Fuzz, Penetration, and AI Testing for SoC Security Verification: Challenges and Solutions

Kimia Zamiri Azar, Muhammad Monir Hossain, Arash Vafaei, Hasan Al Shaikh, Nurun N. Mondol, Fahim Rahman, Mark Tehranipoor, and Farimah Farahmandi

Abstract—The ever-increasing usage and application of system-on-chips (SoCs) has resulted in the tremendous modernization of these architectures. For a modern SoC design, with the inclusion of numerous complex and heterogeneous intellectual properties (IPs), and its privacy-preserving declaration, there exists a wide variety of highly sensitive assets. These assets must be protected from any unauthorized access and against a diverse set of attacks. Attacks for obtaining such assets could be accomplished through different sources, including malicious IPs, malicious or vulnerable firmware/software, unreliable and insecure interconnection and communication protocol, and side-channel vulnerabilities through power/performance profiles. Any unauthorized access to such highly sensitive assets may result in either a breach of company secrets for original equipment manufactures (OEM) or identity theft for the end-user. Unlike the enormous advances in functional testing and verification of the SoC architecture, security verification is still on the rise, and little endeavor has been carried out by academia and industry. Unfortunately, there exists a huge gap between the modernization of the SoC architectures and their security verification approaches. With the lack of automated SoC security verification in modern electronic design automation (EDA) tools, we provide a comprehensive overview of the requirements that must be realized as the fundamentals of the SoC security verification process in this paper. By reviewing these requirements, including the creation of a unified language for SoC security verification, the definition of security policies, formulation of the security verification, etc., we put forward a realization of the utilization of self-refinement techniques, such as fuzz, penetration, and AI testing, for security verification purposes. We evaluate all the challenges and resolution possibilities, and we provide the potential approaches for the realization of SoC security verification via these self-refinement techniques.

Index Terms—SoC Security Verification, Fuzzing, Penetration Testing, AI-based Testing.

1 Introduction

With the ever-increasing adoption and utilization of the integrated circuit (IC) supply chain horizontal model through the last decade, a wide range of parties and entities have to be involved to contribute and accomplish part(s) of the various stages of design, fabrication, testing, packaging, and integration, which results in forming expansive globalization and distribution of the semiconductor supply chain. Globalization by outsourcing significantly reduces the logistic/manufacturing cost and time-to-market, and allows companies access to advanced nodes, specialized resources, and cutting-edge technologies [1]. However, the main impact of globalization is the loss of control/trust that raises a wide range of threats in the supply chain, including IP piracy, IC overproduction, malicious functionality insertion, etc., that can induce catastrophic consequences with huge financial/reputation loss [2].

Over the past decade, as demonstrated in Fig. 1, the same trend has been witnessed for (i) the involvement of numerous entities, (ii) complexity and heterogeneity of modern SoCs, (iii) re-usage of existing third-party IPs, (iv) the security threats variety, and (v) spending growth against the threats. As demonstrated, this trend faces a significantly higher rate in recent years showing why urgent action is required for automation of the SoC security verification in the current time frame. While the SoC designs and architectures are getting larger, more complex, and more heterogeneous, with a variety of IPs that have several highly sensitive assets such as encryption keys, device configurations, application firmware (FW), e-fuses, one-time programmable memories, and on-device protected data, guaranteeing the security of these assets against any unauthorized access or any attack is paramount. As demonstrated in Fig. 2, a typical SoC architecture comes with numerous security IPs, such as cryptographic cores (encryption/decryption), True Random Number Generator (TRNG) modules, Physical Unclonable Function ((PUF) units, one-time memory blocks, etc. With either generation, propagation, or usage of the assets by these IPs, the distribution of these security assets may happen across the SoC architecture, and since some of these IPs are from third-party vendors, any unauthorized or malicious access to these assets can result in company trade secrets for device manufacturers or content providers being leaked.

The involvement of third-party vendors, which compromises the trustworthiness of the entire SoC, is not the only source of vulnerability in the SoC architecture. Security vulnerabilities can also emerge as the consequence of design/implementation/integration drawbacks through different stages of the SoC design flow, which makes the designing of a secure SoC more challenging. These vulnerabilities can also emerge in different forms, such as information leakage, access control violation, side-channel or covert channel leakage, presence or insertion of malicious functions, exploiting test and debug structure, and fault-injection-based attacks [3], [4], [5]. Some of these security

• *K. Z. Azar, M. M. Hossain, A. Vafaei, H. Al Shaikh, N. N. Mondol, F. Rahman, M. Tehranipoor, and F. Farahmandi are with the Department of ECE, University of Florida, Gainesville, FL, USA, 32611.*
E-mail Addresses: {k.zamiriazar, hossainm, arash.vafaei, hasanalshaikh, nmondol}@ufl.edu, {fahimrahman, tehranipoor, farimah}@ece.ufl.edu

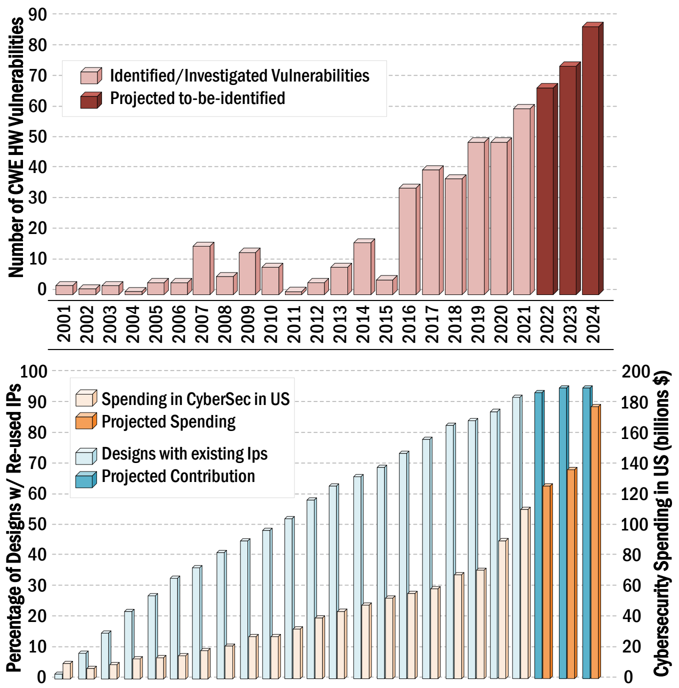

Fig. 1: IP Re-use vs. Threats vs. Security-related Spending.

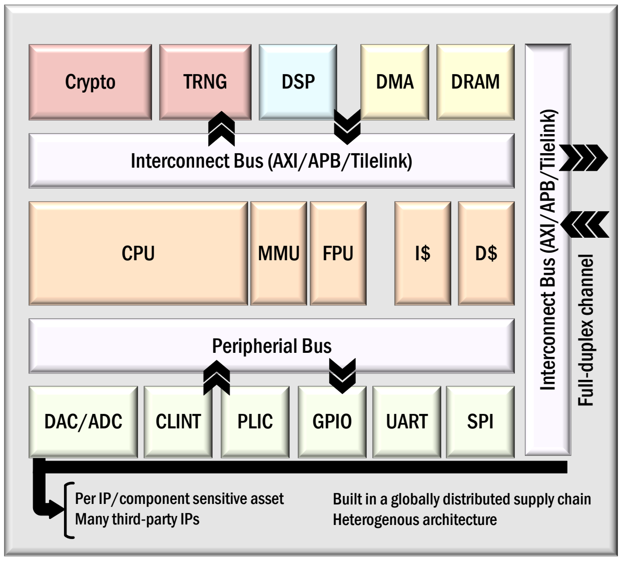
Crypto
TRNG
DSP
DMA
DRAM
Interconnect Bus (AXI/APB/Tilelink)
CPU
MMU
FPU
I$
D$
Peripherial Bus
DAC/ADC
CLINT
PLIC
GPIO
UART
SPI
Interconnect Bus (AXI/APB/Tilelink)
Full-duplex channel
Per IP/component sensitive asset
Many third-party IPs
Built in a globally distributed supply chain
Heterogenous architecture

Fig. 2: An SoC Design with the Integration of a wide variety of IPs.

vulnerabilities may be introduced unintentionally by a designer during the transformation of a design from specification to implementation. This is because the designers at the design house spend most of their design, implementation, and integration efforts to meet area, power, and performance criteria. For verification, they mostly focus on functional correctness [6]. Additionally, the flaws in the computer-aided design (CAD) tools can unintentionally result in the emergence of additional vulnerabilities in SoCs. Moreover, rogue insiders in the design houses can intentionally perform malicious design modifications that create a backdoor to the design. An attacker can utilize the backdoor to leak critical information, alter functionalities, and control the system.

It should be noted that many of the existing security countermeasures introduced in the literature or widely used in the industry, such as hardware obfuscation, watermarking, metering, camouflaging, etc. [7], [8], [9], [10], [11], [12], have nothing to do with such SoC-level vulnerabilities, as many of these security vulnerabilities originate precisely from unexpected interactions between layers and components, and traditional techniques fail at catching these cross-layer problems or do not scale to real-world designs. Therefore, apart from the existing hardware security countermeasures that might be applied to the design, the security verification is required to be evaluated meticulously, particularly for the SoC-level vulnerabilities.

Considering that the different IP components of a SoC have their own highly sensitive security assets that should be accessed/exploited by some other components, most system design specifications include a (limited) number of security policies that define access constraints and permissions to these assets at different phases during the system execution. As SoC complexity continues to grow and time-to-market shrinks, verification of these security policies has become even more difficult to ensure that the design under investigation does not violate these policies, whose violation can lead to the emergence of vulnerabilities or backdoors for attackers. It might not be beyond the possibility that the definition of a set of well-established security properties can help verification engineers alleviate the problem. However, some of these policies are related to different phases of the design flow, from specification to implementation, and they might involve the design house, different IP vendors, and the integration team. Additionally, the realization of these policies may require a multi-layer implementation through a combination of hardware, firmware, and software in the SoC. Moreover, the definition of these policies might face significant changes or refinements across the design flow, which makes many of them invalidated.

The definition of the security policies and their expansion is also entirely dependent on the pre-defined and pre-assessed scenarios and test cases that consist of prohibited and illegal actions. For SoCs becoming larger and more complex with larger sets of IPs and components integrated into it, the process of detection, gathering, and building all these test-cases that lead to the definition of security policies are becoming worse. Hence, *unknown* security vulnerabilities will still appear in SoC transactions, leading to breaches of confidentiality, integrity, or authenticity, despite verification being properly performed based on the known security policies. Additionally, due to the lack of reciprocal trust between different entities involved in SoC design and implementation, or based on the time of security verification (i.e., pre- or post-verification), the access of the security verification engine/tool will vary to the system, from full access with knowledge about all internal operations, wires, registers, etc., to *NO* access to the internal knowledge of the system. The access differs case by case; however, in all cases, it will affect the outcome of security verification, in terms of security policies conformance, performance, the complexity of security verification flow, etc.

A literature review on the software testing domain reveals that the procedure of software testing has been suffering for almost two decades from the same challenges and difficulties. Software testing is the main way of verifying software, from specification to release, against the defined requirements and accounts for about half of the cost and time of development [13], [14]. For instance, considering the main objective of software testing and verification, which is systematically evaluating the software, in carefully controlled circumstances, the scope of software testing is heavily dependent on the knowledge and internal access, which categorizes the software testing into three main breeds, namely (i) white box with full knowledge of code, (ii) black box with no knowledge relevant to the internal structure of the code, and (iii) gray box with limited knowledge of the internal structure. In this case, the automation of verification depends on the breadth of testing. Recently, in software testing, with the emergence of automated and semi-automated techniques, like the usage of self-guidance and self-refinement concepts (e.g., artificial intelligence, machine learning, the mutation-based approaches like fuzzing [15], the testing procedure has been evolved tremendously in this domain. These techniques are highly successful in detecting software vulnerabilities since they are automated, scalable to large codebases, do not require the knowledge of the underlying system, and are highly efficient in detecting many security vulnerabilities.

The utilization of existing techniques for security verification of modern SoCs is mainly limited to the expert review, and they do not provide acceptable scalability [16]. The top view of such techniques has been demonstrated in Fig. 3. In such solutions, the conventional formal verification techniques, in spite of significant recent advances in automated formal technologies (functional verification) such as satisfiability (SAT) checking and satisfiability modulo theories (SMT), that is also widely used for the evalution of IP protection techniques (hardware obfuscation) [17], [18], [19], cannot promise the desired scalability for the security verification of modern SoCs, and the gap between the scale and complexity of modern SoC designs and those which can be handled by formal verification techniques has continued to grow. Similarly, symbolic execution and model checking [20] are suffering from scalability for verification at the SoC level. As an instance, the work in [3], [21] proposed a technique to analyze the vulnerabilities of FSM, called AVFSM. However, apart from the FSM, a SoC contains other modules (exception handlers, test and debug infrastructures, random number generators, etc.) which must be inspected during security verification. Authors in [22] provided a method to write the security-critical properties of the processor. They found that the quality of security properties is as good as the developer's knowledge and experience. Moreover, there is a lack of comprehensive threat model definition, which must be considered while developing security properties. There are some other approaches which have developed security properties and metrics by considering only a small subset of vulnerabilities (e.g., the vulnerability in hardware crypto-systems [23], side-channel vulnerabilities [24], [25] and abstraction level limitations like the behavioral model [26]).

Also, since many of the security vulnerabilities in the SoC originate precisely from unexpected interaction between layers and components, identifying novel methodologies is hard because researchers often do not have enough access to all parts of the system, which is particularly true for proprietary hardware micro-architectures. Authors in [27] demonstrate how commercially available tools can fail to detect security-relevant RTL bugs including some that originate from cross-modular and cross-layer interactions. The work in [28], presented a methodology to infer security-critical properties from known processor errata. However, manufacturers only release errata documents enumerating known bugs. Unknown security vulnerabilities can exist in a SoC design that are not listed in available errata documents. Another approach for finding security bugs is information flow tracking (IFT) techniques. The authors in [29], [30], [31], [32], [33] utilize IFT and statistical models to map hardware security properties. However, this technique requires design instrumentation and tainting all the input variables, which require more computational time and memory resources. Hence, IFT and statistical modeling, which requires expert knowledge of the design, become more complex with increasing design complexity. There is an increasing need for methodologies and frameworks for security verification of modern SoCs that are scalable to large and complex designs, highly automatic, effective, and efficient in detecting security-critical vulnerabilities.

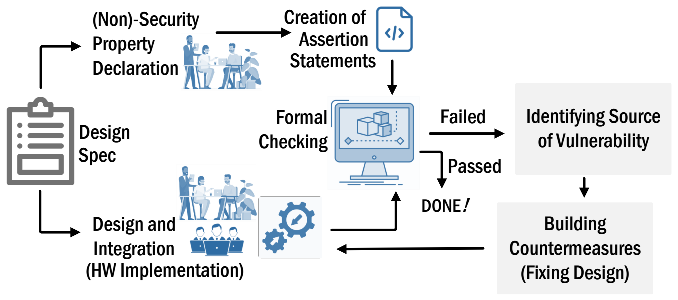
(Non)-Security Property Declaration
Creation of Assertion Statements
Design Spec
Formal Checking
Failed
Identifying Source of Vulnerability
Passed
DONE!
Design and Integration (HW Implementation)
Building Countermeasures (Fixing Design)

Fig. 3: Property-Driven Functional Verification.

Based on the trend of software verification testing techniques and their efficacy for the evaluation of software specifications and requirements, it is evident that the same but futuristic trend will Potentially happen in the area of SoC security verification. However, there definitely exist numerous limitations and challenges, in terms of the concept migration, implementation, assumptions, metrics, and the outcome. Hence, with such a gap, and due to notable lack of detailed and comprehensive evaluation on SoC security verification, in this paper, we will examine and re-evaluate the principles and fundamentals of SoC security verification through automated and semi-automated architectures. Moving forward, with the ever-increasing complexity and size of SoCs and with the contribution of more and more IPs and less and less trustworthiness between components, SoC security verification through a more closed environment, like the gray and black box model, will get more attention. Hence, in this paper, with more focus on such models, and by trying to get the benefit of semi-automated or automated approaches, like AI or ML, fuzz testing, and penetration testing, we provide a comprehensive overview of SoC security verification as follows:

1) We first identify the source of vulnerabilities indi-

cating the necessity of an automated verification framework for the SoC security verification.

2) We then define the assumptions for SoC security verification based on different factors like the designer's desire, followed by the requirements of the framework, such as scalability, high coverage, etc.
3) We examine the possibility of engaging software approaches, with more specific concentration on self-guided or self-refinement approaches, such as fuzz, penetration, and AI testing for SoC security verification.
4) We discuss about the future research directions and challenges for implementing an automated verification framework to identify security vulnerabilities based on the self-refinement approaches.

The rest of the paper is organized as follows. Section 2 provides the source of SoC security vulnerabilities at different stages of an SoC lifecycle. Section 3 will review the terminologies that are directly and indirectly related to the SoC security verification. Then, by reviewing the challenges for SoC security verification in Section 4, we will define the main assumptions required to be considered through SoC verification in Section 5, which is followed by the description of SoC security verification flow in Section 6. Then, the models for the usage of fuzz testing, penetration testing, and AI testing will be covered in Sections 7-9. We elaborate upon the future possible research directions and the existing challenges for these self-refinement approaches in Section 10, and we conclude the paper in Section 11.

2 SoC Security: Source of Vulnerabilities

Fig. 4 firstly shows the main steps of a modern IC supply chain which is plunged in globalization with the involvement of multiple IPs. As also demonstrated in Fig. 4, an SoC design can encounter security vulnerabilities during different stages of its design and life cycle, each is excited from a unique source. Beginning from the very early stages of the IC design to its fabrication, the following are the main sources of such security vulnerabilities in the SoC design and implementation:

(**SV1**) *Inadvertent designer mistakes*: The development of multiple IPs/components may be distributed between different design teams as well as third-party IPs. This can result in a non-clear definition over the interaction between these IPs, non-sophisticated exception handling for inter-component communications, limited knowledge about the behavior of neighboring components, (communication) protocol malfunctioning, and lack of understanding of security problems due to the high complexity of the designs and variety of assets. Hence, it may result in different forms of vulnerabilities, such as secret information leakage or losing the reliability of the SoC.

(**SV2**) *Rogue employee (insider threat)*: Unlike (**SV1**), deliberate malfunction can be invoked by the rogue employee(s) that pose significant security threats to the security of the whole SoC.

(**SV3**) *Untrusted third-party IP vendors*: Similar to (**SV2**), with violation of rules/protocols of communication between their IPs and other components, or unauthorized interactions with no monitoring, third-party IPs can pose the same security issues.

(**SV4**) *EDA optimizations*: Almost all efforts in the development and improvement of CAD tools have been directed to optimize and increase the efficiency of synthesis, floorplanning, placement, and routing in terms of area, power, throughput, delay, and performance. These CAD tools are not well-equipped with the understanding of the security vulnerabilities [21] of integrated circuits (ICs) and can, therefore, introduce additional vulnerabilities in the design. As an instance, the CAD tools can potentially but unintentionally merge trusted blocks with the untrusted ones, or move a security asset to an untrusted observable point, which opens the possibility of different attacks like eavesdropping on the side-channel analysis.

(**SV5**) *Security compromising stages of IC supply chain*: For modern SoCs, design-for-test (DFT) and design-for-debug (DFD) are designed to increase the observability and controllability for post-silicon verification and debug efforts. However, increasing the observability and preserving security are two conflicting factors. Test and debug infrastructures may create confidentiality and integrity violations. For example, in [26], scan chains have been exploited to leak security-critical information, and numerous studies through the last decade have evaluated the utilization of DFT/DFD infrastructure for attacking the design [10]. Similar to (**SV4**), many of these vulnerabilities are emerged unintentionally due to the accomplishment of different stages of the IC design flow.

(**SV6**) *Impact of hardware Trojan insertion*: This case is a derivation of (**SV2**) and (**SV3**), in which SoC designs are also prone to many maliciously introduced threats, such as hardware Trojans. These hardware Trojans can be inserted by untrusted entities involved in a number of design and supply-chain stages including third-party IP (3PIP) vendors and rogue in-house employees causing sensitive information leakage, denial-of-service, reduction in reliability, etc. in the SoC. Also, insider threats are particularly dangerous since they have full observability and access to the whole design and source files. When a chip is deployed into the final design, and Trojan was inserted during stages, an attacker can monitor physical characteristics of a design (such as delays, power consumption, transient current leakage) to recover secret information.

(**SV7**) *Lack of trustworthiness in EDA Tools*: Although it is mostly assumed that the CAD tools are trusted nodes within the IC design flow, modernizing SoC design infrastructure with compatibility for cloud-based design development, and leveraging fully distributed processes with remote EDA tools violates this assumption. In such cases, software tools cannot be considered trusted anymore (the current assumption is that even for cloud-based systems, it is just assumed that the cloud infrastructure is secure, which is not always correct.), and the SoC designs are accordingly prone again to numerous malicious threats.

(**SV8**) *Untrusted fabrication site*: The consequence of this case is similar to (**SV7**). In this case, the fab site as an untrusted entity, with having full knowledge about the layout of the design, can apply manipulation before the fabrication for further usage after the fabrication.

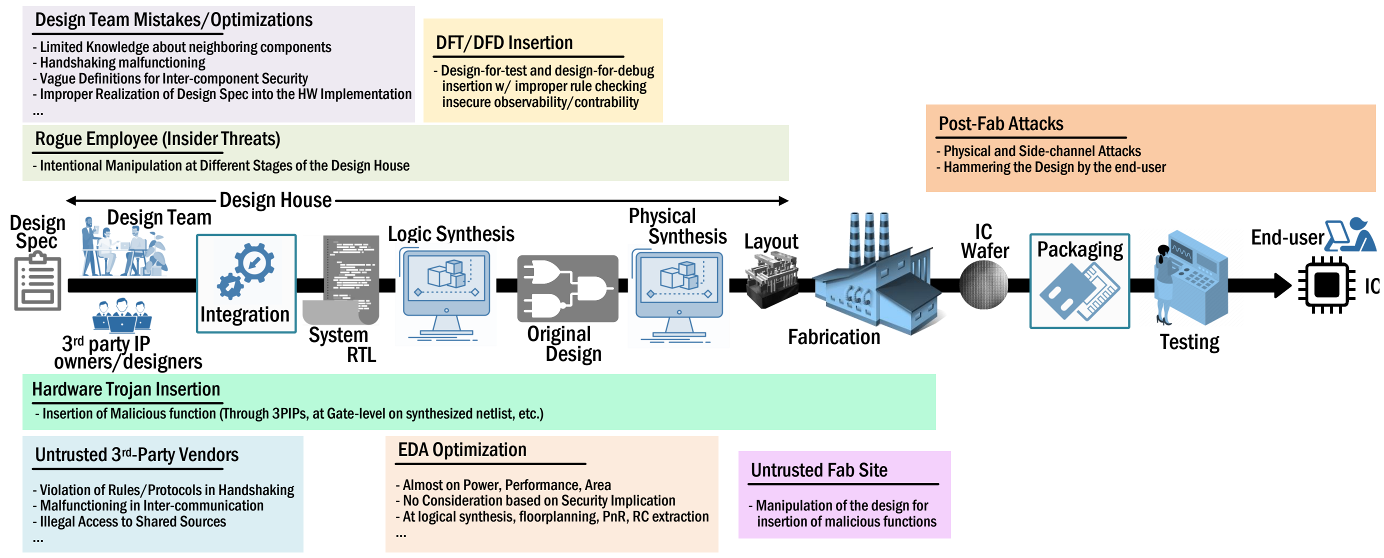
Design Team Mistakes/Optimizations
- Limited Knowledge about neighboring components
- Handshaking malfunctioning
- Vague Definitions for Inter-component Security
- Improper Realization of Design Spec into the HW Implementation
...
DFT/DFD Insertion
- Design-for-test and design-for-debug insertion w/ improper rule checking insecure observability/contrability
Rogue Employee (Insider Threats)
- Intentional Manipulation at Different Stages of the Design House
Post-Fab Attacks
- Physical and Side-channel Attacks
- Hammering the Design by the end-user
Design House
Design Spec
Design Team
Integration
3rd party IP owners/designers
System RTL
Logic Synthesis
Original Design
Physical Synthesis
Layout
Fabrication
IC Wafer
Packaging
Testing
End-user
IC
Hardware Trojan Insertion
- Insertion of Malicious function (Through 3PIPs, at Gate-level on synthesized netlist, etc.)
Untrusted 3rd-Party Vendors
- Violation of Rules/Protocols in Handshaking
- Malfunctioning in Inter-communication
- Illegal Access to Shared Sources
...
EDA Optimization
- Almost on Power, Performance, Area
- No Consideration based on Security Implication
- At logical synthesis, floorplanning, PnR, RC extraction
...
Untrusted Fab Site
- Manipulation of the design for insertion of malicious functions

Fig. 4: Potential Threats within SoC Design Flow.

(**SV9**) *Efforts on SoC design optimization*: Most of the designers are always concentrating on minimizing the overhead in terms of area, power, performance. However, in many of these optimization cases done by the designer, the outcome can possibly incur forms of security issues, such as sharing memory banks between non-trusting processes in multi-tenanted systems that were the main consequence of spectre and meltdown attacks [34], [35].

(**SV10**) *Hammering by the end-user*: This group of threats can be related to either logical or physical attributes and specification of the components integrated into the SoC, such as memory. Similar to rowhammer attack on DRAMs, logic-level hammering and repeating of different sequences of action(s) around the targeted component might lead to the reveal of some information, and then by repeating the scenario (sequence of actions), some form(s) of information breach can happen in the SoC.

Table 1 provides a top view of these sources of vulnerabilities and their characteristics. It clearly shows that there is a critical need to verify the security of the SoC at each of the stages and verify the trustworthiness of an SoC. However, if the vulnerabilities reach the post-silicon stage, there would be limited flexibility (almost none) in changing or fixing them. Moreover, the cost of fixing the design is significantly higher as we advance through the later stages of the design. Furthermore, vulnerabilities that reach the manufacturing stage will cause revenue loss. Therefore, it is essential to develop efficient security verification approaches to ensure the security and trustworthiness of SoC designs with more concentration at the pre-silicon stage.

3 SoC Security Verification: Terminology

The main aim of this section is to provide the basics and principles around the concept of SoC security verification. For this purpose, we comprehensively review the main terminologies that are directly related to this trend.

3.1 Security Assets in SoC

For any IP component, firmware, software, etc., involved and integrated into the SoC, there exists a set of information or data, whose leakage can lead to catastrophic consequences with huge financial/reputation loss. This sensitive information or data are known as security-critical values, a.k.a. *security assets*, that must be protected against any potential form of threat. Any successful retrieval of such information or data in an illegal way might result in trade secret(s) loss for device manufacturers or content providers, identity theft, and even destruction of human life. These assets are usually known to the designers, and they are defined based on the specifications of the design. As an instance, encryption/decryption or private key in cryptographic primitives are assets, and the location and usage of them are known for the designers through the SoC design and implementation [36]. The following gives us some insight about the main primitives in an SoC that must be considered as the security assets:

(**SA1**) **On-device key**: (*Secret/Private key(s) of an encryption algorithm*) These assets are stored on a chip mostly in some form of non-volatile memory. If these are breached, then the confidentiality requirement of the device will be compromised.

(**SA2**) **Manufacture firmware**: (*Low level program instructions, proprietary firmware, protected states of the controller(s)*) These assets may have intellectual property and system-level configuration values and compromising these assets would allow an attacker to counterfeit the device.

(**SA3**) **On-device protected data**: (*Sensitive user data + meter reading*) Leakage of these assets is more related to identity theft, and an attacker can invade someone's privacy by stealing these assets or can benefit himself/herself by tampering these assets.

(**SA4**) **Device configuration**: (*Service/Resource access configuration*) These assets determine which particular services or resources are available to a particular user and an attacker may want to tamper these assets to gain illegal access to the resources.

TABLE 1: Overview of Source of SoC Security Vulnerabilities.

Category	Source of Vulnerability	Design Flow Stage	Examples of Vulnerabilities	Threat Model
(SV1)	Inadvertent designer mistakes	software, firmware, boot loader, register file, cache, register-transfer (RT) level, high-level (HLL)	(1) Insecure implementation of controller circuit (FSM) or boot loader, (2) Incorrect synchronization or protocol handshaking between IPs (master/slave), (3) Incorrect mutual exclusion of write/execute operation leading to illegal access.	Insecure boot mode, Inter-IP illegal accesses, Information breach, malfunctioning of security operations
(SV2)	Rogue employee at design house (insider threat)	software, firmware, boot loader, register file, cache, RTL, HLL	(1) manipulating hardware, firmware, software, that facilitates obtaining the security assets after fabrication, including logic analyzer module insertion, disabling security permissions/policies, etc.	Insecure boot mode, Inter-IP illegal accesses, Information breach, malfunctioning of security operations
(SV3)	Untrusted third-party IP vendors	HLL, RTL, Gate-level	(1) Continuous watching/monitoring the bus for obtaining information by the IP, (2) Incorrect protocol handshaking that leading to illegal actions (illegal memory write/read).	Inter-IP Illegal actions, By-passing IP-based checks (security checks)
(SV4)	EDA optimizations	logical or physical synthesis flow	(1) Insecure optimization of the design, such as sharing (merging between trusted and untrusted region), insecure control flow, insecure data flow.	Information breach, Inter-IP illegal accesses, Flaws in security policies implementation
(SV5)	Security compromising stages of IC supply chain	Gate-level, design service provider (DFT/DFD)	(1) Opening backdoor for attacks through test/debug infrastructures, (2) Reading internal values of the design	Information breach
(SV6)	Impact of hardware Trojan insertion	Gate-level	(1) manipulating hardware that facilitates obtaining the security assets after fabrication, (2) Insertion of Trojan for malfunctioning	Information breach, Inter-IP privacy violation, Malfunctioning
(SV7)	Lack of trustworthiness in EDA Tools	logical and physical synthesis flow	**(SV2)** + **(SV6)**	**(SV2)** + **(SV6)**
(SV8)	Untrusted fabrication site	GDSII at fabrication site	**(SV6)**	**(SV6)**
(SV9)	Efforts on SoC design optimization	RTL, HLL	(1) Incorrect optimization with open corner cases that leads to security vulnerabilities	Information breach, inter-IP illegal accesses, insecure protocol implementation
(SV10)	Hammering by the end-user	Post-silicon over the fabricated chip	(1) applying continuous and repeating tests on specific target based on physical or logical reflects	Information breach

(**SA5**) **Entropy**: (*Random numbers generated for cryptographic primitives*), These assets are directly related to the strength of cryptographic algorithms embedded into the SoC, e.g., initializing vector or cryptographic key generation. Successful attacks on these assets would weaken the cryptographic strength of a device.

Choice of security assets varies design by design and abstraction layer by an abstraction layer. Mainly, the declaration of security assets is heavily dependent on the *security policies* that is defined by the designers of different component integrated into the SoC. Hence, apart from these general security assets listed here, which are known to the hardware designers, the security assets begin to expand within the SoC due to the interaction of different IPs. Consequently, with the increase of the list of security assets, SoC security verification through traditional methodologies, e.g., formal-based and satisfiability-based approaches, becomes almost impractical.

3.2 Security Policies/Properties in SoC

Based on the source of vulnerabilities, the threat model per each source, and security assets defined for the design under investigation, a set of requirements will be defined, whose realization will assist the design team to guarantee the protection of the security assets against the given threat models. Different threat model categorization can be evaluated for SoC verification, such as (i) overwriting or manipulating confidential data by the unauthorized entity (integrity violation), (ii) unauthorized disclosure of confidential information/data (confidentiality violation), and (iii) malfunctioning or disruption of function/connectivity of a module (availability violation) [37].

For SoC security verification, (*security policies/properties* define a mapping between the requirements and some design constraints. Then, to fulfill the desired protection level, these constraints must be met by building some infrastructures, and these infrastructures are built by the IP design team(s) or SoC integration team. The definition of security policies/properties is dependent on different actions/behaviors located at multiple stages or abstraction layers, and they might also be updated or refined through various stages [38], [39]. As an instance, the requirement defined as: *switching the chip from functional to test mode should not leak any information related to the security assets*, in a typical SoC can be mapped to constraint (policy/property) defined as *An asynchronous reset signal assertion for scan chain is required for secret/private key registers while the chip's mode is switching from the functional to the test mode*.

Per each requirement, for mapping to a security policy/property, details and underlying conditions must be considered meticulously, and then these conditions/constraints must be met to guarantee the protection of the asset(s). It is evident that the definition of security policies/properties may vary depending on multiple factors, like architecture and components of the SoC, interconnections and bus interface, state of the execution (e.g., boot time, normal execution, test time), and the state in the development life-cycle (e.g., manufacturing, production, test/debug), etc. Below we provide a categorization for general policies that are required to be considered for security purposes in SoCs. This categorization covers both system-level and lower-level security policies/properties.

(**SP1**) *Definition of access restriction policies*: This set of policies/properties is one of the main requirements that define the constraints regarding how different components, either hardware, framework, or software, can access security assets. The definition of access restricting policies/properties directly/indirectly affects other policies/properties as well. For instance, given specific restricted access will change the data flow or control flow in the SoC. So, other policies/properties can be met/violated based on a newly defined access restriction policy/property.

(**SP2**) *Definition of data/control flow restricting policies*: In many cases, particularly security assets related to cryptographic primitives, By observing the responses of the components under investigation to a sequence of actions, the security assets can be retrieved with no direct access. In such cases, even though (**SP1**) has been defined and established properly, since there exists an insecure form of data/control flow, the confidentiality would be violated. Unlike (**SP1**), the realization of this group of policies/properties requires highly sophisticated protection techniques with advanced formulation/model. So, to keep the complexity of security policies at a reasonable degree, this policy/property is more efficient to be employed for high-critical assets with high confidentiality requirements.

(**SP3**) *Definition of HALT/OTS/DOS restricting policies*: This policy/property indicates the liveness of components throughout the execution of different operations. This mostly can be done by checking the status signals per each request for the component, to make sure that there is no halt, out-of-service (OTS), or denial-of-service (DOS) that violates the system availability requirements. Policies/properties related to the malfunctioning can also be part of availability violation (as the component does not provide the correct functionality or correct timing behavior).

(**SP4**) *Definition of insecure sequence execution restricting policies*: An authorization always is required once a component needs to get access to a security asset in the SoC. However, the flow between the authorization and getting access must be flawless with no possibility of changes that invalidates the access control. One of the examples for this group of policies is time-of-check to time-of-use (TOCTOU), which shows in the middle of authorization and usage, the changes of the state of the security asset can lead to some invalid/illegal actions which are happening only when the resource is in an unexpected state.

(**SP5**) *Definition of insecure boot restricting policies*: The process of the boot may involve multiple critical security assets, including the definition of access restriction policies, cryptographic and on-device keys, firmware, etc., and any security leakage can lead to multiple vulnerabilities at different layers. Policies for protecting the boot can be defined individually related to (**SP1-4**) or unified on a set of actions/requirements.

(**SP6**) *Definition of inter-component integrity policies*: This policy is more likely low-level, i.e., IP-level inter-communication, showing that any (secure) communication between two components must be kept intact and with no changes done by a third component.

(**SP7**) *Definition of inter-component confidentiality policies*: Similar to (**SP6**), at low-level, any inter-communication between two components must be kept fully secure and confidential between these two components, and there should be no possibility for any third component to receive any part of inter-component communicated data.

(**SP8**) *Definition of inter-component authenticity policies*: Another low-level policy that verify the authenticity of the component requesting a security asset.

The definition of security assets, security policies, and building a clear relationship between them are indispensable preliminary steps of SoC security verification. In the context of security verification, a security policy/property must be a complete statement that can check assumptions, conditions, and expected behaviors of a design [40], which is more likely a formal description of the design behavior/specification. The coverage of security policies/properties can be considered as a metric for the security assessment of an SoC, and the violation of the policies/properties implies that the design should be fixed.

3.3 Security Policy/Property Languages in SoC

To build the automated SoC security verification, a straightforward constraint definition is required to verify that based on the specification of the security policies/properties, the design adheres to those properties. To meet such requirements, there should be a unified language for the declaration of the security properties, and the security verification framework must be able to convert this language to hardware implementation and testing. Due to the dynamic nature of the threat model, the language must be rich and amenable to be expanded as needed and to specify characterizations like sensitivity levels, affectability, and observability. As an instance, security policies/properties determine the secure vs. insecure region(s), and accordingly, the security language must be able to check policies like confidentiality, as the sensitive data from the secure region should not leak to any insecure region.

These days, designers mostly use one of the powerful assertion languages such as *property specification language* (PSL) [41] and *SystemVerilog Assertions* (SVA) [42] to describe interesting behavioral events of a design. These languages use temporal logic representations such as Linear Temporal Logic (LTL) [43] and Computational Tree Logic (CTL) [44]. Languages based on LTL and CTL usually describe design behaviors and properties in four layers: Boolean expression, sequence, property specification, and assertion directive layers. These layers can be used on top of different HDL languages including Verilog and VHDL.

3.4 Pre-silicon vs. Post-silicon Verification in SoC

SoC security verification can be done either before or after the fabrication stage, called pre-silicon and post-silicon verification, respectively. Generally, in the pre-silicon verification, the verification target is typically a model of the design (a representation of the design at a specific design stage, like post-synthesis netlist, or the generated layout) than an actual silicon artifact. The pre-silicon verification activities consist of code and design reviews, simulation and testing, as well as formal analysis. These tests are running at different corner cases with constrained inputs. One of the biggest advantages of pre-silicon verification is its high observability as the design representation is available and any internal signal/wire/register can be observed and verified. However, since it is mostly simulation-basis at MHz speed limited by the simulator performance, it takes a lot of time for verification of all policies/properties.

On the other side, post-silicon verification can be considered as one of the most crucial yet expensive and complex forms of the SoC verification, in which a fabricated, but pre-production silicon of the targeted SoC (initial prototypes of the chips that are fabricated and are used as test objects) will be picked as the verification vehicle, and then a comprehensive set of tests will be executed on it. The goal of post-silicon verification is to ensure that the silicon design works properly under actual operating conditions while executing real software, and identify (and fix) errors that may have been missed during pre-silicon verification. In post-silicon verification, since the silicon is used as the verification vehicle, tests can run at target clock speed enabling the execution of long use cases at a much smaller time slot. However, it is considerably more complex to control or observe the execution of silicon than that of a pre-silicon verification, as the access and observability of many internal nodes will be lost.

3.5 Adversarial Model in SoC

To ensure that an asset is protected, the design team needs, in addition to the security policies/properties, to define and determine comprehension of the power of the adversary. The notion of the adversary can vary depending on the asset being considered. For instance, regarding the cryptographic and on-device keys as the security asset, the end-user would be an adversary, and the keys must be protected against the end-user. As another example, 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. So, based on the definition of security assets, source of vulnerabilities, and security policies/properties, the adversary model also needs to be clearly defined helping to have a stronger SoC security verification mechanism. Hence, 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.

3.6 Verification Model in SoC

The SoC security verification can be done at different stages each focusing on different sets of security policies/properties. Targeting the design at different stages will affect the model defined for the verification. In addition, the model can be also related to the adversarial model as the source of threats. For instance, for the untrusted foundry with having access to all masking layer information, to malicious end-user that has access to the fabricated chip mostly as a black-box, the security verification can be modeled differently. Hence, based on the access provided for the SoC verification framework, it can be generally divided into three main categories: white, gray, and black. The definition of these verification models in SoC is very close to their definition at the software level. In the white verification model, all internal wires, signals, nodes, registers, etc., are fully observable allowing the security properties/policies to be implemented in detail and very specifically based on the requirement. The black verification model treats the SoC as almost a black box, and only the general ports and potentially scan pins are available for testing. The gray model is any verification model that stands between white and black models, and the level of access to internal parts may be different per each case.

3.7 Scope of Security Verification in SoC

With the involvement of multiple abstraction layers in a complex and heterogeneous SoC, based on the security policies/properties associated with each abstraction layer, the SoC security verification can be categorized into three main domains:

(**SC1**) *Low-Level (Hardware-level)*: Once the security vulnerabilities arise from the underlying hardware, such as RTL and gate-level netlist, the scope of verification needs to be concentrated at logic-level (low-level), such as hardware Trojan insertion, counterfeiting, fault injection, etc.

(**SC2**) *Platform-Level (System-level)*: In this category, the vulnerabilities associated with the inter-IP communication and system-level bugs that exploited mostly by untrusted third-party components during run-time. As an example, any confidentiality and integrity violation for any inter-component communication that can lead to information leakage or unexpected actions can be considered as system-level vulnerabilities that require system-level security policies/properties definition.

(**SC3**) *Software-Level (Framework-level)*: This group refers to vulnerabilities arising from the intercommunication between hardware parts and the software or the framework. Additionally, network-based vulnerabilities, like communication of an embedded computing unit with off-chip modules or cloud can be categorized as a member of this group. In this group, the definition of security policies/properties would be more at the higher level of abstraction combined with checking flags/status at hardware levels.

4 SoC Security Verification: Challenges

Based on what we learned so far, for the automation of SoC security verification, we need to accomplish some steps: (i) identification of the source of vulnerabilities (**SV**s), that helps to define the threat models (adversarial model); (ii) indicating the security assets per component (**SA**s); (iii) definition of security policies/properties (**SP**s) based on the threat model and the chosen security assets; (iv) formalizing the security policies/properties using the unified language with consideration of the security domain (**SC**s); and (v) implementation and testing. To accomplish these steps, there exist challenges showing why new approaches like the self-refinement technique, i.e., fuzz, penetration, AI testing, are needed for the SoC security verification. Following section covers some of the biggest challenges that cannot be solved using the existing approaches, like formal satisfiability-based techniques, model checking, information flow tracking, etc.:

(i) *Preciseness*: For almost all aforementioned steps, the course of action(s) decided and accomplished by the designer(s) requires the highest precision to make the whole SoC security verification procedure a successful practice. Precisely evaluation of **SV**s, and understanding the threat models, precisely choosing **SA**s, precisely defining **SP**s, and formalizing using the selected language directly and significantly affects the outcome o the framework. For SoCs getting more complex and more heterogeneous, precisely indicating exacts **SV**s, **SA**s, and **SP**s becomes more and more challenging. The designer(s) needs to know all underlying information about all components, modules, frameworks, their intercorrelation, handshaking, their corresponding security levels, etc., which is almost impractical in modern SoCs.

(ii) *Multi-stage/layer verification*: Definition of **SV**s, **SA**s, and **SP**s are fully dependent on the stage of the design flow, and the abstraction layer of the design. There is no guarantee that when the security verification is passed in one stage of the design, then the same **SP**s will be passed in another abstraction layer. Changes and refinements per each stage, and moving from one abstraction layer to another one, might arise new vulnerabilities. A simple example for this case is the transitions done by synthesis tools, like high-level synthesis (HLS) and other computer-aided design (CAD) tools, which may add new data/control flow that can be exploited leading to a new vulnerabilities [21], [45]. This is why the invocation of SoC security verification is required at different stages and different layers of abstraction.

(iii) *Verification vs. Dynamicity*: Based on how the **SV**s, **SA**s, and **SP**s are defined, dynamicity might happen during runtime, meaning that there is potentially new **SA**s introduced when a specific set of operations are executed on the original **SA**s. This will propagate the original ones and based on the dependency/relation to other variables, some other variables may preserve critical information that must be considered as new **SA**s. Additionally, the threat models will be updated over time, resulting in new **SV**s, which lead to the introduction of new **SP**s. Also, protecting the design against one **SV** may make it vulnerable to the other one. For example, protecting a design against information leakage may create side-channel leakage that can be exploited by an attacker to retrieve sensitive information.

(iv) *Unknownness*: As we mentioned previously, the utilization of conventional techniques and tools, i.e., formal satisfiability-based techniques, model checking, information flow tracking, etc. for SoC security verification is mainly limited to the expert review, and they do not provide acceptable scalability. This is getting worse when the side-effect of dynamicity comes to the action, which is the introduction of unknown vulnerabilities. With unknown vulnerabilities, there is no precise definition for **SV**, **SA**, and **SP**, and they might be caught by accident/chance through either the tests by the designer(s) or by the attacks (like hammering) by the adversary. This is when the self-refinement techniques' contribution plays an important role, by using smart hammering and testing, to detect such vulnerabilities before releasing the SoC into the field/market.

5 SoC Security Verification: Assumptions

To have a successful SoC security verification solution, there exist some fundamental assumptions that must be considered meticulously as the most basic requirements of the proposed solution:

(i) *Time of Verification*: Since moving towards the final stages of SoC design, implementation, and testing makes the evaluation and investigation much harder, it is paramount to accomplish the verification, particularly for primary **SA**s and **SP**s at the earliest possible stage. Assuming that the vulnerabilities reach the post-silicon stage, the verification model will change from a low-level white model to a chip-level black model, which makes the verification much harder. Moreover, the cost of fixing the design is significantly higher as we advance through the later stages of the design. Furthermore, vulnerabilities that reach the manufacturing stage will cause revenue loss. According to the rule-of-ten for product design [46], modifying a design at the later stages of the SoC design flow is ten times costlier than doing in the previous steps.

(ii) *Evolutionary Mechanism*: Almost all existing security verification frameworks that are relying on conventional formal-based, satisfiability-based, or model-checking-based tools, tend to provide a binary response to the **SP**(s) associated with a set of **SA**s. So, the analysis behind the verification is very limited to a specific set of events, and the verification solutions just indicate that no flow occurs or that there is at least one that violates the defined **SP**(s). However, they do not provide any indication in between, e.g., predicting that we are approaching a vulnerability or not (majority part of **SP**(s) will be satisfied/passed). Providing evolutionary response (like providing feedback) can help the framework by itself to mutate the test-cases based on the collected data and in a smart way and narrow the search space for reaching to potential vulnerabilities. For building such structure, a definition of some security metrics or coverage metrics is needed as well to guide the test-cases to approach the vulnerability and eventually get the global minimal.

(iii) *Hammering-based Verification*: As mentioned in Section 4, due to the dynamic nature of threat model as well as the propagation of **SA**s that results in the introduction of newer **SA**s, unknown security vulnerabilities will appear in SoC transactions, leading to breaches of confidentiality, integrity, or authenticity, even while the verification has been done appropriately based on the known security policies. Hence, it is crucial for a security verification framework to be capable of finding both known and unknown vulnerabilities in the SoC, even though there exists no clear/precise definition for the actual vulnerabilities in the targeted SoC. This is when evolutionary-based verification comes into action and may help to detect such unknown vulnerabilities by a smart hammering, and helping to avoid such scenarios before moving to the next design stage.

(iv) *Hardware-software Co-Verification*: Some hardware vulnerabilities in SoC designs are not explicitly vulnerable unless triggered by the software [27]. In some cases, it might be possible to formalize such vulnerabilities via a complex sequence checking at the hardware level, but if the verification mechanism allows doing either hardware-software co-analysis or software-level verification, the definition of **SP**(s) can become more straightforward. So, the SoC security verification framework should handle such interactions as well to ensure the security of the SoC.

(iv) *Verification at Different Level of Access*: As the contribution of proprietary third-party IPs in modern SoCs is getting more and more, it decreases the visibility of the verification engineer from the internal specifications of the SoC. Therefore, the security verification framework should be able to verify the security of the SoC considering gray box or black box model.

6 SoC Security Verification: Flow

Considering the aforementioned challenges and assumptions, to get the benefit of self-refinement techniques for SoC security verification, the followings are the major steps that must be considered as fundamentals of the SoC security verification for both known and unknown vulnerabilities:
(**step1**) *Risk Assessment*: The main purpose of this step is to identify the **SA**(s). The **SA**(s) will be determined based on the ownership, domain, usage, propagation, static or dynamic nature, etc., and the outcome of this step would be a semantic definition as the requirements of verification activities.
(**step2**) *Definition of Adversarial Entry Region*: Per each defined **SA**, this step defines the most intuitive adversarial actions (**SV**s) around the **SA** that might lead to governing the assets, such as different entry point candidates, data/control relevancy between assets, and untrusted region(s), etc.
(**step3**) *Definition of Security Policies/Properties (Known)*: Based on **SA**(s) and **SV**(s), each vulnerability is converted to a set of rules and then those rules will be converted to a set of properties (**SP**s). For widely arisen vulnerabilities that can be categorized as known vulnerabilities, the formalism can be done by definition of assertions that monitor illegal events/sequences. The richability of the unified language plays an important role in this step (e.g., LTL [43] and CTL [44]) that pave the way for converting the **SP**s to hardware implementation.
(**step4**) *Definition of Security Policies/Properties (Unknown)*: Unlike known vulnerabilities, in case of unknown vulnerabilities (e.g., any possible scenario that leads to security asset leakage), the policy/property can be defined in the form of a cost function, which represents the evolutionary behavior of the vulnerability and tries to approach a certain state/location of the design in a different way to trigger unknown vulnerabilities. Cost functions can be considered as a relaxed or higher-level version of assertion-based **SP**s, and by using self-refinement tools, as discussed hereinafter, self-mutation can help to move towards testing data that excite the conditions leading to the specific class of vulnerability under investigation.
(**step5**) *Hardware Implementation of Security Verification Model*: For both known and unknown cases, by using the unified languages, all **SP**s must be converted to hardware implementation. For known cases, it is more likely assertion-based scenarios that check the sequence of events for a specific incident. For unknown cases, it is based on instrumentation and mutation, which helps to build the evolutionary mechanism.
(**step6**) *Security Verification and Testing*: This step involves running verification, testing, and refinement-based tools based on the definition of **SP**s and their hardware implementation that lead to verifying the security of the SoC. This step can be done at different stages of the IC design, from high level to GDSII as pre-silicon, and after fabrication on initial prototypes as post-silicon.

In the following Sections, we will discuss how self-refinement techniques and tools, like fuzz, penetration, and AI testing can be engaged along with the consideration of cost function definition for finding both known and unknown vulnerabilities of the SoC.

7 SoC Security verification: Fuzzing

The words *fuzz testing* or *fuzzing* represent randomized testing of programs to find out the anomalies and vulnerabilities. Fuzzing is usually an automatic or semi-automatic approach that is intended to cover numerous pre-defined (instrumented) corner cases of invalid inputs to trigger the existing vulnerabilities in a program. Fuzzing was first applied by Miller et al. [47] to generate random inputs, which were provided to `Unix` to find the specific inputs that cause crashes.

Generating random inputs without any feedback from the design under investigation is referred to as blind fuzz (close to random) testing which suffers from low coverage, especially when more complex input models are introduced. This has led to additional stages being introduced to the fuzzing platforms. Generally, three main steps are involved in a fuzzing that will be invoked iteratively on the program:
(i) *Test scheduling*, which relates to the problem of ordering the initial seeds of each fuzz iteration in a way that leads to full coverage as fast as possible.
(ii) *Mutation steps*, which incorporate a variety of methods ranging from a simple crossing of seeds to optimized genetic algorithms in order to produce new seeds for further exploit generation.
(iii) *Selection*, where the useful seeds are pruned out of all generated seeds. this process requires metric evaluation as the prominent way of deciding which seeds result in better coverage of the whole design under test [48].

Amongst the existing and widely used fuzzer tools, american fuzzy lop (AFL) [15] is one of the most popular software fuzzers that uses an approximation of branch coverage as its metric, while another famous fuzzer named Hongfuzz [49] accounts for the unique basic blocks of code visited. Based on the benchmarks used by different fuzzers, the baseline of fuzzing, crash type, coverage mode, and seed formulation, past fuzzing mechanisms divided into numerous groups, whose details and comparison can be found in [50]. The following first shows how the verification model can change the way fuzzer acts on the targeted program, and then we will investigate how the fuzzer can be engaged for SoC security verification. As discussed previously, the purpose of using such self-refinement approaches is to overcome the scalability issues of formal verification methods [51].

7.1 Formal Definition of Fuzz Testing

Based on the verification model, fuzzing-based techniques can be classified into three categories: white box, gray box, and black box, which are defined based on the availability of information during verification or run-time phases. This information may include the source code of the hardware or software designs, detail information about the security specification and functionalities, code coverage, control and data flow graphs, execution (simulation or emulation) related information such as CPU utilization, memory usage, etc. The following describes these three categories of fuzzing.

7.1.1 Black-box Fuzzing

Black-box testing does not take any information from the program under test. This approach also does not obtain the input format of the program, rather it generates random inputs which are mutated from a given seed data provided as arguments or a file. This approach can use some predefined rules, such as bit flipping, varying input length (bits), sign reverse, etc. to generate mutated inputs. Some recent black-box fuzzing-based approaches use grammar or input specific information to obtain partially valid inputs [52]. Black-box fuzzing is very convenient to use when the design specification is unknown and little information is available about the internal parts of the design under test. On the contrary, it is very challenging to generate test cases for a program with a very large number of execution paths due to the lack of diversity of mutated inputs, which may usher the failure of reaching the corner cases. Again, due to the inherent blindness, black-box fuzzing-based approaches struggle with the code coverage in practical use cases of finding vulnerabilities.

7.1.2 White-box Fuzzing

Unlike black-box fuzzing, in white-box fuzzing, all of the information of the target design is transparent and available for the verification engineers to use. This approach was first introduced by Godefroid et al. [53]. White-box fuzzing utilizes as much information as needed to guide the seed generation effectively. This approach analyzes symbolic constraints for all conditional statements in the program in order to develop the path constraints for all possible executions. Integrating a coverage-maximizing metric to white-box fuzzing can accelerate the approach to find the inputs triggering the vulnerabilities in the program [53]. Although theoretically it seems that white-box fuzzing can generate inputs to cover all of the possible execution paths, practically it's very challenging to achieve due to too many execution paths and time restrictions of running a program.

7.1.3 Gray-box Fuzzing

Gray-box fuzzing is a hybrid approach that mixes black-box and white-box fuzzing. Gray-box fuzzing obtains partial information of the design under verification. For example, while in white-box fuzzing we could instrument the whole design for attaining code coverage, here we might be only able to instrument the final binary and not the code itself, or in case of a SoC design, we might have access to the bus interface without being able to probe anything else. A directed information feedback can largely assist in guiding the mutation engine so that it can cover more control paths and hence find the vulnerabilities in a short time. The common method of mutation strategies for this type of fuzzing is genetic algorithm [54], taint analysis [55], etc. Taint analysis assists to focus on mutating the inputs which have a larger impact on the targeted vulnerabilities. As this fuzzing approach possesses some information about the design, it can implement a targeted feedback system to trigger the vulnerabilities and explore new paths in the program, which significantly improves the detection capability and coverage of the fuzzing approach.

7.2 Fuzzing Hardware like Software

The complexity of hardware/software security co-verification is caused by complications that arise from interfacing numerous sub-modules from both sides of the table. The approaches presented to this day have failed to propose a scalable method with sufficient coverage that captures the hardware and software vulnerabilities in a unified platform simultaneously. Fuzzing has proved to be a powerful tool for detecting vulnerabilities in real-world software programs that encompass huge code bases. This has opened the door for many researchers to investigate the possibility of applying the same mechanisms to a full system that includes hardware, firmware as well as software components.

7.2.1 Hardware to Software Abstraction

In order to have a coherent model of the system so the previous fuzzing efforts can be re-used, one method is to translate the hardware to the software world. Utilizing state-of-art fuzzing tools for hardware verification is a promising concept which does not enforce the development of a new platform for RTL verification. In order to do that, RTL designs are translated to equivalent software models by a hardware translator, such as Verilator [56], which generates C++ programs of the given RTL designs. The generated C++ classes are instantiated by a wrapper which acts as the main function and stimulates the program under test for simulation. The wrapper, which is a test-bench in hardware terms, is designed in a way that holds functionalities and security properties that act as a cost function which provides feedback information for future iterations. The generated C++ programs can also be instrumented to increase the visibility of the internal simulations. Instrumentation can also help to achieve code coverage, line coverage, branch coverage, etc. to evaluate the overall coverage during simulation.

A metric which we call the cost function is introduced to measure the extent to which the security of a SoC design has been compromised and how close are the test scenarios to activating a targeted vulnerability. In that sense, the Cost function is helping to build the evolutionary mechanism for a vulnerability that guides fuzzing towards detecting the vulnerability in a significantly shorter time period compared to blind fuzzing. The Cost function may also include general metrics such as code coverage, line coverage, branch coverage, etc., which can be obtained through instrumentation of RTL designs or translated software models. The cost function works in tandem with the mutation engine of the fuzzer to assist in generating better mutated test cases that can trigger various corner cases.

Figure 5 shows how the transformation of RTL code to an executable binary that encompasses instrumentation takes place. Also this figure represents how the cost function interacts with the fuzzer outputs as well as the mutation engine. The special characteristic that is worth mentioning here is the fact that the cost function is selected based on

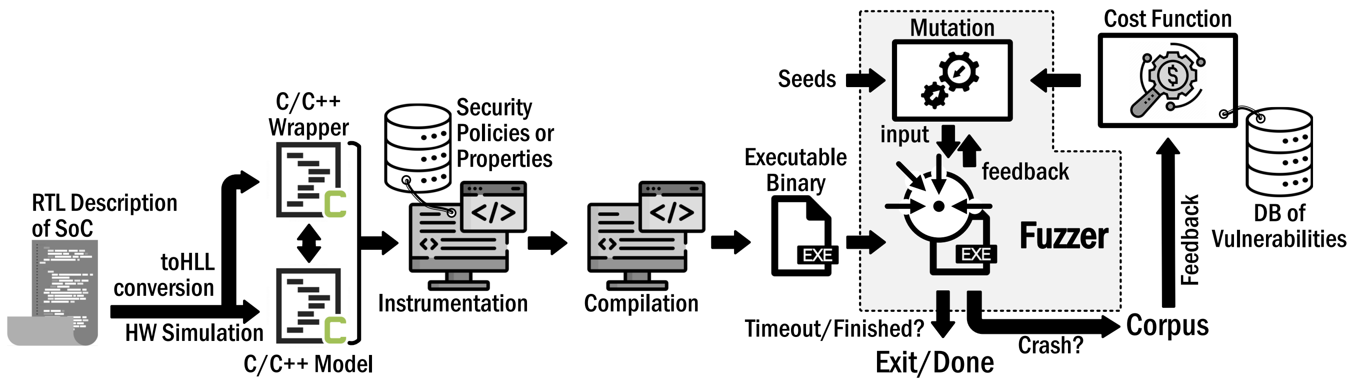

Fig. 5: A Fuzzing Framework on Software Level of RTL.

the vulnerability instance that is currently being targeted. This means that instead of a general feedback from the fuzzer outputs, one can make more suitable decisions for further analysis because now the feedback is tailored for the targeted bug in the running verification session.

7.2.2 Prior Art HW-to-SW Fuzzing

This section reviews the studies that incorporate hardware-to-software abstraction conversion for realizing software fuzzing for hardware verification and the prominent deficiencies of each method are mentioned. The authors in [57] proposed a mutational coverage-guided fuzzing-based framework in order to resolve modern SoC verification challenges, such as co-verification of hardware and firmware, as well as scalability. The framework includes adversarial behavior and coverage metrics to evaluate the security properties written in their proposed logic HyperPLTL (Hyper Past-time Linear Temporal Logic). However, the primary disadvantage of this technique is that it requires a huge amount of technical expertise, particularly for building **SP**s, and hence incurs higher chances of erroneous and low coverage results which could eventually limit the scope of bugs detection. Tripple et al. [51] developed software models of hardware and then applied fuzzing on the software leveraging Google's OSS-fuzz. The main problem with this approach is due to its dependence on general metrics such as code coverage rather than a dedicated cost function targeted at hardware model properties.

Moghimi et al. [58] utilized fuzzing by mutating the seeds of existing Meltdown variants in order to discover various Meltdown-type attack variants. The authors provided randomly mutated inputs to the associated faulty loads and used the cache as proof of covert channel for side-channel leakage. Oleksenko et al. in [59] developed a dynamic testing methodology, SpecFuzz for identifying speculation execution vulnerabilities (e.g., Spectre). SpecFuzz instruments the program and runs the simulation for speculative execution in software-level traversing all possible reachable code paths that may be triggered due to branch mispredictions. During simulated execution of the program, speculative memory accesses are visible to integrity checkers which is combined with traditional fuzzing techniques. SpecFuzz can detect potential spectre like vulnerabilities but it is only useful for this certain type of attacks that exploit speculation in CPU pipeline.

DiffFuzz proposed in [60] is a fuzzing-based approach in order to detect side-channel vulnerabilities related to time and space. This technique analyzes two copies of the same program which have different secret data while providing the same inputs to them both. The authors developed a cost metric which estimates the resource consumption, such as the number of executed instructions and memory footprint, between secret-dependent paths for both programs. However, this technique is primarily dependent on the details of microarchitectural implementation information. The more visibility into the microarchitectural state, the more coverage is achieved. Unfortunately, obtaining good visibility is not always available for many hardware designs which binds these techniques to low accuracy. Yuan Xiao et al. in [61] developed a software framework leveraging fuzzing concepts to identify Meltdown-type vulnerabilities in the existing processors. The authors build up the code using some templates, which are executed to find out the vulnerabilities. The authors leveraged cache-based covert channel and differential tests to gain visibility into the microarchitectural state changes, which eventually helps to analyze the attack scenarios.

Ghaniyoun et al. [62] proposed a pre-silicon framework for detecting transient execution vulnerabilities called IntroSpectre. IntroSpectre is developed on top of Verilator. The authors resolved the challenges of lacking visibility into the microarchitectural state by integrating it into the RTL design flow which makes it identify unreachable potential side-channel leakages. The authors utilize the fuzzing approach to generate different attack scenarios consisting of code gadgets and analyze the logs obtained from simulation to identify the potential transient execution vulnerabilities.

7.2.3 Limitations and Challenges

There are many rudimentary differences between fuzzing a software program and a RTL hardware model, the first one of which is due to the difference in input arguments. A digital circuit has the notion of input ports that take different values in each cycle, unlike software that reads its inputs from a variety of sources including arguments, files, or through OS system-calls. Fuzzing requires a solid definition of the format of the input so it can do meaningful mutations and generate new passable tests. So the actual input to be fuzzed should be thoroughly explained to the fuzzer when working with a translated hardware design.

The format of the input has a great impact on how well the mutation engine performs.

The second issue to address is due to the difference in software versus hardware coverage metrics. Many fuzzers depend on instrumentation to obtain coverage metrics and direct the seed generation towards uncovered sections of the design. While some metrics such as branch and line coverage can approximately be mapped to each other in both hardware and software models, other metrics such as FSM coverage or toggle rate are not translatable [51]. The traditional hardware verification uses these metrics to target specific classes of vulnerabilities and any effort in software domain should comply with these previous platforms as well.

The third dominant issue with fuzzing on the translated hardware is with regard to the cost function. Software fuzzers look for crashes, exceptions and memory checks as the vulnerabilities that could exist in a software model. These scenarios are not convertible to hardware designs, particularly low-level or platform-level vulnerabilities, because these forms of errors like exceptions do not exist in the hardware realm. Hardware is inherently different when it comes to targeting vulnerabilities because software crashes can still happen even when the hardware is fully functional and secure. Fuzzing the translated model without introducing our own notion of hardware vulnerabilities through a cost function will result in the fuzzer expending its resources on detecting bugs induced by the translator rather than the hardware model itself [63]. Another problem related to this issue is due to the changes in variables and functions when performing the translation. This makes the translation of properties from hardware RTL model to software a crucial task that has not been investigated.

7.3 Direct Fuzzing on Hardware RTL

The verification faces additional challenges with enlarging and growth of the design's size and scope, e.g., the verification for a full-scale complex and heterogeneous SoC. The primary challenge relevant to the SoC security verification is the lack of end-to-end verification methods which can resemble the behavior of every hardware component and the run-time implications of software components or framework(s) executing on it, i.e., hardware/software co-verification with maximum coverage. Again, scalability is the biggest challenge in modern SoC verification due to the complexity and massive size of the designs. In order to tackle scalability, an automated and systematic verification platform is required. However, automation in SoC verification is very difficult for several reasons. Firstly, the verification engineer faces extreme challenges to precisely assess the security policies/properties of an SoC design due to the enormous number of components that interact with each other. Identifying the security policies/properties in a comprehensive fashion largely influences the quality of SoC verification. Secondly, the verification engineer faces challenges in modeling the attack scenarios that may happen in the SoC. These attacks may include side-channel based, direct hardware or software exploitable attacks that could run on the SoC. It's very challenging to prepare different attack models from the specifications for different untrusted third-party IPs [64], untrusted OEM firmware [65] and untrusted software [20]. Again, the verification engineer may not get a complete manual for third-party IPs. Hence, SoC verification still is a burdensome task that demands significant research to develop a robust and scalable solution.

In such an environment the emulation-based fuzzing methods, with more concentration on gray-box fuzzing mode, can greatly improve the performance of the simulation-based approaches by running the design on an FPGA and interfacing the fuzzer to the prototype design under test in a more realistic manner. Figure 6 represents a general platform for interfacing FPGA with a test generator fuzzer. The difference of this method with what was discussed in Figure 5 is mostly in the bitstream generation as opposed to a binary under test and the fact that the test cases are introduced to the actual design through a direct memory access channel. The instrumentation in this approach is outputted through the probe analyzers available at FPGA monitoring implementation and it helps the cost function to better guide the mutation engine.

Direct fuzzing on FPGA-accelerated simulation incurs some major challenges such as resetting the FPGA, which is required to take the program to a particular known state to decrease the verification time or defining the branch coverage, which is needed to track the verification coverage. In order to solve the first challenge, memory snap-shooting techniques can be used to reset the program and set the desired state as a preparation for each test intended to be performed with new fuzzing inputs. In order to estimate the branch coverage, branches are mapped to multiplexers which output one of two input values in each cycle.

Kevin Laeufer et al. in [66] proposed a coverage-directed fuzzing approach for RTL testing on FPGAs leveraging ideas from the software testing community (RFuzz). The authors proposed a new approach for coverage metrics, which uses multiplexer as branch coverage. RFuzz employs FPGA accelerated fuzzing in order to speed up the fuzzing procedure, which may increase specialized hardware costs and also is limited to language support. However, it is still lack of the definition of the cost function for security verification of the design under investigation. Also, this mechanism supports a very limited set of design types, as they cannot accomplish the testing while the processor is in place, and the verification is required to be intercorrelated between hardware, software, and firmware.

In general, fuzzing can be a promising solution for developing an end-to-end mechanism for full system verification and it requires a minimum amount of adjustment since it is already well-developed in other areas of research, standalone simulation-based fuzzing or FPGA-based interfacing still cannot cover a wide variety of threat models and source of vulnerabilities showing further investigation is inevitable in this domain.

8 SoC Security Verification: Penetration

Penetration testing (PT) is a methodology to assess the vulnerabilities in an application or a network and exploit those weaknesses to gain access to the resources. Since accessing security-critical resources of a computing system necessitates the exploration of exploitable vulnerabilities, vulnerability assessment (VA) is an indispensable precursor

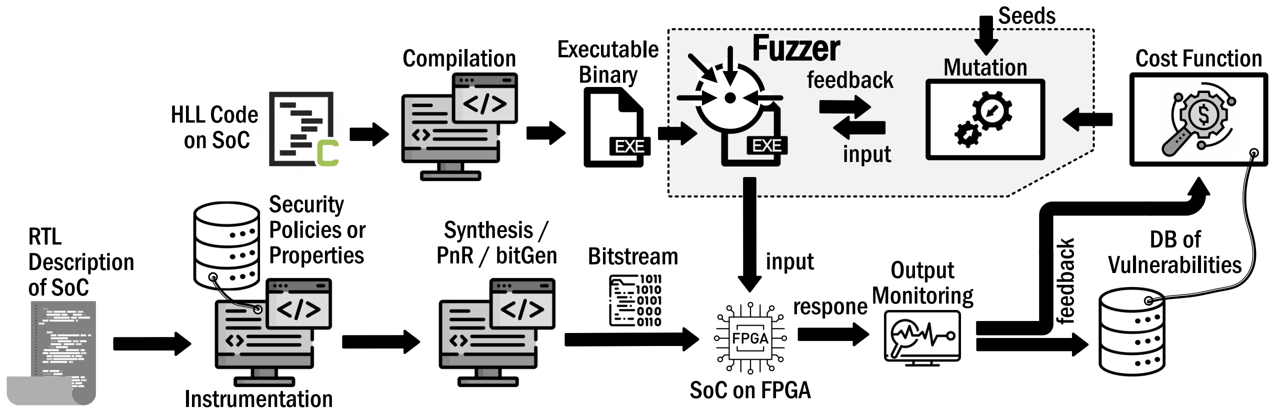

Fig. 6: A Framework for Direct Fuzzing of RTL.

to PT. In literature, therefore, VA and PT are often considered a single framework to assess the security of a network or application [67].

8.1 Higher Abstraction Penetration Testing: Types, Practices and Objectives

Motivated by the ever-growing attack surface of software, the secure software development lifecycle has been adopted by leading software companies over the last two or three decades. Security issues are considered, analyzed, and attempted to be discovered at even early stages of development [68]. Consequently, in the software development lifecycle, VAPT has become a well-defined methodology to weed out bugs and vulnerabilities. In fact, there are governmental and private organizations that accredit individuals and groups based on their ability [67], [69]. Open Web Application Security Project (OWASP) is a non-profit organization, for example, that has published and sanctioned a detailed step-by-step process of performing penetration testing of web applications and firmware [70]. Open Source Security Testing Methodology Manual (OSSTMM) also offers detailed guidelines on how to conduct VAPT on different systems [71]. CREST, Tigerscheme, and the Cyberscheme are some other professional bodies that provide industrial certifications and qualifications. CREST in particular provides the intelligence-driven red team penetration testing framework to assess the robustness of an operational team against cyberthreats [72]. Mitre sponsors the maintenance of two detailed databases of known software vulnerabilities- the CVE (Common Vulnerabilities and Exposure List) and the CWE (Common Weakness enumeration). These two databases contain extensive information on most common software vulnerabilities, example implementations, and even discuss potential mitigation solutions. Therefore, VAPT is a well-delineated methodology in software and network security that has been implemented in practice successfully over the course of the last two or three decades.

The main objective of a penetration test pertaining to an application or network is to find out exploitable vulnerabilities. Colloquially, VAPT has also been described as ethical hacking [73]. This terminology is inspired by the fact that often, the best approach to finding unanticipated vulnerabilities in a computer system is to *hack* into the system from the outside looking in. This is essentially a *simulated attack*, in which the penetration test engineer tries to anticipate the course of action that malicious actors might adopt while trying to compromise the security of the system. An analogy is often used in penetration testing colloquialism, where it is compared to hiring a thief to break into one's own house to find the loopholes in the implemented security system. The term penetration in this context, therefore, stands for acquiring legitimate access to resources illegitimately. We present the typical steps in a typical VAPT approach in the following:

(i) *Reconnaissance*: In this step, the tester gathers extensive knowledge on the application, network, or computing system to be pen tested. The principal goal of this step is to get familiarized with the particular technology, protocols, software versions, IP addresses, the configuration being used by the system. Websites, job postings, and even social engineering can be used to gather information [74]. *Httrack*, *Harvester*, and *Whois* are some tools that can help in this step.

(ii) *Scanning*: In the scanning step, the system or network is at first probed to find points of entry. An example is *Nmap* program to find open ports in a network. Subsequently, a commercially available tool like *Nessus* is used to find known vulnerabilities in the system.

(iii) *Exploitation*: From the list of known vulnerabilities, the tester comes up with an *attack* or exploitation plan. The goal of this step is to identify the sequence of actions that can be taken to gain access to unprivileged resources in the system. *Metasploit* is an open-source tool that is frequently used in academic and professional settings to realize this step. Effective exploit management (search, upgrade, documentation) or a large number of payloads (tasks that are done after successful exploitation of the target system) are available in Metasploit. In general, payloads can be either simple and focused on a single activity (for example, user creation) or complex and comprehensive and provide more advanced functionality.

(iv) *Post Exploitation*: The post exploitation step documents the steps taken to gain access to non-permitted resources (if successful). It might also involve the tester attempting to escalate privileges already gained in the system. The documentation of the steps taken gives valuable insight into the weakness of the system.

8.2 Formal Definition of Penetration Testing

Similar to other testing approaches, based on the depth and level of access as well as verification model, there are three types of penetration testing:

8.2.1 Black-box Pen Testing

The testers do not have any prior access to any resources on the test target when performing black box penetration testing. They are expected to figure out all of the minutiae of the system, as well as any flaws, depending on their previous experience and individual expertise. The tester's primary goal is to audit the external security boundary of the test target; as a result, the tester replicates the activities and procedures of an actual attacker who may be located at a location other than the test target's boundary and who does not know anything about the target. OSSTMM makes a distinction within what would be typically referred to as black-box PT between blind and double blind testing. In double blind testing, the target is not notified ahead of time of the audit whereas in blind testing it is informed ahead of time.

8.2.2 White-box Pen Testing

Contrary to black box PT, the testers are provided with all of the internal information about the system. This is meant to simulate an attack from an internal threat like a malicious employee. White box PT offers higher granularity of testing while at the same time offering the benefit of not relying heavily on trial and error as is common in black box PT.

8.2.3 Gray-box Pen Testing

Gray box PT is somewhere in between black box and white box methodologies in terms of the information available to the testers.

8.3 Penetration Testing on Hardware: Definition

Compared to the software domain, hardware penetration testing is in its infancy. The term has been used as a stand in for merely vulnerability assessment or for post-silicon testing and debugging in hardware security literature. Authors in [75], for example, equate penetration testing of hardware to post-silicon debugging and testing. The examples they provide amount to testing the software layer being run on the hardware and not the hardware itself.

In keeping with the stated goals of software PT, we believe that the principal objective of hardware PT should be to discover vulnerabilities in the hardware that can be exploited. However, certain differences must be noted. Firstly, vulnerabilities in hardware can be purely hardware-oriented such as malicious modification of the hardware description to compromise the integrity and confidentiality of the device [4], side-channel leakage [76], and fault injection [77]. Alternatively, they can be the source of cross-layer vulnerabilities which can be exploited through the software layer of the computing stack. Spectre and Meltdown vulnerabilities, which leverage the weakness in hardware implementation of speculative execution are examples of these types of vulnerabilities [34], [35]. Secondly, the after-deployment *simulated attack* scenario to probe for vulnerabilities provides limited benefits when translated to hardware. In contrast to software that can be updated with a patch once a vulnerability has been discovered, hardware can not be easily patched (especially ASICs). A pre-silicon testing methodology would be much more beneficial to the designers and verification engineers.

In light of the challenges and foregoing differences with the software domain, we define pre-silicon hardware penetration testing as a testing methodology that propagates the effects of vulnerability to an observable point in the design in spite of cross-modular and cross-layer effects present in the design. In contrast to randomized testing which develops test patterns without the knowledge of the vulnerability it is seeking to detect, hardware penetration testing assumes a gray or black box knowledge of the specification of the design and a gray box knowledge of the bug or vulnerability it is targeting. The gray box knowledge of the bug implies that the tester has knowledge of the type of bug or vulnerability it is, and how it might impact the system but not the precise location of its origin or the precise point in the design where it might manifest in a complex SoC. Penetration in this context, therefore, refers to the propagation of a vulnerability from an unobservable point in the design to an observable point.

8.4 Penetration Testing on Hardware: Framework

In this section, we demonstrate how a binary particle swarm optimization (BPSO) based penetration testing framework can be used as a promising solution for the SoC security verification domain.

8.4.1 Binary Particle Swarm Optimization (BPSO)

The particle swarm optimization (PSO) is an evolutionary computation technique motivated by the behavior of organisms. PSO has been widely employed in a range of optimization situations due to its simplicity and ease of implementation. The PSO method is initialized by randomly placing a population of individuals, called particles, in the search space and then searching for the optimal solution by updating individual generations. At each iteration, for the jth index in the ith particle of the swarm, the position and velocity of the particle are updated through the following equations:

$$\begin{aligned} v_{i,j}(t+1) = wv_{i,j}(t) + c_1R_1(p_{besti,j} - x_{i,j}(t)) \\ + c_2R_2(g_{besti,j} - x_{i,j}(t)) \end{aligned} \quad (1)$$

$$x_{i,j}(t+1) = x_{i,j}(t) + v_{i,j}(t+1) \quad (2)$$

R_1 and R_2 are uniformly distributed random numbers between 0 and 1, c_1 and c_2 are acceleration coefficients and w is called positive inertia constant.

Kennedy and Eberhart modified the continuous variable PSO algorithm for binary spaces in [78], which came to be known as the Binary PSO (BPSO) algorithm. The adaptation for binary spaces is done through a set of constraints. The constraints are imposed on equation 2 in the following form:

$$x_{i,j}(t+1) = \begin{cases} 0, & \text{if } rand() \geq S(v_{i,j}(t+1)) \\ 1, & \text{otherwise} \end{cases} \quad (3)$$

where $S(.)$ is the sigmoid function used for transforming the velocity.

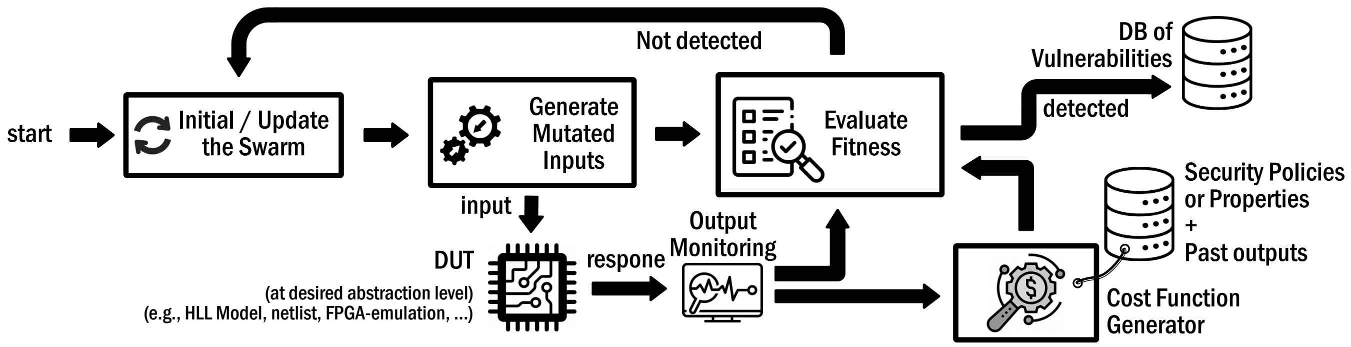

Fig. 7: A BPSO based Hardware Penetration Testing Framework for Detecting RTL Vulnerabilities.

8.4.2 BPSO-based Hardware Penetration Testing

Based on our earlier discussions on hardware penetration testing, the framework shown in Figure7 can be a BPSO-based pen testing approach on hardware that can be applicable for SoC security verification. At the core of the vulnerability detection process is the cost function generator. This generator (ideally automatically) generates a mathematical function that describes the vulnerability that the tester is attempting to detect. The input to the design is described in terms of a binary vector which is mutated upon based on the evaluation of cost function for a generation of input test vectors. For each generation of the swarm, the algorithm tries to minimize (or maximize) the cost function (that helps to build the evolutionary mechanism). The observed output can be any observable point in the SoC including hardware signals during functional simulation, memory address contents available from simulation or emulation, observable points or signals created by RTL instrumentation or scan chain insertion, and even the output of a user space program. The cost function generator generates the mathematical description of the vulnerability based on the observable output point, a description of security policy, and an append-only database of observed past outputs for a particular sequence of inputs applied or actions performed. The cost function generator must rely on keeping a record of outputs attained for *a sequence* of inputs since only a sequence of inputs can trigger certain hard-to-detect vulnerabilities. For example, the AES-T1100 Trojan described in the Trust-Hub database [79] gets activated upon the application of a predefined sequence of plaintext.

In order to apply such BPSO-based pen testing framework, there are three prerequisites that must be met:
(i) The tester should possess a preliminary knowledge of vulnerability in addition to the impact it might have on observable output and how it can lead to the violation of predefined security policies of the device. We argue that this is a reasonable supposition since a significant portion of RTL hardware vulnerabilities have well studied effects. For example, hardware Trojans can cause integrity, confidentiality, and availability violations in a circuit [6], [80]. Security unaware design practices can lead to a design having unanticipated leakage of information and assets to an observable point or to an unauthorized 3PIP in the design [81], [82], [83], [84]. Furthermore, there are open source databases (e.g. Common Weakness Enumeration) that catalogue commonly found vulnerabilities in hardware along with the impact they might have. The framework's primary application would be to test vulnerabilities for which the tester has a high level working knowledge of how they can result in a security policy violation.
(ii) The tester should have access or visibility to certain points in the design anticipated to be affected by the triggering of the vulnerability. The encryption key used in the crypto core of an SoC is an asset. To test whether a vulnerability exists in the design which can lead implicit or explicit flow of this asset to a PO, the observable point in the design would be the PO. On the other hand, if we are testing to check if the vulnerability enables flow of the asset to an unauthorized 3PIP, the observable point should be the SoC bus through which this type of transaction might take place.
(iii) The tester should have reasonable (but not necessarily exact) knowledge of how to trigger the vulnerability. This in turn would dictate the input to mutate on. For example, let us consider the debug unit vulnerability described in [85] where the password check of the JTAG interface could potentially be bypassed by resetting the unit. In this case, the hardware debug signals exposed to the outside world would be the relevant inputs. For the key asset scenario described earlier, the input to mutate would be physical signals of the crypto core exposed to the outside world or a user space program that can access AES resources. Similarly, for triggering software exploitable vulnerabilities the input to mutate would be the data and control flow of a user space program.

This stands in contrast to random and blind fuzz testing, which find vulnerabilities by applying random inputs to the design which in turn can unexpectedly lead the design to a non-functional or vulnerable state. The BPSO algorithm is suited for vulnerability detection at the pre-silicon level since any input to a digital design can be considered in terms of binary vectors. Discernibly hardware signals are binary quantities. The user space programs run on modern SoCs can also be described in terms of binary vectors by translating the associated program into corresponding assembly instructions. Additionally, to incorporate sequential inputs, each particle in the swarm can be considered as input vectors applied at different clock cycles.

8.4.3 Validity of Gray-Box Assumptions

Since the BPSO-based penetration test framework assumes knowledge on the part of the tester and the availability of

RTL code, the readers might assume that this violates the gray-box testing goals of the framework. We now discuss why the prerequisite knowledge assumed earlier does not necessarily violate gray-box assumptions especially in the context of SoC verification. Modern SoCs can contain tens of different third-party IPs, many of which are too complex with their implementation details abstracted away by integrating CAD tools. Even during functional verification, the verification engineer would only have a high level knowledge of the vulnerability and functionality of the integrated 3PIP and not the minutiae of RTL implementation. In such cases, it can become extremely challenging for the verification engineer to trigger the vulnerability with existing verification tools due to complex transactions occurring inside the SoC, implicit timing flows, or unanticipated security unaware design flaws. We also note that formal verification tools such as Cadence JasperGold®, even with definitive knowledge of the impact a vulnerability might have, can throw false positives or suffer from state explosion problems [86].

9 SoC Security Verification: AI Testing

Pre-silicon verification is an essential but time-consuming, and tedious part that consumes about 70% of the total time allocated for hardware design flow [87]. Similar to fuzz and pen testing, machine learning (ML) can be used for the SoC security verification to make the process automatic and evolutionary-based. ML can be used in different aspects of the verification process, such as generating new test cases that cover more functional states, producing new stimuli and input patterns for increasing coverage, analyzing the test results, etc. The biggest challenge of using ML will be selecting appropriate data, models, and ML techniques through trial and error to get the desired automation and coverage that the state-of-the-art techniques are unable to provide.

9.1 Higher Level Machine Learning

In general, three commonly used words - machine learning (ML), deep learning (DL), and neural network (NN) - are sub-fields of AI. While intelligent devices use AI to replicate human thought processes, ML is the mechanism to develop its intelligence. ML uses statistical models and extracts patterns from data to train a system without direct instructions. ML facilitates an intelligent approach to continue learning and improving through feedback. Even though it is common to use machine learning and deep learning interchangeably, they are not the same. DL is a sub-field of ML, whereas NN is a sub-field of DL. Fig. 8 shows the simplified version of the machine learning workflow. While applying machine learning, the first step is to formulate the problem statement. After that, collecting appropriate data is an essential stage to train the model to solve the problem like a human. The overall performance of the ML model depends on how representative the training data is. ML model learns from some extracted features of the relevant data. The higher the amount of data, the better the training would be. Hence, ML data must be the adequate amount (the more the data, the better), must have appropriate depth and feature (if inherently 2d data is represented by 3d data, the training method will not be effective). Data also have to be representative unbiased and should include every corner case possible. Once the developers have enough data, they can move forward to train different ML models to see which model works best for their problem statement and collected data.

Fig. 9 shows different tasks a machine learning model can perform (classification, anomaly detection, etc). Depending on the task, developers have to choose a method of training, for example, supervised or unsupervised learning. Once the developers have selected the task, training method and the model itself; they can then fine-tune the best model for their application and deploy it for usage.

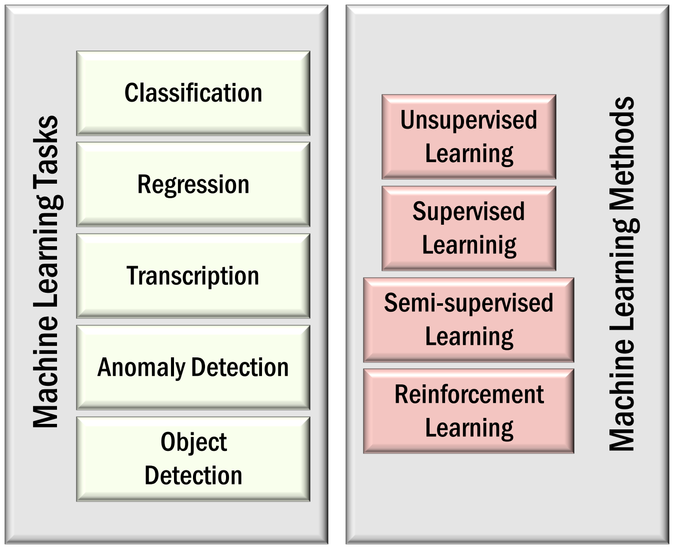

Fig. 9: Machine Learning Task and Methods.

Using ML to alleviate some manual interventions by verification engineers has become a recent research trend. In these research works, authors have used ML to automate different aspects of hardware verification. For example, Hughes et al. [88] used a combination of supervised and reinforcement learning to generate constrain-random stim-

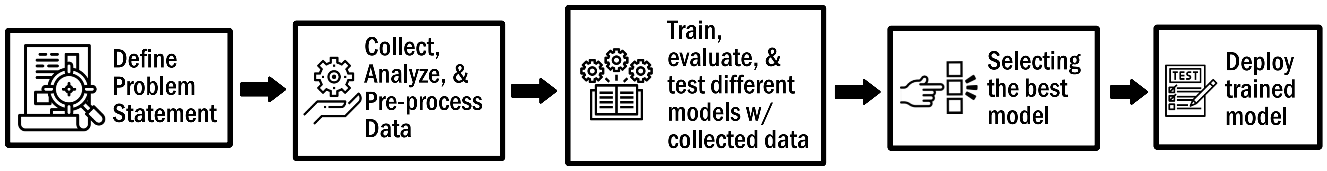
Define Problem Statement
Collect, Analyze, & Pre-process Data
Train, evaluate, & test different models w/ collected data
Selecting the best model
TEST
Deploy trained model

Fig. 8: Machine Learning Workflow.

ulus, which will ensure to hit the hard-to-hit combination in highly complex functional design space. Hutter et al. [89] used AI to automatically tune the decision-making procedure of bounded model checking SAT solvers, which in result would boost the verification procedure. Sometimes verification engineers need to recreate a failure to trace back the input that causes system failure. However, stimulating the origin of a system failure is time-consuming and computationally draining. Gaur et al. [90] proposed an ML model for this debugging purpose. The ML model will be trained to calculate the switching probability of the design output, which can be used to simulate system failures with negligible overhead.

9.2 AI for Hardware Verification

When trying to use ML for hardware verification, the ML workflow shown in figure 8 has to be followed. Figure 10 illustrates the workflow on how ML can be used to enhance hardware verification process. In the following subsections, different requirements and possible challenges of using this workflow for hardware verification are discussed in detail.

9.2.1 Requirements/Workflow for using ML in Verification

Hardware verification is done in every abstraction level of the hardware design flow. For example, after materializing a design's concept and architectural specification, behavioral verification is done on the RTL level. Next, functional verification is done at a gate level, transistor level, or during DFT insertion. Finally, the synthesized layout goes through a physical verification process. Therefore, while implementing ML for automating verification, the first requirement is to select the level of abstraction, meaning which method (behavior, function, or physical verification) to use and which level (RTL, gate, transistor, or layout) to use. This is the first step of using ML in pre-silicon verification as shown in figure 10.

Next, verification engineers must choose what part of the verification to automate through ML. For example, ML can generate stimuli, generate new test cases to increase code or branch coverage, produce new guided, constrained-random, or entirely random inputs to hit more functional or behavior nodes. Therefore, it is required to select the abstraction level and the role of ML in automation as a problem statement as the first part of ML workflow. As shown in figure 10, verification engineers have to divide the verification steps into two parts. The first part can be automated through ML, and the second part is the state-of-the-art verification steps that do not need further optimization. Verification engineers have to build problem statements and cost functions corresponding to the first part and leave the second part.

One factor that goes into consideration while establishing a problem statement is if the verification engineer has full access to the design (white-box setting) or has only access to primary input and primary output of the design (black-box setting). Traditionally design engineer and verification engineer are two different entities, and verification engineer does not require any knowledge of the design itself. Also, if the IP came from a 3PIP vendor, white box knowledge of the IP is inaccessible, as often 3PIPs are encrypted. However, the level of access to the design information (white-box/ gray-box / black-box setting) is crucial for collecting training data for the machine learning model.As shown in figure 10, to get the appropriate training data, the verification engineer may need to instrument the IP. This training data have to fulfill the requirement of being comprehensively representative of the complete problem statement, unbiased, and should cover all corner cases. After collecting required training data and analyzing the data structure, it is mandatory to extract features that represent the problem statement the best. These data features will be used to statistically model the problem statement and give human-like predictions using ML models.

Depending on the data available on the IP (as the model can have one of the black, gray, or white-box approaches), the verification engineer must select the best ML method (shown in figure 9) suited for the predefined problem statement and collect data features. Selecting an appropriate model will be a trial and error process because the model will work differently for different problem statements and data features.For example, in figure 10, different ML models (model 1, model 2,.... , model N) are trained using the same training data and the performance of all these different ML models are compared on the basis of the same cost function, and the best performing model will be selected for automation. While selecting the appropriate method, the verification engineer must also consider the computational power, platform, and resource available for training.

After analyzing, optimizing, and evaluating the trained Model, the automated part of the verification process will be integrated with the verification steps that do not require further optimization. This integrated part will produce an accelerated ML-based hardware verification method.

One of the critical requirements of using ML in verification is forming an evaluation matrix and objective function to measure the performance of the ML model. This objective function and evaluation matrix must be dynamic, comprehensive, and analog so that reinforcement feedback can be provided for increasing model accuracy.

9.2.2 Challenges of ML-based Verification

The accuracy and performance of the ML model depend hugely on the quality and quantity of collected data, and collecting an adequate number of relevant, unbiased, comprehensive data is always a challenge. It takes numerous man-hours, computational resources, and manual interventions to collect these data. However, most of the collected data are noisy, biased, and not comprehensive in real-life cases. For this reason, pre-processing and data sorting imposes a challenge for hardware verification. Extracting appropriate features from collected data is another demanding task. While building a model, verification engineers need to spend most of their time analyzing the data to get the appropriate specification and feature depth of their model that represents the problem statement the best. Selecting the wrong data feature will result in poor model performance. This is why careful selection of data features is an essential task.

Efficient model selection has one of the highest impacts. The model structure and depth must be coherent and complementary with the data structure and must train itself

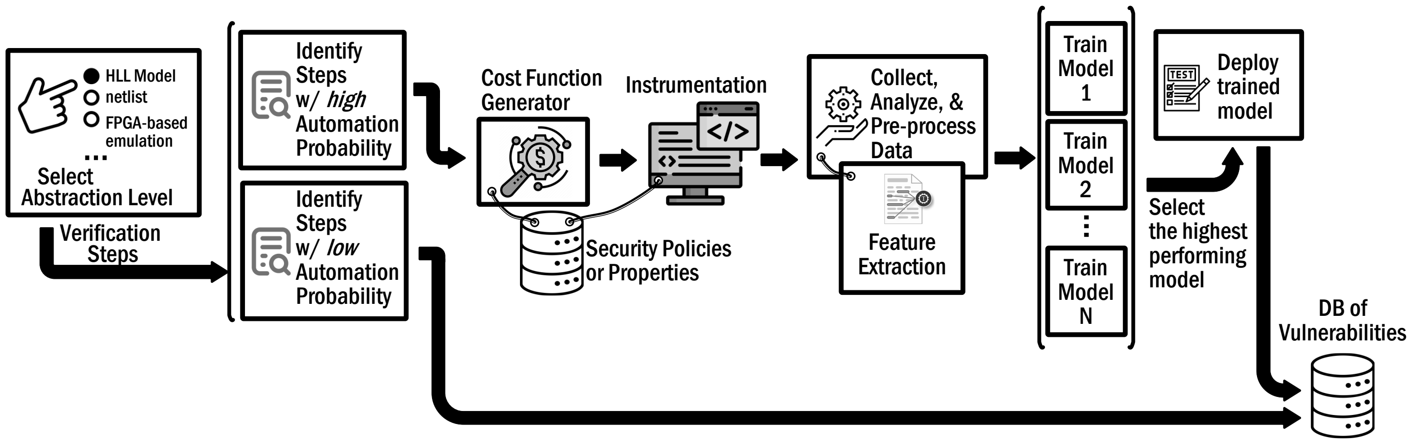

Fig. 10: Workflow of Hardware Verification using ML.

from the given data features. As there are many ML methods and models to choose from; training different models and comparing different methods can give the best-suited model for a particular hardware verification problem. How and where to instrument the code, what platform to use, and what properties to check are also very challenging questions from the verification perspective. Testing, developing, and implementing appropriate objective functions are challenging in real life. For example, one objective function may be appropriate for a certain scenario and irrelevant for another. Developing a generalized, scalable, and appropriate objecting function poses a challenge from this perspective.

10 SoC Security Verification: Future Research Directions/Challenges

In this section, we provide possible directions for further research on SoC security verification using fuzz, penetration, and AI testing. Considering the scalability issue that is ever-increasing by facing with larger and more complex SoCs, introduction of reliable and efficient verification techniques using these self-refinement mechanisms is inevitable. In spite of extensive research efforts in developing scalable and automated security verification techniques over the years, there are still many challenges remain to design secure and trustworthy SoCs. While formal verification tools like JasperGold can be used for known vulnerabilities, here we demonstrated how by appropriately definition of security policies/properties and evolutionary-based cost-functions, smarter approaches such as fuzz, penetration, and AI testing that are capable of generating smart test-cases can extend the scope for both known and unknown SoC security vulnerabilities, and could be a promising direction in SoC security verification. Although recent years show some preliminary usage of these techniques, as discussed previously, many of these techniques still needs significant improvement, in terms of performance as well as coverage. In addition, depending on the security vulnerability, source of vulnerability, and security policy/property, in many cases there will be a need for developing hybrid approaches combining the inherent advantages of different security verification methods to detect a wide variety of security vulnerabilities in emerging SoCs. In addition, while design-time security verification techniques can detect certain types of vulnerabilities, it is infeasible to remove all possible vulnerabilities during pre-silicon security verification. Due to observability constraints in fabricated SoCs, post-silicon security verification (on initial prototypes or FPGA-based emulation) approaches should be considered as well. So, the future research needs to employ both post-silicon and pre-silicon security verification. Finally, security verification tools need to check for various security vulnerabilities across different phases in the design cycle. Specifically per each technique, the following draws some of the possible future research directions:

10.1 Fuzz Testing

While utilizing fuzz testing for SoC verification is a new concept, still the majority of studies explore the best-fitted conventional methods adopted from the software realm. The threat model of software designs is truly different from the hardware. Future work in this field should address this issue by introducing equivalent cost functions and threat surfaces that can be measured and evaluated. Information flow analysis, static code analysis, and dynamic heuristics developed based on simulation flows have proved to be good candidates for the development of metrics, especially in a white-box scenario.

The second problem yet not addressed properly is due to the fact that most recent studies focus on constructing a general approach that works for all possible vulnerabilities in the design. The metrics mentioned above can be more efficient in detecting hardware vulnerabilities if they consider the scope of the vulnerability targeted at each session of testing. For example, a full exploration of the system while generating many test cases for full system security assurance is not necessary when the memory interface is the only untrusted entity in the system. This can help with the scalability of the fuzzing approach while a general method might be more suitable for detecting new vulnerabilities.

The automation required for developing an end-to-end fuzzing approach is another issue that needs further investigation. There still exists a gap for easy-to-use, plug-and-play software based on fuzz testing that can be incorporated as is into other verification tools.

10.2 Pen Testing

The efficacy of the pre-silicon hardware Pen Testing framework, such as the previously-mentioned BPSO-based architecture, is dictated by how effectively the associated cost functions can encapsulate the vulnerability being targeted. The effectiveness of the cost function, in turn, is contingent upon the tester's ability to identify corresponding inputs and effects of the vulnerability. As we mentioned previously, there is an ever growing database of vulnerabilities that the community understands how to trigger (at a high level) and what impacts they might have. However, due to the modular design practices prevalent in the industry today, the visibility and accessibility within the design is getting reduced. This means gaining access to the signals or points of interest may be a challenge especially taking time constraints into consideration. For example, the designer may anticipate that the impact of a vulnerability may be visible through the common bus used in the SoC. However, in a pre-silicon setting, the time taken to simulate the design to appropriate number of clock cycles such that the vulnerability is triggered, may become unacceptably high. In such cases, FPGA emulation of the design can be considered as a promising approach to speed up the process.

We note that the previously-mentioned BPSO based framework assumes no particulars on how the observable point is observed. It can be through simulation, emulation or any other approach preferred by the tester based on time and cost considerations. So long as the tester can provide the algorithm with observed outputs, the algorithm would be able to mutate the input based upon the feedback provided by evaluation of the cost function. Manual formulation of cost functions can become non-scalable if the designers want to test a large variety of vulnerabilities across different platforms and architectures. The best approach to tackle this challenge is to devise an automatic cost function generation methodology based on a general description of the type and scope of the vulnerability as well as microarchitectural implementation details.

10.3 AI Testing

State-of-the-art verification processes require hours and hours of manual intervention but fail to achieve desired coverage goals. Moreover, so many different IPs (hard, soft, firm IPs) are integrated from so many different vendors in a practical system that developing scalable and reusable test cases becomes a daunting task. ML has the potential to overcome both of these issues faced by the traditional verification process. However, ML has its own challenges because the usage of ML for hardware verification is still in its early stages. Once verification engineers pass the initial hurdle of trial and error and start exploring ML for each verification stage, specification of the collected data, extracted best features, appropriate model structure - everything required to build an automated structure will be established. From these established structures and using transfer learning of ML, different IPs with different specifications can be automated to increase verification coverage. Of course, there will always be unique cases where manual interventions will be needed. However, with the help of reinforcement learning, the rate of human intervention requirement will reduce exponentially. One of the possible future direction would be building of this kind of unbiased, scalable, reusable ML model which will increase verification coverage and drastically decrease manual efforts required by the existing process. The best way to implement all of these is to tackle one abstraction level at a time, starting from the RTL level. Verification engineers have to generate an automated method to check the scope of using ML in each step of the verification process and start utilizing ML for the promising steps. Also, generating an evolving cost function with dynamic behaviors is another task. After collecting data and training the model, verification engineers have to check the performance of the whole verification step, with and without ML. If the ML model outperforms the traditional approach with lower overhead, then the ML approach should be established as the standard procedure.

11 Conclusion

In this paper, we re-evaluated the fundamentals and principles of SoC security verification. By reviewing the definitions, requirements, and existing challenges through the SoC security verification, we investigated the possibility of utilization of self-refinement techniques for building a more efficient and scalable methodology for security verification of the design, specifically for complex and heterogeneous SoCs. We discussed the need for further investigation on these techniques, and by assessing the challenges and the possible directions, we hope this article will serve as a navigator for building the next steps in this domain.

References

[1] A. Yeh, "Trends in the Global IC Design Service Market," *DIGITIMES*, 2012.

[2] M. Rostami, F. Koushanfar, and R. Karri, "A primer on hardware security: Models, methods, and metrics," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1283–1295, 2014.

[3] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "Avfsm: A framework for identifying and mitigating vulnerabilities in fsms," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016, pp. 1–6.

[4] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE design & test of computers*, vol. 27, no. 1, pp. 10–25, 2010.

[5] G. K. Contreras, A. Nahiyan, S. Bhunia, D. Forte, and M. Tehranipoor, "Security vulnerability analysis of design-for-test exploits for asset protection in socs," in *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2017, pp. 617–622.

[6] K. Xiao, D. Forte, Y. Jin, R. Karri, S. Bhunia, and M. Tehranipoor, "Hardware trojans: Lessons learned after one decade of research," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 22, no. 1, pp. 1–23, 2016.

[7] Y. Alkabani and F. Koushanfar, "Active Hardware Metering for Intellectual Property Protection and Security," in *USENIX Security Symposium*, 2007, pp. 291–306.

[8] J. Rajendran, M. Sam, O. Sinanoglu, and R. Karri, "Security Analysis of Integrated Circuit Camouflaging," in *Proceedings of the ACM SIGSAC conference on Computer & communications security*, 2013, pp. 709–720.

[9] D. Forte, S. Bhunia, and M. M. Tehranipoor, *Hardware protection through obfuscation*. Springer, 2017.

[10] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "From cryptography to logic locking: A survey on the architecture evolution of secure scan chains," *IEEE Access*, vol. 9, pp. 73 133–73 151, 2021.

[11] N. N. Anandakumar, M. S. Rahman, M. M. M. Rahman, R. Kibria, U. Das, F. Farahmandi, F. Rahman, and M. M. Tehranipoor, "Rethinking watermark: Providing proof of ip ownership in modern socs," *Cryptology ePrint Archive*, 2022.

[12] H. M. Kamali, K. Z. Azar, F. Farahmandi, and M. Tehranipoor, "Advances in logic locking: Past, present, and prospects," *Cryptology ePrint Archive*, 2022.

[13] D. Beyer and T. Lemberger, "Software verification: Testing vs. model checking," in *Haifa Verification Conference*. Springer, 2017, pp. 99–114.

[14] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez, "Moving fast with software verification," in *NASA Formal Methods Symposium*. Springer, 2015, pp. 3–11.

[15] (2018) American fuzzy lop (afl) fuzzer. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[16] W. Chen, S. Ray, J. Bhadra, M. Abadir, and L.-C. Wang, "Challenges and trends in modern soc design verification," *IEEE Design & Test*, vol. 34, no. 5, pp. 7–22, 2017.

[17] K. Z. Azar, H. M. Kamali, H. Homayoun, and A. Sasan, "Smt attack: Next generation attack on obfuscated circuits with capabilities and performance beyond the sat attacks," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 97–122, 2019.

[18] K. Z. Azar, H. M. Kamali, , H. Homayoun, and A. Sasan, "Nngsat: Neural network guided sat attack on logic locked complex structures," in *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE, 2020, pp. 1–9.

[19] K. Z. Azar, H. M. Kamali, F. Farahmandi, M. Tehranipoor, "Warm Up before Circuit De-obfuscation? An Exploration through Bounded-Model-Checkers," in *International Symposium on Hardware Oriented Security and Trust (HOST)*, 2022, pp. 1–4.

[20] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2016, pp. 337–342.

[21] A. Nahiyan, F. Farahmandi, P. Mishra, D. Forte, and M. Tehranipoor, "Security-aware fsm design flow for identifying and mitigating vulnerabilities to fault attacks," *IEEE Transactions on Computer-aided design of integrated circuits and systems*, vol. 38, no. 6, pp. 1003–1016, 2018.

[22] B. Kumar, A. K. Jaiswal, V. Vineesh, and R. Shinde, "Analyzing hardware security properties of processors through model checking," in *2020 33rd International Conference on VLSI Design and 2020 19th International Conference on Embedded Systems (VLSID)*. IEEE, 2020, pp. 107–112.

[23] B. Yuce, N. F. Ghalaty, and P. Schaumont, "Tvvf: Estimating the vulnerability of hardware cryptosystems against timing violation attacks," in *2015 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2015, pp. 72–77.

[24] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 106–117.

[25] A. Nahiyan, J. Park, M. He, Y. Iskander, F. Farahmandi, D. Forte, and M. Tehranipoor, "Script: A cad framework for power side-channel vulnerability assessment using information flow tracking and pattern generation," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 3, pp. 1–27, 2020.

[26] H. Salmani and M. Tehranipoor, "Analyzing circuit vulnerability to hardware trojan insertion at the behavioral level," in *2013 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. IEEE, 2013, pp. 190–195.

[27] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "{HardFails}: Insights into {Software-Exploitable} hardware bugs," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 213–230.

[28] R. Zhang, N. Stanley, C. Griggs, A. Chi, and C. Sturton, "Identifying security critical properties for the dynamic verification of a processor," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, pp. 541–554, 2017.

[29] W. Hu, A. Althoff, A. Ardeshiricham, and R. Kastner, "Towards property driven hardware security," in *2016 17th International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 2016, pp. 51–56.

[30] M. Tiwari, J. K. Oberg, X. Li, J. Valamehr, T. Levin, B. Hardekopf, R. Kastner, F. T. Chong, and T. Sherwood, "Crafting a usable microkernel, processor, and i/o system with strict and provable information flow security," in *2011 38th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2011, pp. 189–199.

[31] R. Kastner, J. Oberg, W. Huy, and A. Irturk, "Enforcing information flow guarantees in reconfigurable systems with mix-trusted ip," in *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. The Steering Committee of The World Congress in Computer Science, Computer ..., 2011, p. 1.

[32] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "On the complexity of generating gate level information flow tracking logic," *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 3, pp. 1067–1080, 2012.

[33] J. Oberg, W. Hu, A. Irturk, M. Tiwari, T. Sherwood, and R. Kastner, "Information flow isolation in i2c and usb," in *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2011, pp. 254–259.

[34] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[35] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.

[36] E. Peeters, "Soc security architecture: Current practices and emerging needs," in *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2015, pp. 1–6.

[37] S. J. Greenwald, "Discussion topic: what is the old security paradigm?" in *Proceedings of the 1998 workshop on New security paradigms*, 1998, pp. 107–118.

[38] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Soc security verification using property checking," in *2019 IEEE International Test Conference (ITC)*. IEEE, 2019, pp. 1–10.

[39] N. Farzana, F. Farahmandi, and M. Tehranipoor, "Soc security properties and rules," *Cryptology ePrint Archive*, 2021.

[40] P. Mishra, M. Tehranipoor, and S. Bhunia, "Security and trust vulnerabilities in third-party ips," in *Hardware IP Security and Trust*. Springer, 2017, pp. 3–14.

[41] M. Gruninger and C. Menzel, "The process specification language (psl) theory and applications," *AI magazine*, vol. 24, no. 3, pp. 63–63, 2003.

[42] S. Vijayaraghavan and M. Ramanathan, *A practical guide for SystemVerilog assertions*. Springer Science & Business Media, 2005.

[43] A. Pnueli, "The temporal logic of programs," in *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*. IEEE, 1977, pp. 46–57.

[44] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri, "Nusmv: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer*, vol. 2, no. 4, pp. 410–425, 2000.

[45] C. Dunbar and G. Qu, "Designing trusted embedded systems from finite state machines," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, pp. 1–20, 2014.

[46] D. M. Anderson, *Design for manufacturability: How to use concurrent engineering to rapidly develop low-cost, high-quality products for lean production*. CRC press, 2020.

[47] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[48] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sep. 2019, pp. 1–15. [Online]. Available: https://www.usenix.org/conference/raid2019/presentation/wang

[49] (2018) honggfuzz. [Online]. Available: http://honggfuzz.com/

[50] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 2123–2138.

[51] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," *arXiv preprint arXiv:2102.02308*, 2021.

[52] J. De Ruiter and E. Poll, "Protocol state fuzzing of {TLS} implementations," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 193–206.

[53] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *NDSS*, vol. 8, 2008, pp. 151–166.

[54] R. L. Seagle Jr, "A framework for file format fuzzing with genetic algorithms," 2012.

[55] M. M. Hossain, F. Farahmandi, M. Tehranipoor, and F. Rahman, "Boft: Exploitable buffer overflow detection by information flow tracking," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2021, pp. 1126–1129.

[56] [Online]. Available: https://www.veripool.org/verilator/

[57] S. K. Muduli, G. Takhar, and P. Subramanyan, "Hyperfuzzing for soc security validation," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[58] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz, "Medusa: Microarchitectural data leakage via automated attack synthesis," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1427–1444.

[59] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "Specfuzz: Bringing spectre-type vulnerabilities to the surface," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1481–1498.

[60] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 176–187.

[61] Y. Xiao, Y. Zhang, and R. Teodorescu, "Speechminer: A framework for investigating and measuring speculative execution vulnerabilities," *arXiv preprint arXiv:1912.00329*, 2019.

[62] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "Introspectre: a pre-silicon framework for discovery and analysis of transient execution vulnerabilities," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 874–887.

[63] A. Tyagi, A. Crump, A.-R. Sadeghi, G. Persyn, J. Rajendran, P. Jauernig, and R. Kande, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," 01 2022.

[64] A. Basak, S. Bhunia, T. Tkacik, and S. Ray, "Security assurance for system-on-chip designs with untrusted ips," *IEEE Transactions on Information Forensics and Security*, vol. 12, no. 7, pp. 1515–1528, 2017.

[65] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial soc design," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–2.

[66] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.

[67] S. Shah and B. M. Mehtre, "An overview of vulnerability assessment and penetration testing techniques," *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 27–49, 2015.

[68] H. H. Thompson, "Application penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 66–69, 2005.

[69] W. Knowles, A. Baron, and T. McGarr, "The simulated security assessment ecosystem: Does penetration testing need standardisation?" *Computers & Security*, vol. 62, pp. 296–316, 2016.

[70] [Online]. Available: https://owasp.org/www-project-web-security-testing-guide/latest/3-The_OWASP_Testing_Framework/1-Penetration_Testing_Methodologies

[75] H. Khattri, N. K. V. Mangipudi, and S. Mandujano, "Hsdl: A security development lifecycle for hardware technologies," in *2012 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 2012, pp. 116–121.

[71] [Online]. Available: https://www.isecom.org/OSSTMM.3.pdf

[72] [Online]. Available: https://www.crest-approved.org/what-is-star-fs/index.html

[73] R. Baloch, *Ethical hacking and penetration testing guide*. Auerbach Publications, 2017.

[74] T. Dimkov, A. Van Cleeff, W. Pieters, and P. Hartel, "Two methodologies for physical penetration testing using social engineering," in *Proceedings of the 26th annual computer security applications conference*, 2010, pp. 399–408.

[76] M. He, J. Park, A. Nahiyan, A. Vassilev, Y. Jin, and M. Tehranipoor, "Rtl-psc: Automated power side-channel leakage assessment at register-transfer level," in *2019 IEEE 37th VLSI Test Symposium (VTS)*. IEEE, 2019, pp. 1–6.

[77] H. Wang, H. Li, F. Rahman, M. M. Tehranipoor, and F. Farahmandi, "Sofi: Security property-driven vulnerability assessments of ics against fault-injection attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[78] J. Kennedy and R. C. Eberhart, "A discrete binary version of the particle swarm algorithm," in *1997 IEEE International conference on systems, man, and cybernetics. Computational cybernetics and simulation*, vol. 5. IEEE, 1997, pp. 4104–4108.

[79] H. Salmani, M. Tehranipoor, and R. Karri, "On design vulnerability analysis and trust benchmarks development," in *2013 IEEE 31st international conference on computer design (ICCD)*. IEEE, 2013, pp. 471–474.

[80] B. Shakya, T. He, H. Salmani, D. Forte, S. Bhunia, and M. Tehranipoor, "Benchmarking of hardware trojans and maliciously affected circuits," *Journal of Hardware and Systems Security*, vol. 1, no. 1, pp. 85–102, 2017.

[81] X. Zhang and M. Tehranipoor, "Case study: Detecting hardware trojans in third-party digital ip cores," in *2011 IEEE International Symposium on Hardware-Oriented Security and Trust*. IEEE, 2011, pp. 67–70.

[82] X. Zhang and H. Salmani, "Integrated circuit authentication: hardware trojans and counterfeit detection." 2014.

[83] A. Nahiyan, M. Sadi, R. Vittal, G. Contreras, D. Forte, and M. Tehranipoor, "Hardware trojan detection through information flow security verification," in *2017 IEEE International Test Conference (ITC)*. IEEE, 2017, pp. 1–10.

[84] S. Bhunia and M. Tehranipoor, "The hardware trojan war," *Cham,, Switzerland: Springer*, 2018.

[85] S. Gogri, P. Joshi, P. Vurikiti, N. Fern, M. Quinn, and J. Valamehr, "Texas a&m hackin'aggies' security verification strategies for the 2019 hack@ dac competition," *IEEE Design & Test*, vol. 38, no. 1, pp. 30–38, 2020.

[86] F. Farahmandi, Y. Huang, and P. Mishra, *System-on-Chip Security*. Springer, 2020.

[87] H. Kaeslin, *Digital integrated circuit design: from VLSI architectures to CMOS fabrication*. Cambridge University Press, 2008.

[88] W. Hughes, S. Srinivasan, R. Suvarna, and M. Kulkarni, "Optimizing design verification using machine learning: Doing better than random," *arXiv preprint arXiv:1909.13168*, 2019.

[89] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu, "Boosting verification by automatic tuning of decision procedures," in *Formal Methods in Computer Aided Design (FMCAD'07)*. IEEE, 2007, pp. 27–34.

[90] P. Gaur, S. S. Rout, and S. Deb, "Efficient hardware verification using machine learning approach," in *2019 IEEE International Symposium on Smart Electronic Systems (iSES)(Formerly iNiS)*. IEEE, 2019, pp. 168–171.