communicate with them (and "fool" them into violating requisite policies). Protection policies against such adversaries are complex, since it is unclear *a priori* which IPs to trust under this model. The typical approach taken for security in the presence of intruding adversaries is to ensure that a rogue IP $\mathcal{A}$ cannot subvert a trusted IP $\mathcal{B}$ into deviating from a policy.

VI. ELEMENTS OF SECURITY ARCHITECTURE

Given a plethora of complex policies and protection requirements under different classes of potential adversaries, how would we go about designing authentication mechanisms to ensure policy enforcement? Unfortunately, the state of the practice today depends heavily on human creativity. The typical approach today is to develop a baseline architecture definition which is then repeatedly refined through the following two steps:

- use threat modeling to identify potential threats to the current architecture definition (see below);
- 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 NoC to the respective IPs. Each participant in this process has sensitive assets (either static or created) during different phases of the system execution, and the security architecture must account for any potential access to these assets at any point, possibly under the relevant adversary model.

There has been significant 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, with a large number of mechanisms and architectures. One of the most common TEE architectures is the trusted platform module (TPM), which is an international standard for a secure cryptoprocessor designed to secure the hardware by integrating cryptographic keys into devices [22]. It covers methods for secure generation of cryptographic keys and limitation of their use, the requirements from random number generator, as well as 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 [23], [24]. Below we discuss three TEE frameworks specifically developed for SoC designs: Samsung KNOX, Intel® Software Guard Extension (SGX), and ARM Trustzone®. Note that in spite of differences motivated by the isolation and separation targets, the underlying architectural plans for these TEEs are similar, viz., a combination of hardware support (e.g., secure operating modes, virtualization), and software mechanisms (e.g., context switch agents, integrity check).

A. Samsung KNOX [25]

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

- trusted boot, i.e., 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.

B. ARM Trustzone [26]

TrustZone technology is a system-wide approach to 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 non-secure. Hardware supports access control and permissions for the handling of secure/non-secure 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 normal 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.

C. Intel SGX

SGX [27] 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 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 (i.e., 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) architecting a parameterized "skeleton" design that can be easily instantiated to diverse policy implementations; and 3) developing techniques for synthesizing policy implementation from high-level descriptions. Recent academic and industrial research has attempted to address some of these issues. Li *et al.* [28] provide a language and synthesis framework for certain security policies. Basak *et al.* [29] provide a microcontrol-based flexible framework for implementing diverse security policies. There have been optimized architectural support for specific classes of policies, e.g., control-flow integrity [30], Trojan resistance [31]. However, in spite of such work on pieces of the problem, we are still far away from a robust, configurable security architecture as necessary for robust system design. Some key deficiencies include interplay of secure access control with on-chip instrumentation, definition of security architectures that are configurable for different phases of system lifecycle, and lack of a centralized IP for policy implementation in the SoC design, which makes it difficult to evaluate policy compliance.

VII. THREAT MODELING

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

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

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

3) *Attack surface identification:* For each asset, identify potential adversarial actions that can subvert policies governing the asset. This requires identification, analysis, and documentation of each potential "entry point,"

This article has been accepted for inclusion in a future issue of this journal. Content is final as presented, with the exception of pagination.

i.e., any interface that transfers data relevant to the asset to an untrusted region. The entry point depends on the category of the potential adversary considered in the attack, e.g., a covert-channel adversary can make use of nonfunctional design characteristics such as power consumption or temperature to infer the ongoing computation.

4) *Risk assessment:* The potential for an adversary to subvert a security objective does not, in and of itself, warrant mitigation strategies. The risk assessment and analysis are defined in terms of the so-called DREAD paradigm, composed of the following five components: a) damage potential; b) reproducibility; c) exploitability, i.e., the skill, and resource required by the adversary to perform the attack; d) affected systems, e.g., whether the attack can affect a single system or tens or millions; and e) discoverability. In addition to the attack itself one needs to analyze the likelihood that the attack can occur on-field, motives of the adversary, etc.

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

A. Implementation Example

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

In current industrial practice, performing the above activities manually over the range of system assets and policies is a daunting manual task. Admittedly, there are available tools to assist in the different steps, e.g., documenting steps in threat modeling and severity identification [32], [33]. Nevertheless, the key architectural decisions and analysis still depend highly on human insights.

VIII. SECURITY VALIDATION OVERVIEW

Designing resilience into designs is one aspect of security assurance. The other critical aspect is validating that the security objectives of the product are indeed satisfied. Security validation is different from most other kinds of validation (such as functional or power or timing) since the requirements are typically less precise. 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 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. In practice, validation plan includes diverse activities that range from the science to the art and sometimes even "black magic."

A. Functional Validation of Security-Sensitive Design Features

This is essentially an 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 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 vanilla 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.

B. Validation of Deterministic Security Requirements

Deterministic security requirements are validation objectives that can be directly derived from security policies. They include access control restrictions, address translations, etc. 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 memory whose address translates to an address in the protected range must be aborted. Note that validation of such properties may not be included in functional validation, since DMA access requests for DMA-protected addresses are unlikely to arise for "normal" test cases or usage scenarios.

C. Negative Testing

Negative testing looks beyond the functional specification 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 potential input stimulus to activate such paths.

D. 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.

IX. VALIDATION TECHNOLOGIES

Recall that focused functional validation of security-critical design components forms a key constituent of security validation. Consequently, security validation includes all functional validation tools and methodologies. Functional validation of SoC designs is a mature and established area, with a number of comprehensive surveys covering different aspects [34], [35]. In this section, we instead consider validation technologies to support other validation activities, e.g., negative testing, white-box hacking, etc. These activities inherently depend on human creativity; tools and infrastructures primarily act as assistants, filling in gaps in human reasoning and providing recommendations.

A. Fuzzing

Fuzzing, or fuzz testing [36], 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. Fig. 6 demonstrates a standard fuzzing framework. 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. Smart fuzzing may provide a greater coverage of security attack entry points, at the cost of more upfront investment in design understanding.

B. Penetration Testing

A penetration test, or intrusion test, is an attack on a system with the intention to find security weakness. It is performed by expert hackers often with deep knowledge of system architecture, design, and implementation. 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 number 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

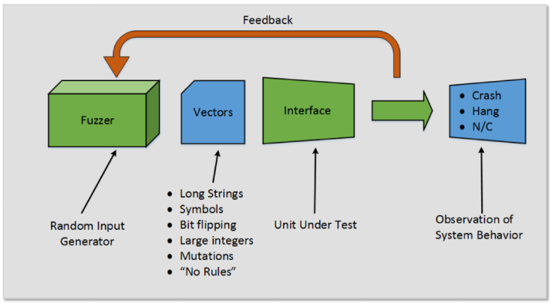

Fig. 6. ***Illustration of the fuzzing framework used in post-silicon security validation of SoC.***

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.

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 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.

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 [37], [38]. 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 (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-complexity 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.

C. 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. In spite of the cost, 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 lightweight 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 [39]. Finally, formal proofs have also been used as certification mechanisms for third party IP vendors to convey security assurance to SoC system integration teams [40].

X. SECURITY-VALIDATION TRADEOFFS

One key source of complexity in developing security assurance solutions in modern computing systems is the number of stakeholders involved. We have already seen the role of architects, designers, and validators. However, the preceding descriptions pitted them in a cooperative role, with the common objective of improving security assurance. The situation is more complex in practice because many stakeholder interests conflict with that of security. A successful SoC design needs to ensure security of the product in the presence of such interoperability needs from a large number of stakeholders. Here we consider one such interoperability requirements, viz., validation itself [21], [41].

Validation occupies a unique position in the context of interoperability. A significant component of validation (including validation of security objectives) involves postsilicon debug. This uses a fabricated, preproduction silicon to run tests and software to find errors that have been missed in presilicon validation. The tests can include functional validation tests, practical hardware/software usage scenarios, deep penetration tests for security, circuit marginality tests, etc. Since the silicon executes at target clock speed (about a billion times faster than an RTL simulation), one can explore deep design states and find errors and vulnerabilities which could not have been possibly detected during presilicon activities. However, it also requires instrumentation of the design with a significant amount of additional circuitry, often referred to as design-for-debug (DfD) circuitry, to provide requisite observability and control during silicon execution. Unfortunately, instrumentation can also account for significant security vulnerabilities. In particular, it is tricky to determine whether an innocuous instrumentation for debug observability compromises some system-level policy. Furthermore, some of the DfD circuitry must remain enabled after postsilicon validation when the product is shipped to customers, e.g., for debugging problems discovered in the field. Many recent security hacks have made direct use of these debug features [42], [43]. Note that a viable solution to this problem is not to simply disable debug features involving sensitive assets. Postsilicon validation itself is also a highly critical activity performed under aggressive schedule, and requires planning that spans across the system lifecycle just like security does. Delays in postsilicon validation also has significant consequences, including the possibility of a company missing product release deadlines or even having to cancel the production, with consequent loss in revenues, reputation, and market share [9].

The tradeoff challenge between security and validation is the following. For postsilicon validation, we must observe design behavior during system execution; however, security policies on certain assets may disallow their observability. Put this way, the challenge may appear to be an instance of inconsistency between requirements from availability and confidentiality/integrity. The DfD architecture is, after all, a collection of IPs that need access to some internal data at different points of system execution; this need may be viewed as an availability requirement. Unfortunately, several factors make the tradeoff between security and debug more challenging than a general conflict between confidentiality and availability. Here we discuss some of the key factors.

A. Ambiguity

Observability requirements are rarely as clear cut as requirements arising from functionality. A key reason is that it is unclear *a priori* which component would exhibit a bug and therefore should be a target for observability. Furthermore, DfD decisions are made by validators and designers having little familiarity with security policies. When security constraints are imposed, often late in the design, one of the following two situations is likely: 1) some critical observability or control is inadvertently removed as a conservative measure; or 2) some subtle security flaw remains.

B. Feed-Through

Security requirements may affect observability indirectly. Consider a signal s in IP $\mathcal{A}$ that we wish to observe during postsilicon debug. Assume further that observing s does not compromise any security policy. However, in order for s to be observed, its value must be routed to an observation point, e.g., an output pin or system memory. If this route includes a high-security IP $\mathcal{B}$ then confidentiality requirement may cause $\mathcal{B}$ to be unobservable during system execution, thereby making s unobservable as well. On the other hand, the placement of IPs $\mathcal{A}$ and $\mathcal{B}$ in the system layout, and consequently, the route of signal s, may only be determined at an advanced stage of the design lifecycle making it impossible to account for that consideration when defining the signals to trace.

C. Lack of Centralization

Both DfD and security components are sprinkled across various IPs in the design. This, coupled with the lack of a rigorous documentation or specification of DfD requirements (and security policies), implies that it is often unclear what the purpose of a specific feature is, how it is excited, and what vulnerabilities it exposes. This makes it hard to determine security risks arising from DfDs.

In current industrial practice, the tradeoff is addressed typically by progressively increasing security features (and constraining DfD) as the design progresses along its lifecycle, from design to manufacturing, and production. Disabling DfD permanently is possible through blowing fuses during manufacturing and production. The situation is more complex for modern SoC designs, with the need to keep DfD features available for patching the product in the field. Nevertheless, the progressive increase is still a valid principle with a few adjustments. The first adjustment is that one cannot permanently disable DfD features because of the need to address this principle. Second, when such reversal is needed it is only for specific stakeholders with special authentication (e.g., an entity authorized to patch a design functionality). Finally, the reversal must be temporary, and once the activity needing the reversal (e.g., fixing an in-field bug) is complete, the system reverts to its default "higher security" state appropriate for the current phase of its design lifecycle.

A key problem in developing a comprehensive solution is that both security assurance and postsilicon validation are complex and elaborate processes, involving significant planning and a large number of stakeholders. Any solution to their tradeoff problem must address a large number of parameters. Below we highlight some of the key parameters. Obviously, no solution exists in current industrial practice, that addresses all of the following. In Section XI, we discuss one emergent architecture, which provides promise to address some of these considerations.

D. HVM Considerations

High-volume manufacturing test is the process of identifying manufacturing defects during production. This is done by placing the fabricated silicon in a tester, where it is exercised with a large number of tests. The test patterns are generated by accounting for the functional definition of the design, the target faults, a fault model, the fabrication process technology, etc. The accuracy of coverage from the results of these tests is highly sensitive to the test patterns being applied and the fidelity of the silicon design with respect to its presilicon netlist model. Consequently, irrespective of security constraints and access control restrictions, the test patterns must work the same way as much as possible on silicon designs as expected from presilicon models, and their results must be accurately observed. Furthermore, it must be possible to have a simple access to the IP being exercised with the test, without requiring too many workarounds.

E. Reusability

A key source of complexity in the current state of the practice discussed in Section VI is the need to manually identify assets and accesses for different products and usage scenarios. This job is highly tedious and error prone. Consequently, solution to the problem must provide a reusable infrastructure for systematically identifying and classifying assets and analyzing usage scenarios.

F. Late Variability

DfD is notorious for late changes in requirements and implementation. Indeed, DfD requirements can change during IP design, SoC integration, or even after a silicon step; the latter can happen on realization that observability or control of certain signals is critical for a future stepping. Consequently, any solution for addressing security challenges with DfD must be easy to adapt with such changing requirements. In particular, it should be possible to quickly validate an updated DfD architecture against a given set of security policies and identify vulnerabilities.

G. Self-Securability

It is obvious that any architecture introduced to address the security-validation tradeoff must be self-securing and must not introduce additional security backdoor (or complexity with debug).

H. Architecture

A decentralized architecture (both for security and DfD) is difficult to follow and can accidentally break or introduce vulnerability. To circumvent this possibility, it is critical that the architecture can be viewed as a centralized IP which can itself be effectively analyzed for possible violation of either security or debug requirements.

XI. EMERGENT TECHNOLOGIES

Given the disturbing recent trend of increasing security attacks on embedded, mobile, and IoT systems, there has been significant research interest to develop technologies for streamlining SoC security specification, architecture, and validation. There are efforts to develop security architectures beyond TEE definitions, integrating them with protocols and creating methods to identify access control at different points of the system execution [44]. There are efforts to extend formal verification technology for security validation, both among EDA vendor tools and through academic and industrial research [45]–[47]. There are also efforts on developing scalable, compositional theories for security assurance [48] and metrics for defining quality of security assurance. It is beyond the scope of this paper to review all the different approaches and the innovative technologies involved. Instead, in this section, we discuss two relatively extensive efforts undertaken by some of the authors themselves, and outline our thinking toward the a comprehensive solution. Note that the point is not to advocate these specific solutions but to provide our own take on the kind of thinking necessary to tame the complexity of security assurance in modern SoC designs.

A. A Centralized Policy Definition Architecture

Recent work [29], [49] has attempted to develop a centralized, flexible architecture called E-IIPS for implementing security policies in a disciplined manner. The idea is to provide an easy-to-integrate, scalable infrastructure IP that serves as a centralized resource for SoC designs to protect against diverse security threats at minimal design effort and hardware overhead. Fig. 7 shows the overall architecture of E-IIPS. It includes a microcontroller-based firmware-upgradable module called security policy controller (SPC) that realizes system-level security policies of various forms and types using firmware code following existing security policy languages. The SPC module interfaces with the constituent IP blocks in a SoC using "security wrappers" integrated with the IPs. These security wrappers extends the existing test (e.g., IEEE 1500 boundary scan based wrapper [50]) and debug wrapper (e.g., ARM's CoreSight interface [51]) of an IP. These security wrappers detect local events relevant to the implemented policies and enable communication with the centralized SPC module. The result is a

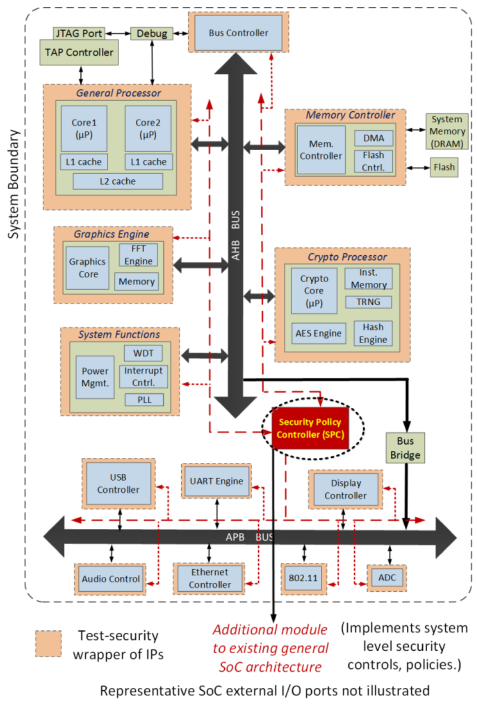

Fig. 7. ***SoC security architecture based on E-IIPS for efficient implementation of diverse security policies.***

flexible architecture and approach for implementing highly complex system-level security policies, including those involving interoperability requirements and trade-offs with debug, validation, and power management. The architecture is realizable with modest area and power overhead [29]. Furthermore, more recent work has shown that the existing design instrumentations, e.g., for DfD, could be exploited in implementing the architecture [49].

Of course, the architecture itself is only one component of the policy definition. Several challenges remain, e.g., 1) defining a language for security policy specification that can be efficiently compiled to SPC microcode; 2) study of bottlenecks related to routing and congestion across communication fabrics in implementing the architecture; 3) implementing security policies involving potentially malicious IPs (including malicious security wrappers or Trojans in the SPC itself), etc. Nevertheless, the approach shows a promising direction toward systematizing policy implementations. Furthermore, by enclosing the policy definitions to a centralized IP, it enables security validation to focus on a narrow component of the design, thereby potentially reducing validation time.

B. A Framework for Security Rule Check

Another critical problem is to define security rules to identify vulnerabilities. Recall that a significant cost in validation (e.g., penetration testing) comes from enumerating vulnerabilities. The vulnerabilities depend on several factors, e.g., the target market segment, supply chain, design and implementation technologies used, etc. Identifying these vulnerabilities early is critical: cost of fixing (or even finding) vulnerabilities at later stages of the lifecycle can be staggering.

This problem is being addressed by an emerging framework: design security rule check (DSeRC). The goal is to analyze vulnerabilities of a design and assess its security level at each design stage. The framework is shown in Fig. 8. Similar to design rule check (DRC), DSeRC will read the design files and user inputs, and check for vulnerabilities at all levels of abstraction. Each vulnerability is tied with metrics and rules so that each design's security can be quantitatively measured. At each level, the DSeRC framework will quantitatively analyze vulnerabilities in a design and provide feedback to the designer.

The first critical thrust for the development of DSeRC is to construct a comprehensive list of vulnerabilities present in SoC design. These include design mistakes, unclear or ambiguous specification, errors in CAD tools, design-for-test (DfT) and DfD insertion, etc. The second thrust is to incorporate each vulnerability with rules and metrics so that security of each design can be quantitatively measured. A sample vulnerability table was developed to present the list of vulnerabilities in SoC designs and their

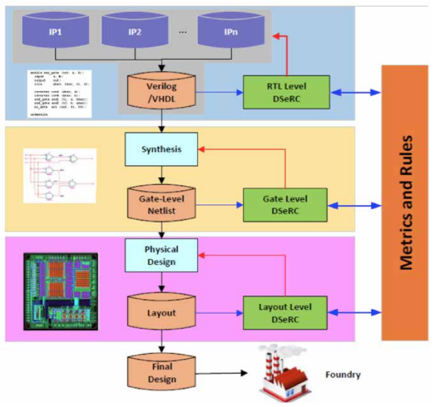

Fig. 8. ***Overall flow of DSeRC framework for design-time security evaluation.***

corresponding metrics and rules as well as additional vulnerabilities that needs to be addressed in the DSeRC framework [52], [53].

Note again that in spite of this progress the framework is still in its infancy. As we discussed earlier, system-level policies are complex and the security vulnerability detection performed during penetration testing is highly creative. Nevertheless, the direction shows promise and if successful, can significantly ameliorate validation challenges.

XII. CONCLUSION

We have presented, for the first time to our knowledge, a comprehensive overview of platform security assurance requirements, resilient architecture, and security validation in modern SoC designs. The goal has been to provide an understanding of the current state of the practice, highlight the key challenges, 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 in SoC security assurance is scary. The complexity involved is staggering and increasing at an alarming rate. On the other hand, we depend on human creativity to identify innovative attacks within a small time window before the system goes to field (and is exposed to attacks from the "bad guys")—an approach that cannot scale over the complexity we are encountering. While there are promising emergent approaches, we are far from creating trustworthy computing devices. There is a critical need to develop a disciplined approach to security assurance from the ground up. Perhaps more importantly, it may require a cooperative research involving different participants, viz., architects, designers, validators, and cross-cutting stakeholders such as power/performance architects and physical design engineers. ■

REFERENCES

[1] T. Fox-Brewster. *Voodoo Hackers: Stealing Secrets from Snowden's Favorite OS Is Easier Than you Think*. [Online]. Available: http://www.forbes.com/sites/thomasbrewster/2015/03/18/hacking-tails-with-rootkits/

[2] C. Kallenberg and X. Kovah, "How many million BIOSes would you like to infect?" in *Proc. 15th Annu. CanSecWest Conf. (CanSecWest)*, 2015.

[3] J. Loucaides and A. Furtak, "A new class of vulnerability in SMI handlers of BIOS/UEFI firmware," in *Proc. 15th Annu. CanSecWest Conf. (CanSecWest)*, 2015.

[4] R. Wojtczuk and C. Kallenberg, "Attacks on UEFI security," in *Proc. 15th Annu. CanSecWest Conf. (CanSecWest)*, 2015.

[5] V. Zimmer, "UEFI, open platforms and the defender's dilemma," in *Proc. 15th Annu. CanSecWest Conf. (CanSecWest)*, 2015.

[6] S. Ray and J. Bhadra, "Security challenges in mobile and IoT systems," in *Proc. 29th IEEE Int. Syst. Chip Conf.*, Sep. 2016, pp. 356–361.

[7] S. Ray, "System-on-chip security design for the Internet of Things," in *Proc. IEEE Custom Integr. Circuits Conf.*, 2017.

[8] D. Kleidermacher and M. Kleidermacher, *Embedded Systems Security: Practical Methods for Safe and Secure Software and Systems Development*. Amsterdam, The Netherlands: ELsevier, 2012.

[9] S. Yerramili, "Addressing Post-silicon Validation Challenge: Leverage Validation and Test Synergy," in *Proc. Int. Test Conf. (ITC)*, 2006.

[10] P. Patra, "On the cusp of a validation wall," *IEEE Design Test Comput.*, vol. 24, no. 2, pp. 193–196, Mar. 2007.

[11] G. Ramamoorthy. (2012). *Market Share Analysis: Semiconductor Design Intellectual Property, Worldwide*. [Online]. Available: https://www.gartner.com/doc/2403015/market-share-analysis-semiconductor-design

[12] E. Messmer. (2014). *RSA Security Attack Demo Deep-Fries Apple Mac Components*. [Online]. Available: http://www.networkworld.com/news/2014/022614-rsa-apple-attack-279212.html

[13] A. Nahiyan, K. Xiao, D. Forte, Y. Jin, and M. Tehranipoor, "AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," in *Proc. Design Autom. Conf. (DAC)*, 2016, pp. 1–6.

[14] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test Comput.*, vol. 27, no. 1, pp. 8–9, Jan. 2010.

[15] Y. Zhou, Y. Fang, and Y. Zhang, "Securing wireless sensor networks: A survey," *IEEE Commun. Surv. Tuts.*, vol. 10, no. 3, pp. 6–28, 3rd Quart., 2008.

[16] S. J. Greenwald, "Discussion topic: What is the old security paradigm," in *Proc. Workshop New Secur. Paradigms*, 1998, pp. 107–118.

[17] L. Davi, A.-R. Sadeghi, and M. Winandy, "Dynamic integrity measurement and attestation: Towards defense against return-oriented programming attacks," in *Proc. ACM Workshop Scalable Trusted Comput.*, 2009, pp. 49–54.

[18] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. 36th IEEE Symp. Security Privacy*, May 2015, pp. 745–762.

[19] P. C. Kocher and B. J. J. Jaffe, "Differential power analysis," in *Proc. 19th Annu. Int. Cryptol. Conf.*, 1999, pp. 398–412.

[20] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Proc. 16th Annu. Int. Cryptol. Conf.*, 1996, pp. 104–113.

[21] S. Ray, J. Yang, A. Basak, and S. Bhunia, "Correctness and security at odds: Post-silicon validation of modern SoC designs," in *Proc. 52nd Annu. Design Autom. Conf.*, 2015, p. 146.

[22] Trusted Computing Group. *Trusted Platform Module Specification*. [Online]. Available: http://www.trustedcomputinggroup.org/tpm-main-specification/

[23] A. Vasudevan, E. Owusu, Z. Zhou, J. Newsome, and J. M. McCune, "Trustworthy execution on mobile devices: What security properties can my mobile platform give me?" in *Trust Trustworthy Computing* (Lecture Notes in Computer Science), vol. 7344. Springer-Verlag, 2012, p. 150—178.

[24] J. M. McCune, B. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proc. ACM EuroSys*, 2008, pp. 315–328.

[25] Samsung. *Samsung KNOX*. [Online]. Available: http://www.samsungknox.com

[26] *Building a Secure System Using Trustzone Technology*, ARM Holdings, Cambridge, U.K., 2009.

[27] Intel. *Intel Software Guard Extensions Programming Reference*. [Online]. Available: https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf

[28] X. Li *et al.*, "Sapper: A language for hardware-level security policy enforcement," in *Proc. Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2014, pp. 97–112.

[29] A. Basak, S. Bhunia, and S. Ray, "A flexible architecture for systematic implementation of SoC security policies," in *Proc. 34th Int. Conf. Comput.-Aided Design*, Nov. 2015, pp. 536–543.

[30] L. Davi *et al.*, "HAFIX: Hardware assisted flow integrity extension," in *Proc. 52nd Annu. Design Autom. Conf.*, 2015, p. 74.

[31] L. Changlong, Z. Yiqiang, S. Yafeng, and G. Xingbo, "A system-on-chip bus architecture for hardware trojan protection in security chips," in *Proc. EDSSC*, Nov. 2011, pp. 1–2.

[32] *Microsoft Threat Modeling & Analysis Tool Version 3.0*, 2009.

[33] J. Srivatanakul, J. A. Clark, and F. Polac, "Effective security requirements analysis: HAZOPs and use cases," in *Proc. 7th Int. Conf. Inf. Secur.*, 2004, pp. 416–427.

[34] J. Bhadra, M. S. Abadir, L. Wang, and S. Ray, "A survey of hybrid technqiues for functional verification," *IEEE Design Test Comput.*, vol. 24, no. 2, pp. 112–122, Feb. 2007.

[35] A. Gupta, "Formal hardware verification methods: A survey," *Formal Methods Syst. Design*, vol. 2, no. 3, pp. 151–238, Oct. 1992.

[36] A. Takanen, J. D. DeMott, and C. Mille, *Fuzzing for Software Security Testing and Quality Assurance*. Norwood, MA, USA: Artech House, 2008.

[37] M. Corporation. (2015). *Microsoft Free Security Tools—Microsoft Baseline Security Analyzer*. [Online]. Available: https://blogs.microsoft.com/cybertrust/2012/10/22/microsoft-free-security-tools-microsoft-baseline-security-analyzer/

[38] F. Software. (2012). [Online]. Available: http://secunia.com

[39] S. Krstic, J. Yang, D. W. Palmer, R. B. Osborne, and E. Talmor, "Security of SoC firmware load protocol," in *Proc. IEEE HOST*, 2014.

[40] E. Love, Y. Jin, and Y. Makris, "Proof-carrying hardware intellectual property: A pathway to trusted module acquisition," *IEEE Trans. Inf. Forensics Security*, vol. 7, no. 1, pp. 25–40, Feb. 2012.

[41] W. Chen and J. Bhadra, "Striking a balance between SoC security and debug requirements," in *Proc. 29th IEEE Int. Syst. Chip Conf.*, Sep. 2016, pp. 368–373.

[42] Homebrew Development Wiki. *JTAG-Hack*. [Online]. Available: http://dev360.wikia.com/wiki/JTAG-Hack

[43] L. Greenemeier, "iPhone hacks annoy AT&T but are unlikely to bruise apple," *Sci. Amer.*, Sep. 2007.

[44] M. R. Sastry, I. T. Schoinas, and D. M. Cermak, "Method for enforcing resource access control in computer system," U.S. Patent 20120079590 A1, Mar. 29, 2012.

[45] E. W. Smith, "AXE: An automated formal equivalence checking tool for programs," Ph.D. dissertation, Stanford Univ., Stanford, CA, USA, 2011.

[46] *JasperGold Security Path Verification App*. [Online]. Available: https://www.cadence.com/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html

[47] R. Kannavara *et al.*, "Challenges and opportunities with concolic testing," in *Proc. NAECON*, Jun. 2015, pp. 374–378.

[48] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *J. Comput. Secur.*, vol. 18, no. 6, pp. 1157–1210, 2010.

[49] A. Basak, S. Bhunia, and S. Ray, "Exploiting design-for-debug for flexible SoC security architecture," in *Proc. IEEE DAC*, Jun. 2016, pp. 1–6.

[50] *IEEE Standard Test Access Port and Boundary Scan Architecture, IEEE Standards* 11491, 2001.

[51] E. Ashfield, I. Field, P. Harrod, S. Houlihane, W. Orme, and S. Woodhouse, "*Serial wire debug and the CoreSight debug and trace architecture*," 2006.

[52] H. Salmani and M. Tehranipoor, "Analyzing circuit vulnerability to hardware Trojan insertion at the behavioral level," in *Proc. Int. Symp. Defect Fault Tolerance VLSI Syst. (DFT)*, 2013, pp. 190–195.

[53] J. Lee, M. Tehranipoor, and J. Plusquellic, "A low-cost solution for protecting IPs against scan-based side-channel attacks," in *Proc. IEEE VLSI Test Symp.*, May 2006, p. 99.

ABOUT THE AUTHORS

Sandip Ray (Senior Member, IEEE) received the Ph.D. degree from The University of Texas at Austin, Austin, TX, USA.

He was a Research Scientist with the Intel Strategic CAD Laboratories, where he was involved in the presilicon and postsilicon validation of security and functional correctness of SoC designs, and design-for-security and design-for-debug architectures. He is a Senior Principal Engineer with NXP Semiconductors, where he leads research and development on security validation for automotive and Internet-of-Things applications. He is the author of three books (two upcoming) and over 60 publications in international journals and conferences. His research involves developing correct, dependable, secure, and trustworthy computing through cooperation of specification, synthesis, architecture, and validation technologies.

Dr. Ray has served as a Program Committee Member for over 40 international conferences, and as a Program Chair for the Formal Methods in Computer-Aided Design. He currently serves as an Associate Editor of the IEEE Transactions on Multi-Scale Computing Systems and Springer HaSS journals.

Eric Peeters received the M.E. degree in electromechanical engineering, the M.Sc. degree in electrical engineering, and the Ph.D. degree in electrical engineering from University of Louvain-la-Neuve, Belgium, in 2002, 2004, and 2006, respectively.

In 2006, he joined the group Thales Alenia Space ETCA in Belgium for one year. Then, in September 2007, he joined TI Germany in Freising (Munich) to work on the development of security products being evaluated through the difficult common criteria process (aiming EAL5+). In September 2010, he moved to TI headquarters in Dallas, TX, USA, where he has been heading the MCU Embedded security group since October 2011 as Security Architect and Manager.

Dr. Peeters has presented invited talks at VLSI Design Conference 2013 and he serves/has served on the technical program committees of various leading security conferences (mainly CHES and CARDIS).

Mark M. Tehranipoor (Senior Member, IEEE) received the Ph.D. degree from the University of Texas at Dallas, Richardson, TX, USA, in 2004.

He is currently the Intel Charles E. Young Preeminence Endowed Professor of Cybersecurity with the University of Florida, Gainesville, FL, USA. His current research projects include hardware security and trust, supply-chain security, VLSI design, and test and reliability. He has published over 300 journal articles and refereed conference papers and has given more than 150 invited talks and keynote addresses. He has published six books and 11 book chapters.

Dr. Tehranipoor is a Golden Core Member of the IEEE, and a member of ACM and ACM SIGDA. He was a recipient of several best paper awards, the 2008 IEEE Computer Society (CS) Meritorious Service Award, the 2012 IEEE CS Outstanding Contribution, the 2009 NSF CAREER Award, and the 2014 MURI Award. He serves on the program committee of more than a dozen leading conferences and workshops. He served as the Program Chair of the 2007 IEEE Defect-Based Testing (DBT) Workshop and the 2008 IEEE Defect and Data Driven Testing (D3T) Workshop, the Co-Program Chair of the 2008 International Symposium on Defect and Fault Tolerance in VLSI Systems (DFTS), the General Chair of D3T-2009 and DFTS-2009, and the Vice General Chair of NATW-2011. He cofounded the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), and served as the General Chair of HOST-2008 and HOST-2009. He serves as an Associate Editor of the *Journal of Electronic Testing: Theory and Applications,* the *Journal of Low Power Electronics,* the IEEE Transactions on Very Large Scale Integration Systems, and *ACM Transactions on Design Automation of Electronic Systems.* He served as the Founding Director of CHASE and CSI centers with the University of Connecticut.

Swarup Bhunia (Senior Member, IEEE) received the B.E. degree (honors) from Jadavpur University, Kolkata, India, the M.Tech. degree from the IIT Kharagpur, Kharagpur, India, and the Ph.D. degree from Purdue University, West Lafayette, IN, USA.

He is currently a Professor with the University of Florida, Gainesville, FL, USA. Earlier, he was appointed as the T. and A. Schroeder Associate Professor of Electrical Engineering and Computer Science with Case Western Reserve University, Cleveland, OH, USA. He has over ten years of research and development experience with over 200 publications in peer-reviewed journals and premier conferences. His research interests include hardware security and trust, adaptive nanocomputing, and novel test methodologies.

Dr. Bhunia received the IBM Faculty Award (2013), the National Science Foundation Career Development Award (2011), the Semiconductor Research Corporation Inventor Recognition Award (2009), and the SRC Technical Excellence Award (2005), and several best paper awards/nominations. He has been serving as an Associate Editor of the IEEE Transactions on CAD, the IEEE Transactions on Multi-Scale Computing Systems, and the *ACM Journal of Emerging Technologies, and the Journal of Low Power Electronics.* He has served as a Guest Editor of the *IEEE Design Test of Computers* (2010, 2013) and IEEE Journal on Emerging and Selected Topics in Circuits and Systems (2014). He has served as a Co-Program Chair of IEEE IMS3TW 2011, IEEE NANOARCH 2013, IEEE VDAT 2014, and IEEE HOST 2015, and in the program committee of many IEEE/ACM conferences.