A Survey on SoC Security Verification Methods at the Pre-silicon Stage

Rasheed Kibria, Farimah Farahmandi, and Mark Tehranipoor
Department of Electrical and Computer Engineering
University of Florida, Gainesville, FL, USA
Email: rasheed.kibria@ufl.edu, farimah@ece.ufl.edu, tehranipoor@ece.ufl.edu

***Abstract*—This paper presents a survey of the state-of-the-art pre-silicon security verification techniques for System-on-Chip (SoC) designs, focusing on ensuring that designs, implemented in hardware description languages (HDLs) and synthesized circuits, meet security requirements before fabrication in semiconductor foundries. Due to several factors, pre-silicon security verification has become an essential yet challenging aspect of the SoC hardware lifecycle. The modern SoC design process often adheres to a design reuse philosophy, integrating multiple functional blocks or Intellectual Property (IP) cores sourced from various vendors onto a single chip. While beneficial for reducing costs and accelerating time-to-market, this approach introduces numerous untrustworthy third-party entities into the supply chain. It increases the potential for introducing security vulnerabilities significantly. Additionally, hardware fabrication, assembly, and testing are frequently outsourced to third-party entities, further exacerbating security risks. Moreover, the growing complexity of SoC designs leads to unanticipated interactions between hardware and software layers, creating potential gateways for attackers to exploit and steal confidential information from devices. In response to these challenges, recent years have seen a surge in the development of innovative SoC security verification techniques. This survey provides an overview of these methods, their high-level working principles, strengths, and weaknesses. By understanding these techniques, designers can better evaluate their effectiveness and select the most appropriate methods aligned with the specific security objectives for their SoC designs.**

***Keywords*-SoC security verification, Code review, Static code analysis, Property-driven formal methods, Penetration testing, Fuzzing.**

I. Introduction

System-on-Chip (SoC) represents a platform where all electronic system components, or intellectual property (IP) blocks, are integrated into a single chip. Modern SoCs can incorporate billions of transistors within a compact area of around one hundred square millimeters. This high level of integration has made SoCs a cornerstone in modern technology and electronics. The SoC's capability to accommodate multiple functional blocks on a single platform minimizes interface and interconnection delays, leading to superior performance compared to traditional integrated circuits [1–4]. Additionally, SoCs are cost-effective in producing, conserving energy, and saving space, further contributing to their widespread adoption. An example of a modern SoC is shown in Fig. 1, which comprises numerous IPs with diverse functionalities (analog, memory, digital, etc.).

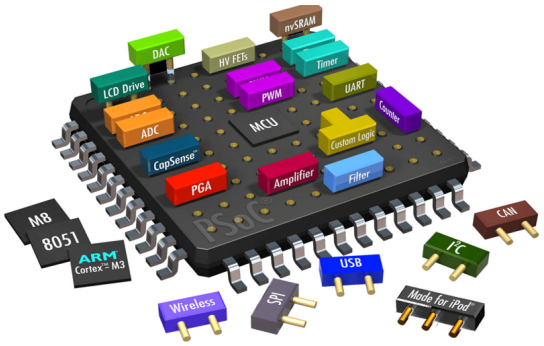

Fig. 1: Example of a modern System-on-Chip (SoC).

The popularity of SoCs is primarily due to their versatility and efficiency. Their compact size makes them ideal for portable devices such as smartphones, cameras, tablets, and other wireless technologies. Furthermore, the ability of SoCs to be integrated into Internet of Things (IoT) devices extends their application to critical national infrastructures, including defense, finance, and transportation sectors. This adaptability enhances the functionality and reliability of these systems and underscores the significance of SoCs in advancing modern technology and maintaining critical infrastructure security and efficiency [1, 2, 5].

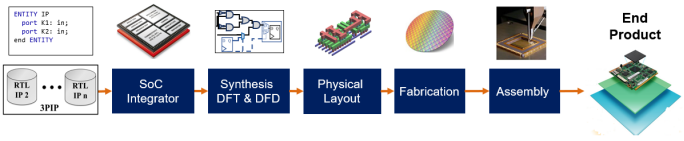

Fig. 2: Modern SoC design flow.

The SoC design flow is illustrated in Fig. 2. It outlines the steps in designing a modern SoC at a very high level. Due to the pressures of shrinking time-to-market and rising production costs, it has become nearly impossible for a single entity to manage the entire process of designing, developing, and fabricating an SoC independently [6–8]. Consequently, the semiconductor industry has transitioned to a horizontal model. The SoC integrator obtains IP blocks from various third-party vendors in this model [5, 7, 8]. The SoC designer then integrates these external IPs with their proprietary IPs to develop the entire SoC's register-transfer level (RTL) description. Once the RTL code is finalized after verification, the SoC undergoes synthesis, converting the high-level design into a gate-level netlist. This netlist enters the design-for-test (DFT)

insertion phase to ensure the SoC can be effectively tested for faults. Often, this phase is outsourced to specialized third-party entities due to its complexity, cost, and the need for specific expertise. After the DFT insertion, the gate-level netlist is translated into a physical layout, leading to the final GDSII file. This file is then sent to a foundry where the physical chip for the SoC is manufactured [5].

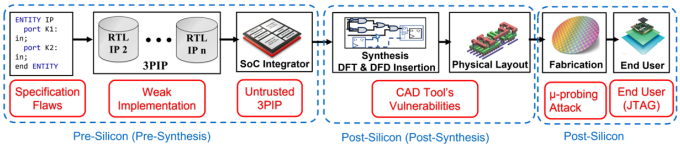

Fig. 3: Sources of security vulnerabilities in the modern SoC design flow.

In the state-of-the-art chip design lifecycle, vulnerabilities are pervasive throughout the entire SoC design process, presenting numerous potential attack surfaces. Attackers might exploit these vulnerabilities to gain unauthorized access to sensitive information once the SoC is deployed in the field. The SoC design flow often begins with defining specifications that may not be security-aware [5]. Even though these specifications are flawless, vulnerabilities can still arise during the implementation phase. Additionally, external vendors' third-party intellectual property (3PIP) may contain malicious functionalities intentionally embedded by rogue employees within the design house [9–11]. Moreover, existing computer-aided design (CAD) tools used during the design translation phase prioritize optimization for area, power, and performance, often at the expense of security considerations. These tools are not equipped to address or mitigate security vulnerabilities, leaving the design susceptible to various threats [12–14]. For instance, an attacker can exploit the JTAG interface [15–17] or inject faults into the final product to execute malicious actions [18, 19]. Since SoCs are integral components in numerous computational devices, these security vulnerabilities may pose significant risks to the overall system security. In Fig. 3, the sources of potential security vulnerabilities are shown, which exist in the modern SoC design flow.

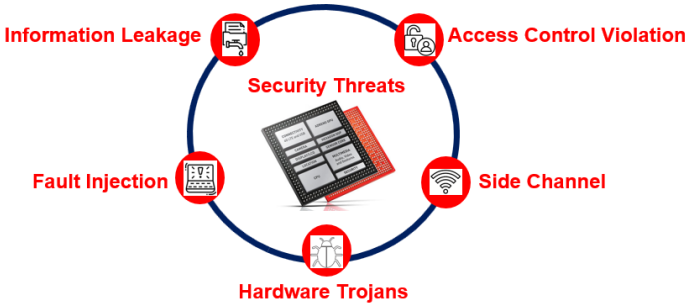

Fig. 4: Some examples of attacks on hardware.

Some examples of attack scenarios on hardware are presented in Fig. 4. These attacks may exploit the security vulnerabilities discussed earlier. Information leakage involves the unauthorized disclosure of confidential information to untrusted entities. This leakage can be intentionally introduced by third-party intellectual property (3PIP) vendors or unintentionally induced by computer-aided design (CAD) tools [20–22]. Inherent hardware vulnerabilities, such as specification flaws or weak implementations, can be exploited through fault injection techniques. Attackers may use methods like power or clock glitches, temperature variations, and light, laser, or electromagnetic emissions to induce faults in the hardware [18, 19, 23–25]. These faults can lead to the leakage of sensitive information, including cryptographic keys, user credentials, and passwords, or result in the unwanted modification of security-critical data, compromising the system's confidentiality and integrity. Hardware Trojans represent another significant threat, involving malicious changes to the design by rogue employees within the design house or foundry. These Trojans are intentionally embedded to leak secret information (resulting in confidentiality violations) or to alter sensitive data (leading to integrity violations) [9–11]. Side-channel attacks pose a unique challenge as they do not require any design modification. Instead, they exploit covert channels or physical parameters, such as power consumption, electromagnetic emissions, or timing information, to extract sensitive information [26–28]. Additionally, attackers may exploit design-for-test (DFT) and design-for-debug (DFD) structures, such as scan chains, compression, and JTAG interfaces, to gain unauthorized access to the design [15–17, 29]. By maliciously exploiting these structures' controllability and observability features, attackers can violate access controls, undermining the security of the SoC.

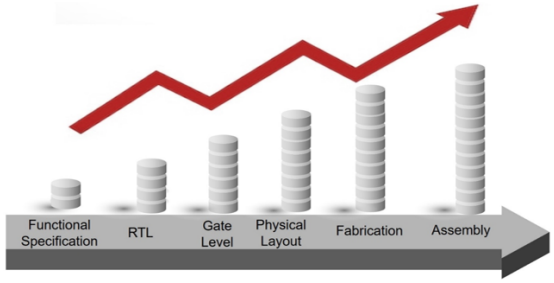

Fig. 5: Relative cost of design modification: *Rule of 10*.

A comprehensive security verification methodology must be adopted to address potential security vulnerabilities and threats effectively. This security verification process should begin at the earliest stages of the SoC design lifecycle to ensure the timely detection and mitigation of security issues [1, 2, 12]. Early-stage security verification is crucial because addressing vulnerabilities becomes significantly more challenging and costly as the design progresses. During the post-silicon stages, designers have limited flexibility to update or modify the SoC design, making it quite difficult to resolve security issues that originate from earlier design phases. The economic implications of performing security verification at the later stages of the SoC design lifecycle are underscored by the *Rule of 10*. It states that modifying a design at later stages of the SoC design flow is ten times more expensive than making changes during earlier stages [1, 2, 12, 30]. A pictorial representation of this notion is shown in Fig. 5. This principle highlights the exponential cost increase associated with identifying and fixing security vulnerabilities during the later design and fabrication

processes. By integrating security verification at the initial stages of the SoC design lifecycle, designers can ensure that vulnerabilities are addressed promptly and effectively, thereby enhancing the security of the final product. Specifically, the primary objectives of this paper are as follows-

- Providing an overview of the state-of-the-art security verification methodologies for SoC designs;
- Discussing various types of vulnerability identification techniques at the pre-silicon stage of the design lifecycle;
- Assessing the advantages and limitations of the techniques, providing their effectiveness in identifying potential security issues during the SoC design process.

The rest of the paper is organized as follows. Section II provides a detailed overview of the challenges and limitations associated with traditional security verification methods. It also presents a high-level overview of state-of-the-art SoC security verification techniques. Section III is focused on code review-based techniques for identifying potential security vulnerabilities present in the RTL codes of an SoC. Section IV presents the security property-based formal verification methodology and its advantages and limitations. Section V illustrates the dynamic security verification methods such as penetration and fuzz testing with their pros and cons. Finally, in Section VI, we conclude with the summary of this survey.

II. SoC Security Verification: Challenges and Techniques

This section first presents the common challenges encountered while performing an SoC security verification. Next, it illustrates the limitations of traditional approaches adopted for security verification. Finally, the section will conclude by providing a high-level overview of the state-of-the-art security verification techniques.

A. Challenges in SoC Security Verification

Although security vulnerabilities can be introduced at various stages of the SoC design lifecycle, effectively counteracting these vulnerabilities remains a tremendous challenge. Moreover, SoC security verification and assurance to identify security issues and mitigate vulnerabilities is a huge challenge and a promising research domain. Challenges associated with the security verification of an SoC are illustrated in Fig. 6. The globalization of SoC production has worsened this issue since the providers often do not guarantee the security of custom and legacy third-party IPs (3PIPs). SoCs integrate IP cores from various domains, including analog, digital, re-configurable, and fabric, all within a single chip. Comprehensive security assurance necessitates the verification of each IP core and thorough cross-domain checks. However, the increasing number of transistors and the growing complexity of SoCs further complicate the security verification process [1, 2, 5, 12, 15].

Amidst the complexity and the pressure to reduce time-to-market for SoCs, design houses primarily focus on optimizing power, performance, and area (PPA), as well as conducting testing and packaging. This focus often neglects the crucial security verification process. Furthermore, there is a lack of security awareness among design engineers. The designers

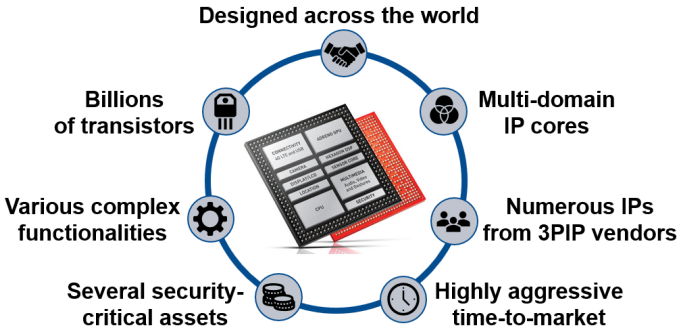

Fig. 6: Challenges encountered while performing security verification of a SoC.

may not fully recognize the importance of protecting security-critical assets. These assets, which can vary widely, are susceptible to numerous attacks. Overcoming these challenges necessitates a coordinated effort. It is imperative to integrate security verification into the entire SoC design process. This approach will ensure that all potential vulnerabilities are identified and mitigated from the earliest stages of development [1, 2, 12, 31].

B. Limitations of Traditional Security Verification Methods

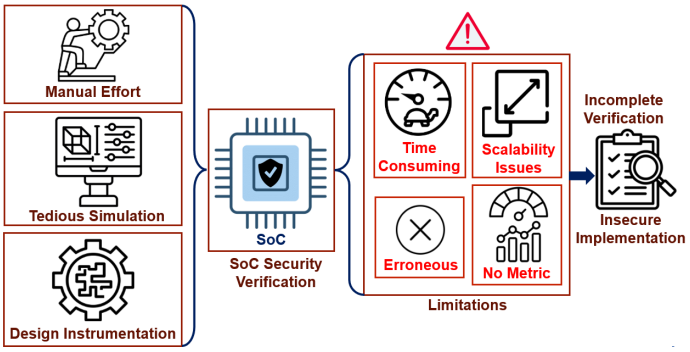

Fig. 7: Limitations of traditional security verification methods.

Despite the increasing importance of security verification in the SoC design lifecycle, there has been relatively little progress in this domain [31]. There are several limitations associated with traditional security verification methods. Such limitations are illustrated in Fig. 7. Traditional security verification methods are primarily manual. It requires engineers to carefully examine thousands of lines of code to identify potential security vulnerabilities. This process is time-consuming and prone to error, making it an inefficient and unreliable approach. Several alternative methods are proposed in existing literature, such as mining anomalies from simulation traces [32, 33] and instrumenting designs with shadow logic to compare behaviors against a golden reference [34–36]. However, these approaches face significant limitations. As the size and complexity of designs increase, these methods encounter state-space complexity and lack scalability. Such drawbacks make them less effective for large-scale and modern SoCs.

Furthermore, there is a lack of standardized metrics for assessing the security of an SoC. Without appropriate metrics, it is difficult to quantify the security assurance provided by

existing techniques or to compare the effectiveness of different approaches. This lack of proper metric definition and formulation emphasizes the need for further research and development in security verification methods. The establishment of scalable and automated security verification techniques, along with appropriate security metrics, is crucial [31]. These techniques are necessary to ensure that SoC designs can effectively withstand emerging security threats. Until these issues are adequately addressed, the security verification of SoCs will remain as a challenging aspect of the SoC design lifecycle.

C. Overview of Pre-silicon Security Verification Techniques

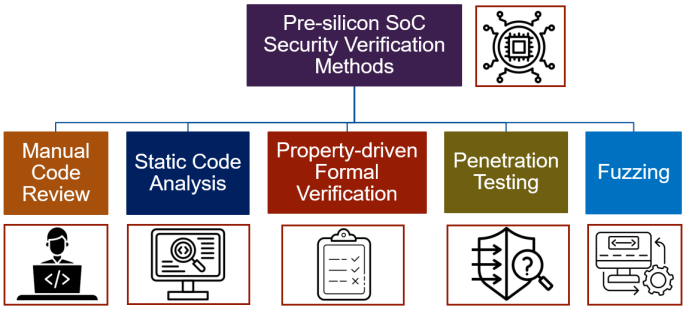

Fig. 8: Pre-silicon SoC security verification techniques.

Numerous SoC security verification techniques have been proposed for the pre-silicon stage of the design lifecycle. Such techniques are shown in Fig. 8. These techniques can be broadly classified into two categories: static security verification and dynamic security verification. The detailed taxonomy is illustrated in Fig. 9. Static security verification methods focus on analyzing the RTL codes of an SoC design without applying any test vectors. This analysis can be conducted through manual or automated code reviews, where the RTL code is examined for potential security issues and vulnerabilities. Moreover, static verification-based methods include property-driven formal verification techniques. In such techniques, a mathematical representation of the design is created and verified against various security properties and requirements.

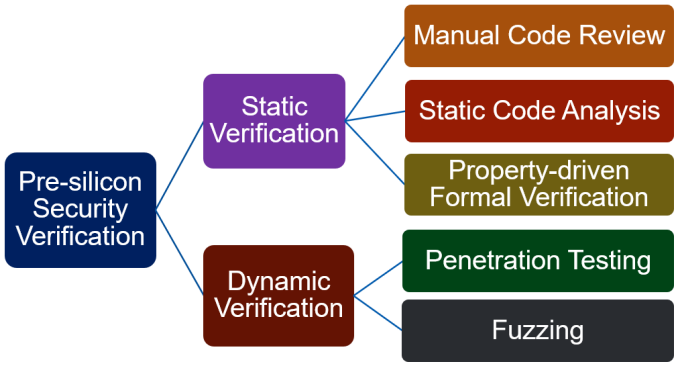

Fig. 9: Taxonomy of SoC security verification techniques at the pre-silicon stage.

On the contrary, dynamic security verification techniques require the application of test vectors to perform security assessments on the SoC design. These methods can simulate real-world scenarios to evaluate the system's response and identify potential security vulnerabilities. Penetration testing and fuzz testing (fuzzing) are prominent examples of dynamic verification techniques. By generating and applying appropriate test vectors, dynamic verification can provide deeper insights into how the SoC behaves under different conditions, capturing runtime interactions with software that static methods may not reveal. In conclusion, these techniques form a comprehensive security verification framework for a large-scale and complex SoC design.

III. Code Review for Security Verification

This section will provide an overview of code review-based techniques for SoC security verification. First, we will present a simple classification of code review-based methods. Next, we will briefly discuss such methods and their advantages and limitations.

A. Introduction

The primary objective of code review for SoC security verification is to identify and mitigate potential security vulnerabilities in the RTL code base. By scrutinizing the RTL codes, code reviews target to reduce the risk of possible security breaches and prevent the exploitation of vulnerabilities. This process is critical in ensuring the overall security and integrity of the SoC, especially as these components become integral to numerous applications, ranging from consumer electronics to critical infrastructures. Code reviews' effectiveness in enhancing security depends on the thoroughness and expertise applied during the review process. A comprehensive code review specifically targets security-related issues within the RTL codes of an SoC design. This process includes examining control logic implementation, often realized using finite state machines (FSMs). Security vulnerabilities can arise from insecure encoding of FSM states [12–14], unsafe implementation of comparison logic, inadvertent or malicious modification of sensitive information [10, 11], and other bad RTL coding practices [37]. Identifying such issues requires a detailed understanding of the SoC design's functional and security aspects. Addressing these vulnerabilities early in the design process can significantly minimize the risk of potential security breaches.

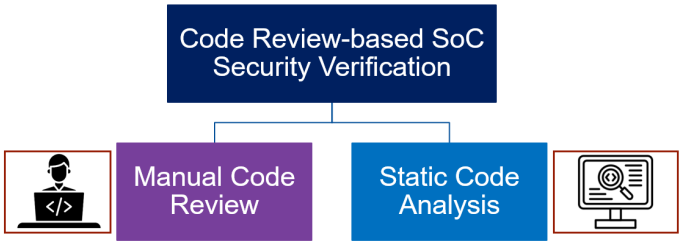

Fig. 10: Classification of code review-based techniques for SoC security verification.

The involvement of hardware security experts or specialized security teams enhances the effectiveness of a code review process. Such professionals possess a good understanding of security principles and potential threats, enabling them to

identify security vulnerabilities that typical RTL designers and verification engineers might overlook. Their expertise in hardware security allows them to identify nuanced security issues and implement appropriate mitigation strategies. Therefore, including security experts in the code review process ensures a more comprehensive and effective evaluation of the SoC's security status. Consequently, it leads to developing SoC designs that are more secure and resilient against various attacks. On a very high level, code review-based SoC security verification techniques can be classified into two broad categories: manual code review and static code analysis. It is depicted in Fig. 10. The following two sections briefly overview these categories and their advantages and limitations.

B. Manual Code Review

1) Overview: Manual code review is a process wherein a human reviewer scrutinizes the RTL source codes of an SoC line by line to identify potential security vulnerabilities that may result from insecure coding practices [38]. It is typically done by an experienced RTL designer who understands security well. This examination involves a detailed inspection of the entire SoC's RTL codebase. The primary objective is to detect and mitigate security risks before they can be exploited. This process requires a deep understanding of RTL design principles and security concerns to uncover potential security vulnerabilities that might compromise the system. The effectiveness of the manual code review largely depends on the expertise and experience of the reviewer [39]. Therefore, it is typically done by a senior RTL designer with extensive knowledge and a strong background in hardware security. Such expertise and experience are required to identify complex security issues that less experienced RTL designers might overlook. This review process is crucial since it allows for the early detection of security vulnerabilities. Moreover, it reduces the possibility of security breaches in the later stages of development or after the SoC is deployed in the field.

2) Methodology: The manual code review process for SoC security verification begins with the code reviewer selecting specific portions of the RTL codebase for thorough scrutiny. Often, a representative sample of the codebase is chosen to effectively identify and address potential security vulnerabilities early in the SoC design lifecycle. This approach allows the reviewer to focus on critical areas susceptible to vulnerabilities. Following the selection, a plan for conducting the security verification is developed. This plan includes identifying potential security threats that may arise during the SoC design lifecycle and determining the specific areas and methods for examining the RTL code. This initial stage is crucial for setting the scope and ensuring a comprehensive and highly structured review process.

The review process involves a careful line-by-line inspection of the RTL code to uncover any security issues. The reviewer must be familiar with RTL coding standards and secure coding practices. It is because such expertise is essential for identifying violations and weaknesses existing in the RTL codes. Utilizing a checklist of known vulnerabilities, common security issues, and risks further aids in this code review process. Upon completing the review, the reviewer compiles a comprehensive report summarizing the findings and providing improvement recommendations. This report is crucial for RTL designers since it will guide them in making necessary modifications to mitigate potential security vulnerabilities. Addressing such issues early in the SoC design lifecycle can reduce the likelihood of successful attacks and security breaches in later stages.

3) Advantages: The manual code review-based method for SoC security verification has some advantages, as described below.

In-depth Understanding: Manual code review provides a unique opportunity for the reviewer to develop an in-depth understanding of the RTL codes and the complex functionalities embedded within the SoC design. Such a grasp of the code is critical in identifying complex vulnerabilities that automated static analysis tools may overlook. Unlike automated tools, which often rely on predefined patterns and rules, a human reviewer can interpret the context behind the code. Therefore, subtle security issues that require context and a deep understanding might be detected. This feature is essential for uncovering complex vulnerabilities that can pose significant risks if undetected.

Furthermore, manual code review enables the reviewer to analyze the logic behind the code. It leads to more accurate assessments of potential security risks. By examining how different parts of the code interact and understanding the underlying design intentions, the reviewer can identify security flaws that might not be apparent through automated analysis alone. This approach allows for a more thorough evaluation of the security status of the SoC. The underlying reason is that the reviewer can consider the code's functional and security aspects. Consequently, manual code reviews contribute to a more robust security verification process, ensuring potential vulnerabilities are identified and addressed effectively early in the SoC design lifecycle.

Customized Solutions: When a potential security vulnerability is identified in an SoC design during a manual code review, the collaborative efforts of the designers and security experts become crucial in developing tailored solutions that address the specific needs of the application. This collaboration between the design and security teams ensures that security considerations are integrated into the RTL design phase without compromising the primary focus on PPA requirements. By leveraging the expertise of both groups, the team can develop more robust and effective fixes that are finely tuned to the application of the SoC. This approach differs from automated static analysis tools, which often provide generic solutions that may not fully align with the unique requirements of each application.

This collaborative approach to addressing security vulnerabilities results in customizable solutions. Designers bring their understanding of the design architecture and functionality, while security experts contribute with their specialized knowledge of potential threats and mitigation strategies. Together, they can thoroughly assess the impact of proposed fixes on security and PPA metrics, ensuring that the solutions are secure and efficient. This method provides an easily adaptable

security verification process, leading to a more secure and high-performing SoC design.

Training Opportunities: Manual code review offers an excellent opportunity for designers to learn from security experts. It facilitates knowledge exchange on secure RTL coding practices and enhances their skills. This collaborative training environment enables designers to gain insights into potential security vulnerabilities and effective mitigation strategies. It improves their understanding of security principles within the SoC design lifecycle. As designers become more proficient in identifying and addressing security issues, the overall security status of the SoC is significantly improved. Moreover, this process creates a security-aware design team equipped with the expertise to integrate robust security measures throughout the design phases, leading to more resilient and secure SoC implementations.

4) Limitations: Besides the advantages, manual code review for SoC security verification has several limitations, illustrated below.

Time Consuming: The foundation of the manual code review process for SoC security verification lies in its nature, where the code is read manually, line by line. This technique makes it easier to identify potential security vulnerabilities embedded within the RTL codes of an SoC design. By carefully examining each line, reviewers can detect subtle security issues that automated static tools might miss. However, such a process is inherently time-consuming due to the complexity and volume of RTL code that needs to be reviewed. Moreover, effective manual code reviews require RTL designers to be proficient in coding and have a deep understanding of security principles. Designers must undergo proper training and practice to achieve the necessary expertise level. Acquiring the skills to conduct manual code reviews effectively can take up to a year of dedicated practice and experience. In summary, this security verification technique is lengthy, repetitive, and tedious, requiring significant time and effort to identify potential security vulnerabilities present in the RTL design of an SoC.

Subjectivity: Since code reviews are conducted on an individual basis, there is a possibility that some security vulnerabilities might get overlooked by the human eye. The primary goal of these reviews is to uncover security issues, but maintaining high consistency across reviews is also critical. Inconsistencies can arise due to the subjective nature of manual code reviews, as different reviewers may have varying levels of expertise, experience, and perspectives on what constitutes a potential security vulnerability. This subjectivity can lead to varying results and potentially leave some vulnerabilities undetected, compromising the system's overall security. The effectiveness of manual code reviews heavily depends on the reviewer's expertise and experience. While reading through the code can identify many problems, the most subtle security issues are often the easiest to miss. These issues may include vulnerabilities arising from insecure coding practices, complex logic, or specific implementation flaws that require a deep understanding of the RTL codebase and security threats. To conclude, the subjective nature of manual reviews can result in overlooked security vulnerabilities in the SoC design lifecycle.

Scalability: Manual code review is associated with scalability issues because it relies on substantial human resources. This requirement can be inconvenient for organizations with limited resources or rapidly expanding RTL codebases. This code review process demands a team of skilled and experienced RTL designers capable of carefully analyzing complex RTL codes to identify potential security vulnerabilities. Moreover, developing a proficient manual code review team is a long-term investment that involves significant training and experience. Engineers must understand RTL design principles and security practices, which may take years to achieve. The process of becoming a good code reviewer requires extensive knowledge and exposure to a variety of security issues and coding practices. Therefore, developing security-oriented code review teams comes with substantial expenses in direct costs and the time and resources dedicated to training and development.

C. Static Code Analysis

1) Overview: Static or source code analysis is an integral component of the code review process for SoC security verification. This white-box security verification methodology is conducted during the RTL design and implementation phase of the SoC design lifecycle. Unlike dynamic verification-based techniques, which require appropriate test vectors, static code analysis thoroughly examines the RTL source codes in a static manner, meaning it analyzes the code without executing it. By scrutinizing the RTL code's structure, logic, and adherence to predefined security rules and patterns, static analysis-based solutions can identify potential security vulnerabilities that may compromise the system's security. This approach is essential in mitigating potential threats early in the design cycle [40].

One of the most notable advantages of static code analysis-based techniques is that they provide immediate feedback to the RTL designers. Designers can promptly address these potential security vulnerabilities by identifying security issues as they are introduced into the RTL codebase. It ensures the RTL design remains secure and resilient against probable attacks on the final implementation. Such feedback is more advantageous than discovering vulnerabilities later in the design cycle, where fixing issues can be significantly more complex and costly. Early detection and remediation of security issues help maintain the overall quality of the SoC design and reduce the risk of severe security breaches after deployment in the field [41]. Therefore, incorporating static code analysis into the SoC design process can significantly enhance the SoC's security posture.

2) Methodology: The high-level overview of static code analysis-based SoC security verification is presented in Fig. 11. As shown in the figure, the RTL source codes of an SoC are first compiled using a Hardware Description Language (HDL) compiler. The HDL compiler is critical in the static analysis-based methodology for SoC security verification. Typically, an HDL compiler is specialized for a specific hardware description language, such as Verilog, SystemVerilog, or VHDL. For instance, a good SystemVerilog compiler can compile

RTL codes written in SystemVerilog language without issue. Selecting a high-quality HDL compiler is a crucial initial step in developing a robust static analysis-based solution for performing security verification. The reason is that the effectiveness and accuracy of the static analyzer heavily depend on the quality of the HDL compiler on top of which it is developed. The primary function of the HDL compiler is to parse the input RTL source codes, identifying and resolving any syntax issues present in the codebase. This process ensures the code is correctly structured and ready for further analysis.

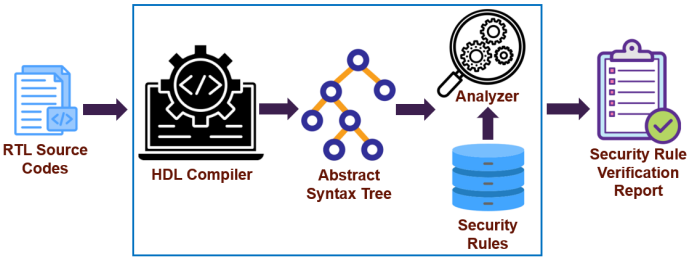

Fig. 11: High-level overview of static code analysis-based security verification.

Upon successful compilation, the HDL compiler generates an Abstract Syntax Tree (AST) representation of the RTL source codes. The AST serves as an intermediate representation, organizing all information from the RTL codes into a structured, tree-like format. This representation is crucial for subsequent processing tasks since it transforms the RTL codes into a well-structured form, eliminating extraneous information that can complicate the analysis. Directly analyzing raw RTL source codes is challenging due to unnecessary details, which may complicate the verification process. However, the AST provides a concise, well-suited structure for further analysis to detect potential security vulnerabilities. Therefore, using an AST representation significantly enhances the efficiency and effectiveness of the static analysis-based technique.

The AST representation of the RTL source codes is subsequently fed into the analyzer, which is the central component of the static analysis-based methodology. This analyzer is crucial since it performs the security verification of the RTL design. The analyzer requires additional input, such as a database of security rules, to accomplish this task effectively. Fundamentally, these security rules are guidelines for identifying insecure RTL coding practices and design flaws that might lead to potential security vulnerabilities. These rules are often derived from popular vulnerability databases such as the Common Weakness Enumeration (CWE), Common Vulnerabilities and Exposures (CVE), and Trust-Hub's security property database. The role of these security rules is pivotal since they aid in formulating specific patterns, templates, or coding styles that may make an RTL design susceptible to various known attacks.

When the analyzer identifies these insecure patterns, templates, or coding styles in the RTL design, it flags these instances as violations of the associated security rules. These flagged violations are then compiled into a report and presented in a human-readable file. This process allows the RTL designers to review and address the identified issues. This report makes the designers aware of potential security vulnerabilities existing in the RTL design of an SoC. Depending on the severity, context, and use case, designers can decide whether to waive certain non-critical violations or implement fixes for the most critical issues. The designers should carefully review the security rule verification report along with the RTL designs. This approach will ensure that the most critical violations are addressed based on the design's specific context and use case. A well-developed static analysis-based solution makes the designers aware of potential security vulnerabilities in their code and provides guidance on appropriate fixes to resolve such issues.

3) Advantages: The static code analysis-based method for SoC security verification has several advantages listed below.

Early Identification of Security Issues: Automated static code analysis-based techniques play a crucial role in identifying security issues within the RTL codes of an SoC design at the earliest stages of the lifecycle. These methods extensively analyze the RTL codes without executing those. Hence, it enables the detection of potential security vulnerabilities such as coding issues, insecure practices, and logical flaws. By incorporating automated static analysis into the initial phases of the design process, potential security threats can be identified promptly. This early detection is crucial as it allows designers to address and mitigate these vulnerabilities before they become deeply embedded in the final product. Moreover, these techniques can operate continuously and consistently in an automated fashion. Therefore, a comprehensive and repeatable method can be achieved for SoC security verification at the RTL design stage, which complements manual code reviews. Automated static analysis ensures a thorough examination across the entire RTL codebase, contributing to a more secure SoC design lifecycle.

Cost Reduction: Static code analysis-based techniques offer a notable advantage in the secure design lifecycle by issuing warnings about potential security vulnerabilities early in the design stage. Unlike manual code reviews, which require extensive expertise and skills, these automated solutions can promptly and efficiently identify security issues. Hence, the dependency on highly skilled code reviewers can be reduced. One of the most important benefits of static code analysis is its ability to uncover potential security vulnerabilities early in the design lifecycle. Therefore, it spreads security awareness among RTL designers and encourages adopting secure RTL coding practices. This approach enhances the SoC's overall security posture and substantially reduces the cost of fixing security issues later in the lifecycle. The complexities and expenses associated with late-stage modifications can be avoided by addressing potential security vulnerabilities early. Thus, static code analysis is crucial to the overall SoC security verification flow.

Speed and Accuracy: Static code analysis-based methods for SoC security verification are highly efficient in identifying potential vulnerabilities in a particular design. Such solutions can rapidly analyze the RTL codebase, typically providing results within a few minutes. It accelerates the security verification process significantly. The speed of these methods ensures that potential vulnerabilities can be detected

and addressed promptly. Additionally, the accuracy of these techniques in detecting security vulnerabilities is relatively high, often approaching 100%. Such high accuracy ensures that potential security threats are reliably identified, thereby enhancing the overall security of the SoC design. Static code analysis-based methodology incorporates fast analysis with high accuracy. Therefore, it is a lucrative technique in the secure design lifecycle to perform security verification effectively, which helps safeguard against potential exploits and security breaches.

4) Limitations: The static code analysis-based technique for SoC security verification has numerous drawbacks, which are discussed below.

False Positives: While static code analysis-based techniques are highly effective in identifying potential security vulnerabilities in SoC designs, they are also susceptible to generating false positives. These occur when the solution mistakenly interprets an obscure or complex RTL coding construct as a potential security threat despite being benign. False positives are an inherent limitation of static analysis techniques, arising from the method's high reliance on predefined rules and patterns to identify potential security vulnerabilities without the contextual understanding that a human code reviewer might possess. The presence of false positives requires additional scrutiny and examination since RTL designers must manually review and validate these flagged security issues to recognize the actual security issues from the generated verification report. Therefore, while static code analysis significantly enhances the efficiency and thoroughness of security verification, it also imposes an additional burden on RTL designers to filter out these false flags raised so that the goal remains to address actual security vulnerabilities.

False Negatives: False negatives represent a notable challenge to be overcome by static analysis-based techniques for SoC security verification. These occur when the method fails to detect an existing security issue within the RTL code. Such a scenario allows potential vulnerabilities to go unnoticed. This failure can originate from inherent limitations in the technique's ability to comprehensively analyze the RTL codes, particularly when faced with highly complex or non-standard coding practices that fall outside the solution's predefined detection patterns. The occurrence of false negatives is concerning since it might create a false sense of security assurance, potentially leaving critical security vulnerabilities present in the SoC design unaddressed. Therefore, while static analysis-based techniques are lucrative for their efficiency and breadth of coverage, it is essential to complement those with other verification methods to ensure a comprehensive security assessment.

Blind Spots: Static analysis-based methods for detecting security issues in the RTL design of an SoC rely heavily on predefined rules, patterns, and coding styles. While this approach effectively identifies many common vulnerabilities, it inherently creates blind spots. These blind spots occur since the techniques are limited to the scope of the rules and patterns they are specifically programmed to recognize. Hence, any security vulnerabilities that do not conform to these predefined criteria may remain undetected. This limitation means that more sophisticated security threats, which do not manifest through recognizable coding styles or patterns, can be easily overlooked by static analysis-based solutions. The dependence on a finite set of identification criteria can thus result in incomplete security verification since the methods may fail to identify vulnerabilities that fall outside their programmed scope. These limitations emphasize the requirement for a hybrid approach to perform security verification, combining static analysis with other techniques, such as formal and dynamic methods, to ensure a more comprehensive detection of potential vulnerabilities.

IV. Property-driven Formal Methods for Security Verification

A. Overview

Property-driven formal methods for SoC security verification have gained much interest in the research community due to their potential to assess and validate complex designs' security aspects rigorously. These methods are based on developing well-defined security properties to address specific threat vectors relevant to the entire system. In security verification and validation, a security property serves as a formal statement that can check the design's assumptions, conditions, and expected behaviors, mainly related to potential security vulnerabilities [31, 42]. By defining these properties, verification engineers can verify that the design adheres to desired security standards and operates as intended under various scenarios. The formal nature of these properties allows for an exhaustive exploration of the design's state space to ensure compliance with security expectations. Moreover, the coverage of the security property can be utilized as a metric for evaluation. It can provide a quantitative measure of how thoroughly the design has been assessed against known security vulnerabilities [1, 2, 43].

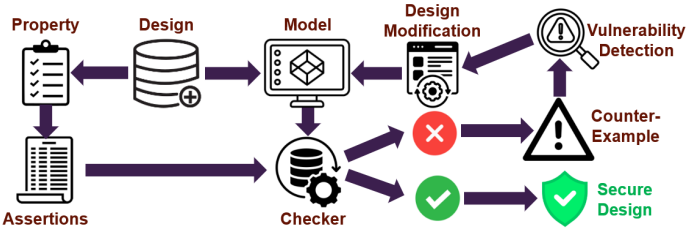

Fig. 12: Overview of property-driven formal method for SoC security verification.

As illustrated in Fig. 12, the security verification process through property-driven formal methods involves extracting a mathematical (formal) model from the design. Next, it is evaluated against a predefined set of security properties. These properties are typically expressed as assertions or cover statements. Such properties represent the expected secure behavior of the design. Once defined, these assertions are verified using a model checker tool. If a property is violated, the model checker provides counterexamples and scenarios demonstrating the property's failure within the design. These counterexamples are essential for the designers since they highlight the presence of potential security vulnerabilities in the design. By identifying and understanding these issues,

designers can iteratively modify the design to address the reported security issues [1, 2, 44].

B. Methodology

To identify a set of security properties for an SoC, a number of steps must be followed. These steps are illustrated in Fig. 13. Moreover, such steps are discussed in the following subsections.

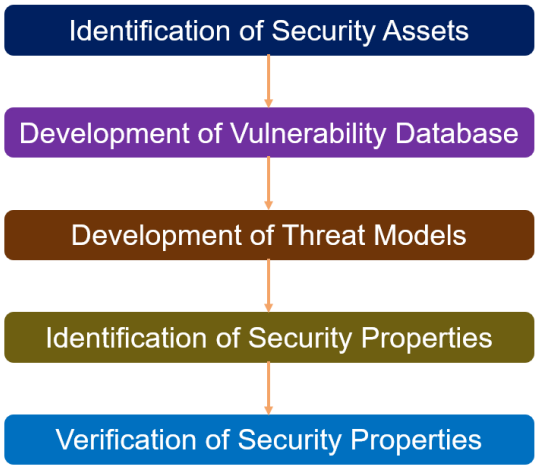

Fig. 13: Overview of property-driven formal method for SoC security verification.

1) Identification of Security Assets: To effectively make an SoC design secure against various security threats, a designer must understand the critical assets present in the system and their associated security levels. The concept of assets in an SoC encompasses both the physical and logical components that require protection, and these assets can vary in their importance depending on the abstraction level being considered. Different levels of abstraction—ranging from high-level architecture to low-level implementation details—may require considerations for asset protection. Moreover, the security requirements for these assets must account for the capabilities and intentions of potential adversaries, who may exploit different vulnerabilities present in the system. Hence, identifying and categorizing these assets based on their significance and the threats they face is crucial in developing security-critical properties for the system [1, 2, 45].

To identify assets, designers must carefully select primary and secondary assets. Primary assets directly influence the system's security, such as the cryptographic keys and plaintext in an AES module. While not directly impacting security, secondary assets can provide adversaries with opportunities to infer sensitive information if left unprotected [45]. For instance, intermediate results during AES computation can leak information about secret keys [42]. When an AES unit is integrated into an SoC, the scope of assets expands to include signals transferring primary assets like keys and plaintexts between modules, classifying them as dynamic or secondary assets. To mitigate potential vulnerabilities, designers must analyze the propagation paths of these assets in the SoC, identifying weaknesses and access points that adversaries could exploit.

2) Development of Vulnerability Database: Unintentional vulnerabilities can emerge at the front-end and back-end stages of SoC design, alongside the risks posed by intentional malicious modifications and exploitation. Once assets have been identified, the next critical step is identifying vulnerability points attackers might exploit to access these assets. It involves comprehensively scrutinizing the design to pinpoint weaknesses that adversaries might leverage to compromise the system. Such vulnerabilities can manifest through poor design practices, the use of Computer-Aided Design (CAD) tools that do not prioritize security [12–14], and Design for Test (DFT) and Design for Debug (DFD) infrastructures that inherently increase the controllability and observability of the design [16, 17], potentially exposing sensitive information to attackers.

Vulnerabilities in SoCs can be broadly classified into three categories. The first category includes vulnerabilities that stem from poor design practices. Existing design methodologies often do not incorporate security requirements, and security-aware design methodologies are not yet established. For example, design practices where the number of execution cycles depends on asset values can inadvertently create security loopholes. These factors contribute to design errors, which may result in security vulnerabilities. The second category comprises vulnerabilities introduced by CAD tools. These tools often perform optimizations without accounting for the varying security levels of different modules, inadvertently introducing new vulnerabilities. The third category includes vulnerabilities introduced by DFT and DFD infrastructures. While these infrastructures are crucial for improving design controllability and observability, they can also increase the risk of exploitation by providing attackers with the means to control or monitor the internal states of the SoC [1].

By surveying existing literature and examining the design's implementation, one can identify potential points of entry or vulnerabilities that an attacker might exploit. For instance, in the case of an AES implementation, attackers might use power and timing side-channel attacks [26–28], trace buffer [46] and scan attacks [47], hardware Trojan insertions [9–11], and physical attacks [48] to extract the encryption key. These strategies exploit weaknesses in the design to gain unauthorized access to sensitive information. By leveraging existing knowledge about these attacks and analyzing design resources, it becomes possible to pinpoint critical vulnerabilities in a poorly designed AES implementation. Examples of such weaknesses include the accessibility of the debug unit to intermediate AES results and the potential for attackers to access the encryption key via plaintext or control signals [42]. Identifying these vulnerabilities is crucial for developing secure design practices and implementing effective countermeasures to protect against these security threats.

3) Development of Threat Models: With a prior understanding of the design implementation, the assets requiring protection identified vulnerabilities, and the associated abstraction levels, threat models can be systematically developed for each asset. These threat models provide a framework for evaluating potential security threats by outlining how an adversary might exploit identified vulnerabilities. Some examples of potential

threats include deviations from expected functionality, which can lead to unintended behaviors, illegal access paths that may allow unauthorized access to sensitive assets, and the corruption of critical data and finite state machine (FSM) states, which can disrupt system operations and compromise data integrity [1, 12–14]. By classifying threat models, designers can prioritize risks and develop countermeasures to mitigate these threats. The threat models can be classified as follows.

Information Leakage: The unauthorized flow of sensitive assets to untrusted IP modules or observable points is a confidentiality violation. However, illegal modification or corruption of these assets is considered an integrity violation. Both violations are regarded in the information leakage threat model since these involve unauthorized access to sensitive data. This model emphasizes the need to protect against scenarios where design assets could be inadvertently or maliciously exposed through observable points, such as primary outputs or DFT and DFD infrastructures [16, 17, 42]. An example of a security property addressing this concern would be the assertion that design assets must not propagate to or be exposed through any observable points in the SoC. This property ensures that sensitive information remains confined within secure boundaries, preventing unintended data leakage that adversaries might exploit.

Denial of Service: Disruption of service or connectivity within SoC modules is a denial-of-service (DoS) threat, making resources unavailable or degrade their performance. This disruption can be caused by numerical exceptions, such as divide-by-zero errors, or by deadlocks that lock up system resources, making them inaccessible. These disruptions not only interrupt normal operations but can also introduce significant latency during recovery processes. Attackers may exploit this latency to leak information or violate access controls by analyzing the timing delays or manipulating system states [49]. A security property can be formulated to ensure that access to security-sensitive entities and memory is strictly restricted during any deadlock or recovery processes that result from a DoS event. This property would prevent unauthorized access to sensitive data and system components, safeguarding against potential exploitation during vulnerable periods of system recovery [1, 2].

Access Control/Isolation Violation: In SoC security, unauthorized interactions between trusted and untrusted IPs, illegal accesses to protected memory addresses, the protected states of controller modules, and out-of-bound memory accesses possess significant security threats. These interactions can lead to unwanted data leakage or manipulation, compromising the integrity and confidentiality of critical system components [50]. Security properties must be established to ensure proper isolation and protection of sensitive entities to mitigate such threats. One such property is that security-critical entities must be isolated from those with lower security levels. This isolation is crucial in maintaining a clear separation between different security domains, ensuring that vulnerabilities in lower-security areas do not compromise high-security assets. Additionally, another security property states that entities with equivalent security requirements should not impact each other's integrity and confidentiality. This property ensures that any interaction between similarly classified entities does not inadvertently lead to data leakage or corruption.

Side Channel Leakage: Side-channel leakage threats arise from vulnerabilities that allow attackers to extract sensitive information by analyzing indirect indicators such as timing variations, power consumption, or electromagnetic emanations [26–28]. These threats exploit the correlation between the physical behavior of a system and its internal data, enabling attackers to deduce confidential information without directly accessing it. Security properties must be defined to ensure that the control flow of a program is independent of asset values to counteract such threats. This independence prevents attackers from deducing asset values by observing the power and timing variations associated with different execution paths. For instance, if execution cycles or power consumption fluctuate based on asset values, attackers can use divide-and-conquer techniques to reveal the asset incrementally. Additionally, attackers may employ fault injection methods, such as power or clock glitching, temperature manipulation, or laser injection, to bypass security mechanisms and compromise the confidentiality or integrity of the design [13, 14, 18]. Security properties can be developed to guarantee that a single bit-flip or stuck-at-fault does not grant unauthorized access to protected states or memory addresses.

Design Tampering: Design tampering refers to any intentional or unintentional system modification that introduces vulnerabilities, potentially leading to information leakage, access control violations, or denial-of-service attacks. A notable example of design tampering is the insertion of hardware Trojans, malicious modifications embedded into unspecified functionalities of a design. These Trojans can create covert channels for data leakage, facilitate side-channel attacks, or enable unauthorized access to critical design assets. They pose a significant threat as they can remain undetected during standard verification processes [10, 11]. Specific security properties must be established to mitigate the risks associated with design tampering. For example, a security property can ensure that no state in the design allows sharing a cryptographic module's key with any non-cryptographic module. This property ensures proper isolation between secure and non-secure components, preventing unauthorized access to sensitive cryptographic keys.

Multiple threat models can be developed for a single asset to address different security concerns. For instance, unauthorized access to the intermediate results of encryption performed by an AES module via a debug port can lead to information leakage and access control violations. Security properties should be developed to ensure that each asset is thoroughly checked against potential vulnerabilities and threats. After identifying the assets, vulnerabilities, associated abstraction levels, and threat models, the next step involves selecting the relevant IPs and SoC transactions that either contain these vulnerabilities or play a role in propagating the corresponding assets. For example, when focusing on information leakage and side-channel attacks, the security properties should prioritize crypto IPs, True Random Number Generators (TRNGs), and asset management units to protect sensitive information. On the contrary, if the concern is denial of service or access control

violations, the emphasis should be on halt/debug units and exception handler units, as these are critical areas where such threats can manifest [1, 2].

4) Identification of Security Properties: The security level of a design is inherently influenced by its assets, design attributes, associated vulnerabilities, and potential threats. Engineers can guide their security verification efforts to address the most relevant concerns by understanding the specific characteristics and possible threats associated with a design. For instance, if a design is physically isolated, security issues such as power side-channel leakage might be mitigated due to the lack of physical access required to exploit such vulnerabilities. Therefore, a security verification engineer can focus on developing security properties that target other vulnerabilities that are more relevant to the operation of the design.

The information gathered from prior steps allows for defining security metrics and identifying corresponding security properties. Two critical security metric definitions can be employed for security evaluation: signal observability and the assessment of confidentiality and integrity. In RTL design, the difficulty of observing signals or accessing specific statements indicates the likelihood of hardware Trojans being present within the design. Security properties should be developed to cover all rare statements, as these are potential hotspots for embedding malicious functionalities. Addressing these rare conditions ensures that unintended or malicious behavior is detected and mitigated. It is essential to evaluate the accessibility of sensitive assets through observable points such as DFT or DFD infrastructures at the gate level to assess confidentiality and integrity. Designers can identify potential security vulnerabilities by detecting how easily these assets can be accessed through such points [1].

A database can be developed to formulate the security properties, mapping threat models to particular properties and vice versa. For instance, for the MSP430 microcontroller, the value of critical CPU registers, intermediate buffers, and the content of data or program memory can be regarded as primary assets, as appeared from the design specification. During debug mode, a debugger's ability to initiate a HALT operation for off-chip debugging introduces a potential entry point for attackers. This debug feature, if exploited, can lead to vulnerabilities when primary assets propagate through the design. Potential security vulnerabilities include incorrect design implementation, intermediate states introduced by CAD tools, accessibility to intermediate results, and the activity of functional modules during HALT operations. These vulnerabilities might allow debuggers unauthorized access to data memory space, the program counter, or redirect code execution during HALT operations, leading to threats such as information leakage or control flow violations. Additionally, write access to program memory might enable firmware modification and code injection attacks, while intermediate state accessibility during HALT operations presents opportunities for physical attacks, such as probing. Analyzing the identified assets, vulnerabilities, and potential threats, designers can develop a set of security-critical properties for the MSP430. These properties should be defined as the negation of the vulnerabilities enlisted in the database.

5) Verification of Security Properties: It is essential to consider the time the property should be verified and to formally model or instrument the design accordingly. Some properties must be checked statically at the design time. For example, ensuring that registers containing the critical status of a CPU are accessed only through valid means is crucial, as any undefined access poses a security threat. Such static verification at design time helps identify potential security flaws early in development. On the other hand, at boot time or reset time, intermediate buffers may be initialized with 'don't-care' values, which might inadvertently create security vulnerabilities. It is necessary to develop security properties that guarantee these 'don't-cares' do not lead to security breaches, such as unauthorized access to critical states, which may enable attackers to extract sensitive information or initiate denial-of-service attacks.

Furthermore, some assets within the design may be generated dynamically at run-time, requiring deploying security-critical properties in the form of sensors or monitors that actively operate during execution. These properties are crucial for preventing malicious functionalities and mitigating denial-of-service scenarios by continuously monitoring system states and behaviors. Therefore, the security properties can be classified into three categories: static or design-time rules, boot-time rules, and run-time rules [1, 31]. Static rules focus on verifying security constraints during the design phase, boot-time rules address potential vulnerabilities during the initialization phase, and run-time rules provide ongoing protection against threats encountered during execution.

It is crucial to employ appropriate tools and techniques that align with the nature of the security properties and the threat models they address. The choice of tools often dictates whether formal modeling or design instrumentation is necessary. For instance, model checker tools are well-suited for formally verifying static or boot-time properties. These tools require translating the design into formal models [51] and expressing the properties as Linear Temporal Logic (LTL) or Computation Tree Logic (CTL) formulas [52], which the model checker can then evaluate to ensure compliance with specified security constraints. Simulation-based assertion checking also plays a vital role in verifying properties by simulating the design and checking assertions dynamically to identify any deviations from expected behavior.

Design instrumentation techniques such as information flow tracking and taint analysis are suitable for run-time properties [35, 36], particularly those related to information flow and confidentiality. These methods help to detect potential information leakage by monitoring how secret values propagate through the design and ensuring they do not reach observable points. Additionally, equivalence-checking tools can be utilized to verify the integrity of the design by comparing different versions and ensuring that malicious modifications or safe design transformations do not introduce vulnerabilities. The outcome of security validation through property checking is typically a pass or fail result, clearly indicating whether the design meets its security requirements. This result can be used in the security assessment process, allowing designers to identify weaknesses and take appropriate actions to enhance

the system security.

C. Advantages

Property-driven formal methods for SoC security verification are associated with numerous advantages. Such advantages are presented below.

Exhaustiveness: Property-driven formal methods are rigorous techniques to verify the security requirements for an SoC design by specifying properties that the design should satisfy. Such methods exhaustively explore all possible states and behaviors of the design to ensure these properties hold. Unlike traditional testbench-based simulation, which can only evaluate a few scenarios, formal methods employ mathematical models to analyze every potential state and transition within a system systematically. This exhaustive nature allows for identifying corner cases and often overlooked situations that may not be easily captured through conventional verification methodologies. By uncovering these scenarios, formal methods play an essential role in detecting complex security vulnerabilities that might remain hidden until exploited. Therefore, these methods provide a promising framework for enhancing SoC security by ensuring that all specified security properties are verified by covering corner cases.

Accuracy: Property-based formal methods can verify security requirements for an SoC design by ensuring a rigorous mathematical proof of correctness. Such techniques validate that a design adheres to the defined security properties. It results in very high accuracy in identifying potential security vulnerabilities present in the RTL design of an SoC. The most critical aspect of formal verification is its ability to achieve 100% verification accuracy. This implies that if the security properties and constraints are correctly defined and the verification process is successful, the design is guaranteed to be free from potential security issues according to the specified properties. This level of certainty is not achievable with testbench-based simulation techniques, which rely on manually constructed test scenarios to explore the potential problems. There is no mathematical guarantee that all potential security vulnerabilities have been detected. Formal methods eliminate corner cases or missed scenarios, as they can exhaustively explore the system's possible states and behaviors.

Automation: Property-driven formal methods completely automate the security verification process using specific properties. This approach begins with an initial setup phase, where security properties and constraints are formulated based on the system's security requirements. Once these properties are defined, formal methods employ automated tools to exhaustively explore the state space, verifying whether the specified properties hold under all possible conditions. This automated process significantly reduces the verification time and effort required compared to traditional manual verification techniques. In this manner, formal methods thoroughly analyze the system's behavior, proving that security properties are satisfied. Hence, it minimizes the risk of overlooking potential security vulnerabilities.

D. Limitations

Formal techniques are also associated with several shortcomings. These drawbacks are illustrated as follows.

Design Dependency: One of the major limitations of property-based formal methods in SoC security verification is that security properties may not be design and architecture-agnostic. Therefore, it can lead to portability issues across different designs. Security properties are often highly related to a given SoC's specific architecture and design features, reflecting the system's unique security requirements and threat models. Hence, a property formulated for one design may not apply to another, even though the two designs have similar functionalities. This lack of portability requires reformulating and adapting security properties for each new design. Such a scenario may increase the overall time and effort necessary for security verification. Moreover, the need for custom properties for each architecture may introduce variability in the quality and comprehensiveness of the security verification process. The underlying reason is that security property definition and application inconsistencies might arise.

Subjectivity: A notable limitation of property-based formal methods is the reliance on the expertise and experience of the verification engineer in formulating effective security properties. The success of these methods depends on the proper specification of properties that capture the design's security requirements and potential threat vectors. This task requires a good understanding of the architecture of the SoC and the possible security threats it may face. Engineers must have expertise in security to anticipate potential threat vectors and translate them into properties that can be verified mathematically using formal verification tools. Critical security aspects may be overlooked without such knowledge, leading to incomplete security verification and residual vulnerabilities. Therefore, the effectiveness of formal methods in security verification is highly dependent on the engineer's insight. Such reliance may make the SoC security verification process subjective.

Scalability: A significant limitation of property-driven formal methods is the challenge of the state-space explosion, particularly for very large SoC-level designs. As the complexity and size of a design increase, the number of possible states that need to be analyzed grows exponentially. For instance, a design with 2048 flip-flops has a substantially ample state space, but adding just one more flip-flop to make it 2049 causes an exponential increase in the state space. This exponential growth can quickly surpass the computational resources available. Therefore, it becomes infeasible for the verification tools to handle the design size or the analysis's complexity. As the state space becomes overwhelmingly large, the time required to explore and verify all possible states can become prohibitively lengthy and lead to a point of diminishing returns. This problem, known as state-space explosion, poses a notable challenge for formal verification techniques in handling large and complex SoC designs despite their accuracy and exhaustiveness.

V. Dynamic Techniques for Security Verification

This section will present an overview of dynamic techniques for SoC security verification. First, we will provide a high-level overview and a simple taxonomy. Next, we will briefly discuss such methods with their associated advantages and limitations.

A. Introduction

Dynamic verification is a testing methodology that evaluates hardware while actively executing a workload. This approach allows for observing the hardware's behavior under realistic operational conditions. The workload can be introduced directly to the hardware through stimuli such as signals applied to external pins, or indirectly through the execution of software or firmware. The environment for dynamic verification can be either virtual, utilizing RTL simulation to model the hardware's operation, or physical, employing FPGA prototyping or the actual hardware. This approach contrasts with static verification, which involves analyzing a theoretical model or representation of the hardware's design without executing workloads. Dynamic verification's primary advantage is its ability to capture runtime behaviors and interactions that static methods might overlook. Therefore, it can provide insights into the hardware's performance, stability, and security in real-world scenarios.

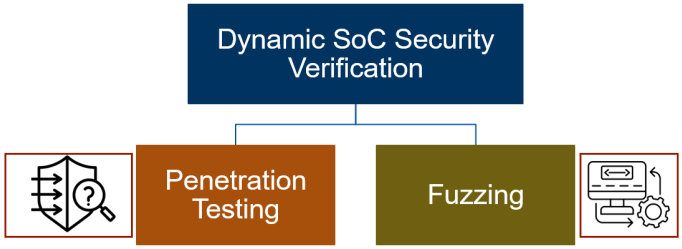

Fig. 14: Dynamic methods for SoC security verification.

Dynamic verification is often designed to support hardware-software co-verification. It is a notable aspect since it ensures the combined verification of software and hardware components. It is typically achieved through hardware-software simulation environments that allow concurrent testing of both layers. Hence, it can highlight potential security issues arising from their interaction. In dynamic verification, two prominent techniques are penetration testing and fuzz testing (fuzzing), as shown in Fig. 14. Fuzz testing involves supplying the hardware with randomly generated or mutated inputs to uncover unexpected behavior, security vulnerabilities, or crashes. This technique is quite effective in identifying corner cases and potential security flaws. Penetration testing, on the contrary, is focused on identifying and propagating the effects of vulnerabilities within an SoC design to observable points rather than examining the cross-modular and cross-layer effects typical in complex systems.

B. Penetration Testing

1) Overview: In the SoC security verification domain, penetration testing is novel compared to its well-established counterpart in software testing. However, the term has often been misapplied in the context of hardware security, serving as a synonym for vulnerability assessment or post-silicon testing and debugging [53]. The provided examples focus on assessing the software layer operating on the hardware rather than conducting a comprehensive security evaluation of the hardware itself. Therefore, penetration testing in the state-of-the-art hardware security literature lacks depth and rigor as opposed to its software counterpart.

Pre-silicon hardware penetration testing is a methodology focused on identifying and propagating the effects of vulnerabilities within an SoC design to observable points rather than examining the cross-modular and cross-layer effects typical in complex systems. Unlike randomized testing, which generates test patterns without specific knowledge of the vulnerabilities it aims to detect, hardware penetration testing operates under the assumption of gray or black box knowledge regarding the design specifications and a gray box understanding of the targeted vulnerabilities. This gray box approach means that testers know the type and potential impact of a bug or vulnerability on the system but lack precise details about its exact origin or manifestation point within the intricate design. Thus, penetration in this context refers to propagating a vulnerability from its initial and unobservable point in the design to a point where it becomes detectable and observable.

2) Methodology: The first framework for hardware penetration testing was introduced in [54]. It is illustrated in Fig. 15. This framework presents a step-by-step methodology for executing hardware penetration testing. Its core emphasis remains on the concept of a cost function. The cost function is a mathematical representation of security policies, which can be potentially violated by exploiting security vulnerabilities. The framework can efficiently guide the penetration testing process by leveraging this representation. The utilization of the Binary Particle Swarm Optimization (BPSO) algorithm lies at the heart of this framework. It generates test patterns aimed at uncovering security weaknesses. These patterns are continuously refined through iterative feedback mechanisms, where hardware signals from the Device Under Test (DUT) are monitored and analyzed in conjunction with the cost function. This iterative refinement process guarantees that the test patterns evolve to target areas where security policies might be violated effectively.

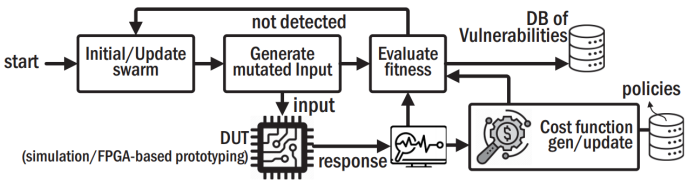

Fig. 15: A framework for hardware penetration testing [54].

The framework's flexibility is enhanced by its ability to apply generated test patterns to both software models of the hardware device under test and FPGA prototypes of the DUT. This dual compatibility allows for testing through both simulation and emulation methodologies. The BPSO algorithm's suitability for binary input patterns enables the framework to encompass a broad range of test scenarios

by encoding various elements into binary bit patterns. For instance, assembly programs can be mutated in remote or software access cases to test the design's resilience against security threats. A significant advantage of this framework is that it does not presuppose the existence of any particular kind of security flaw. Instead, it focuses on detecting violations of security policies, which inherently cover a broad spectrum of vulnerabilities. It makes the framework versatile, adaptable, and applicable to various threat models.

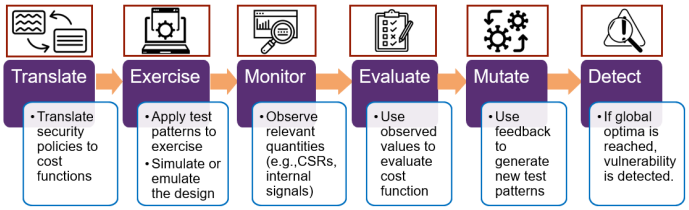

Fig. 16: The steps of hardware penetration testing.

The penetration testing framework encompasses six distinct steps. As illustrated in Fig. 16, the process initiates with translating the device's security policies into mathematical cost functions. These functions are quantifiable representations of security objectives, providing a mathematical metric against which potential security vulnerabilities might be assessed. Once the cost functions are established, the framework advances to the second step, where the design is exercised using test patterns. This step can be conducted through hardware-software co-simulation, applying the test patterns via the software layer, or directly inputting hardware stimuli into the simulation environment. The third step involves monitoring, where testers observe relevant signals identified by the second prerequisite to collect data on how the system responds to these test patterns. This data is then used in the fourth step to evaluate the previously translated cost functions, providing feedback on the system's security posture. The subsequent step, mutation, leverages this feedback to modify the test patterns, optimizing the testing approach to delve deeper into the design's potential weaknesses. Finally, the detection phase is achieved when the mutation process identifies a global optimum, indicating the presence of a security vulnerability in the SoC design. More details can be found in [54].

3) Advantages: The advantages of penetration testing for SoC security verification is presented below.

Software Exploit Generation: Penetration testing can lead to the direct generation of software code snippets that may exploit identified vulnerabilities, potentially compromising system security. This capability arises from the framework's ability to translate security policies into quantifiable cost functions, guiding the generation of test patterns through algorithms like Binary Particle Swarm Optimization (BPSO). By employing these test patterns, the framework can simulate real-world attack scenarios, probing the hardware design to identify weak points where security policies might be violated. The framework can produce corresponding software code snippets targeting these specific weaknesses when vulnerabilities are detected. Such snippets can simulate malicious code execution paths or unauthorized access attempts to exploit the identified vulnerabilities. The ability to generate these code snippets provides designers with valuable insights into potential attack vectors. It assists in addressing these security threats before they can be exploited in a real-world scenario.

Hardware-software Stack Consideration: In penetration testing, security verification of an SoC focuses on the entire hardware-software stack. This holistic approach ensures that vulnerabilities are identified within the hardware components and their interactions with the software layers. The framework initiates with translating security policies into mathematical cost functions encompassing hardware and software security objectives. The framework evaluates the SoC's response at both the hardware and software levels by employing test patterns that can simulate various attack scenarios. This dual-layer examination is crucial as vulnerabilities can often arise from the complex interplay between software applications and hardware configurations, which might not be apparent when examining each component in isolation. During the testing process, co-simulation methods allow the application of test patterns via the software layer while simultaneously monitoring hardware signals, thereby capturing the complete picture of potential security vulnerabilities. This approach enables the identification of vulnerabilities that span the hardware-software interface, allowing for a more effective detection and mitigation of security threats.

Speed: Emulation-based penetration testing offers a substantial advantage in speed compared to simulation when evaluating security at the system level. It often achieves performance improvements of up to 20 times. This speed gain is primarily due to the inherent differences between emulation and simulation methodologies. While simulation-based techniques model and analyze each component and interaction in detail, emulation leverages hardware resources to replicate the SoC design, allowing for real-time execution of test patterns and scenarios. This real-time capability enables emulation to rapidly process large volumes of data and complex interactions. Therefore, emulation is particularly well-suited for penetration testing. The increased emulation speed facilitates more extensive testing within shorter time frames, enhancing the ability to identify and address potential security threats efficiently.

4) Limitations: Apart from its advantages, the penetration testing-based security verification method has some inherent limitations. These drawbacks are illustrated below.

Manual Construction of Cost Function: Constructing cost functions for penetration testing can be manual and complex, particularly in frameworks like the one presented in [54]. Cost functions are crucial in penetration testing, as they provide a quantifiable measure of security objectives and potential policy violations. Crafting such functions manually requires an in-depth understanding of the SoC's architecture, security policies, and potential vulnerabilities. The complexity arises from the need to accurately translate abstract security requirements and potential threats into mathematical representations that can effectively guide the penetration testing process. The manual construction of these functions may require a thorough analysis of the SoC's hardware and software layers.

Subjectivity: In penetration testing, identifying security vulnerabilities within an SoC is intrinsically linked to the

designer's expertise and understanding of potential security threats. These are crucial in formulating an effective cost function. The designer's experience is instrumental in recognizing the possible ways security breaches might occur within an SoC's intricate interplay of hardware and software layers. However, crafting a suitable cost function requires effort. It requires understanding the SoC's architectural complexities and the various attack vectors that adversaries might exploit. This complexity introduces a degree of subjectivity into the process, as the identification of vulnerabilities and the selection of metrics often depend on the designer's understanding and interpretation of potential threats. Therefore, variability in expertise and perspective can lead to differences in how cost functions are constructed, potentially impacting the effectiveness and consistency of the vulnerability assessment.

C. Fuzzing

1) Overview: Fuzz testing, or fuzzing, is a randomized security verification methodology that can uncover program vulnerabilities through an automatic or semi-automatic process. This method is comprised of five stages. These stages are illustarted in Fig. 17.

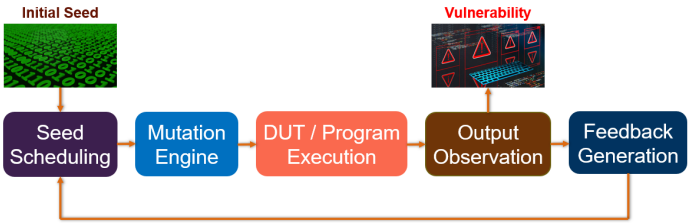

Fig. 17: Stages of fuzz testing.

The initial stage is known as seed scheduling. It selects some efficient seeds that facilitate rapid coverage of potential SoC vulnerabilities. Following this, the mutation stage generates new randomized or directed inputs to trigger security vulnerabilities by traversing corner cases. In the subsequent test execution phase, the program is run using these mutated inputs to observe its output behavior. The fourth stage involves monitoring the run-time behavior of output signals and ports by probing the simulation or emulation platform to detect any vulnerabilities. Finally, the feedback generation phase evaluates the run-time performance and guides the mutation techniques to create more effective input patterns.

Among the prominent fuzzing tools, American Fuzzy Lop (AFL) [55] is notable for utilizing an approximate branch coverage metric, which helps to detect code areas executed during fuzzing. On the contrary, Honggfuzz [56] tracks the unique basic code blocks visited. It provides a different perspective on code coverage. Fuzzing techniques overcome the scalability issues inherent in formal verification methods. These techniques are classified into three types: white-box, gray-box, and black-box, depending on the amount of information available during verification, such as source code, security specifications, functional requirements, control/data flow graphs, and execution metrics from simulations or emulations. Various fuzzers utilize different benchmarks. Hence, it leads to diverse fuzzing mechanisms categorized by baseline performance, crash types, coverage modes, and seed formulation techniques. Detailed comparisons and evaluations of these different fuzzing approaches can be found in [57, 58].

2) Methodology: An effective approach to creating a coherent model of the SoC that enables the use of state-of-the-art software-based fuzzers for hardware verification is to translate the SoC's hardware model into its equivalent software model. This emerging concept leverages existing fuzzing tools without necessitating the development of a new platform for RTL verification. The process involves using a hardware translator, such as Verilator [59], to convert RTL designs into corresponding C++ programs. These generated C++ classes are then instantiated by a wrapper, which serves as the main function and simulates the program under test. This wrapper is analogous to a test bench in hardware design. It is designed to incorporate functionalities and security properties that act as a cost function, which provides feedback for subsequent iterations. Additionally, the generated C++ programs can be instrumented to enhance the visibility of internal simulations. Such instrumentation can help to achieve the target code coverage, line coverage, branch coverage, and other similar metrics. Therefore, it enables a comprehensive evaluation of the overall coverage achieved during simulation.

A metric known as the cost function is introduced to quantitatively assess the extent to which the security of an SoC design has been compromised and to measure the proximity of test scenarios to activating a targeted security vulnerability. This metric plays a vital role in guiding the fuzzing process. It builds an evolutionary mechanism that effectively guides the fuzzing efforts toward uncovering vulnerabilities in a significantly shorter time than blind fuzzing techniques. The cost function can incorporate various general metrics, such as code coverage, line coverage, and branch coverage, which are obtained through the instrumentation of RTL designs or their translated software models. By working in conjunction with the mutation engine of the fuzzer, the cost function helps to generate more effective mutated test cases, therefore enabling the exploration of various corner cases. This relationship between the cost function and the mutation engine enhances the effectiveness of the fuzzing process.

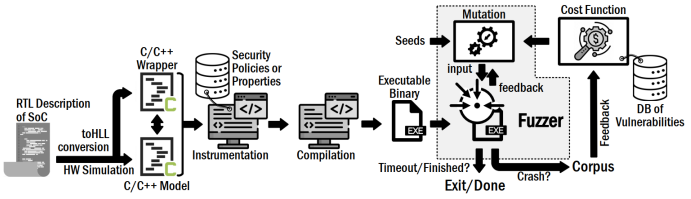

Fig. 18: A fuzz testing framework on software model of an SoC's RTL description [60].

Fig. 18 illustrates transforming RTL code into an executable binary [60]. It also encompasses the instrumentation process. This figure also depicts the interaction between the cost function, the fuzzer outputs, and the mutation engine. A notable characteristic of this framework is the adaptability of the cost function, which is selected based on the specific vulnerability instance currently being targeted. This approach allows for more precise and effective analysis, as the feedback from the fuzzer outputs can be customized to address the

bug under investigation. By focusing the cost function on the targeted security vulnerability, the framework enhances the decision-making process for subsequent analyses significantly. It implies that the fuzzing efforts aim to identify and mitigate specific security issues within the SoC design.

3) Advantages: There are several advantages associated with fuzzing-based security verification techniques. Those advantages are discussed as follows:

Coverage Metrics: Fuzzing is associated with various coverage metrics essential for evaluating an SoC's security status. These metrics are important in guiding the fuzzing process by identifying unexplored areas in the design and assessing the extent to which the SoC has been tested for potential vulnerabilities. Key coverage metrics include code coverage, line coverage, and branch coverage, which determine how much of the RTL codebase has been exercised by the generated test cases. Additionally, hardware-specific metrics such as FSM coverage and toggle rate provide more insights. These metrics altogether increase the likelihood of uncovering security issues. By employing these metrics, fuzzing not only aids in detecting security vulnerabilities but also provides a quantitative measure of the SoC's security posture.

Security-oriented Cost Function: Fuzzing facilitates the effective generation of test vectors to identify security vulnerabilities existing in an SoC design without requiring extensive security expertise from the designers. By employing automated techniques, fuzzing explores various scenarios, including corner cases that are difficult to anticipate through manual testing. The process involves using a mutation engine to systematically alter inputs based on predefined metrics, thereby maximizing the potential for uncovering hidden security vulnerabilities. This approach reduces designers' need for deep security knowledge since the fuzzing tool autonomously generates comprehensive test cases. Furthermore, integrating cost functions tailored to specific vulnerabilities generates test vectors for triggering security vulnerabilities.

Speed: Emulation-based fuzzing offers a substantial speed advantage over simulation when testing hardware at the system level. This efficiency arises from the ability of emulation to execute on physical hardware or field-programmable gate arrays (FPGAs). Unlike traditional simulation, which models the system's behavior in a software environment and often incurs significant computational overhead, emulation directly maps the design onto hardware. It results in faster execution and higher throughput. This speed advantage is helpful for fuzzing, where many test cases must be evaluated rapidly to uncover potential security vulnerabilities. Fuzzing techniques can explore a broader range of input scenarios and system states using emulation within a shorter time frame. Such a capability enhances the coverage and depth of security verification significantly.

4) Limitations: Several limitations exist for fuzz testing-based SoC security verification. Such limitations are illustrated as follows:

Input Dependence: There are several fundamental differences between fuzzing a software program and an RTL hardware model. The most notable one is the nature of the input arguments. In digital circuits, input is provided through input ports that can accept varying values with each clock cycle. It differs from software, which can receive inputs from various sources such as command-line arguments, files, or operating system calls. For fuzzing to be effective, especially for a translated hardware design, there must be a clear and well-defined format for the input data. This input definition is vital since it enables the fuzzer to perform proper mutation and generate valid test cases that the system under test can process. Therefore, when working with hardware models translated into software form, it is essential to properly define the input format for the fuzzer. The way inputs are formatted influences the performance of the mutation engine significantly. The reason is that it determines the variety and relevance of the test cases that can be generated. Properly defined inputs ensure that the fuzzing process can efficiently explore the input space and uncover potential vulnerabilities effectively.

Metric Translatability: Another critical issue when adapting fuzzing techniques from software to hardware models of an SoC is the difference in coverage metrics. Many fuzzers rely on instrumentation to gather coverage metrics, which are then used to guide the generation of seeds toward unexplored sections of the design. While some metrics, such as branch and line coverage, can be approximately mapped between hardware and software models, other metrics unique to hardware, like finite state machine coverage or toggle rate, are not directly translatable to software scenarios. Traditional hardware verification processes utilize these hardware-specific metrics to target distinct classes of vulnerabilities that might not be apparent through standard software-based metrics.

Challenges of Software Equivalent Model: Software equivalent models for hardware simulation introduce numerous challenges due to the fundamental differences in operational characteristics between hardware and software systems. Adapting existing software fuzzers to these models often proves inefficient, as hardware designs require simulation of every clock cycle, precise computation of register values, and detailed handling of complex bit manipulation operations. Such processes are computationally expensive. Additionally, these models must accurately represent hardware-specific components like controllers, system buses, and queues, further complicating the simulation process. Software fuzzers typically depend on program crashes and memory safety checks to detect bugs, but these concepts do not directly apply to hardware that does not experience crashes similarly. Instead, hardware verification focuses on checking designs against specifications to identify logic errors, timing violations, and unintended data or control flows. Furthermore, straightforward operations in a Hardware Description Language (HDL), such as bit manipulation, can become complex and resource-intensive when represented in software. Conversely, complex hardware structures like multiplexers may be simplified to basic conditional statements in software.

VI. Conclusion

In summary, this paper has examined various aspects of SoC security verification. It has underscored the complexities and challenges inherent in conducting a thorough security

assessment of SoC designs. Moreover, it has illustrated the limitations of current security verification methodologies, emphasizing the need for adaptable approaches to address the evolving landscape of security threats. Furthermore, this paper has classified state-of-the-art security verification techniques and presented a taxonomy. Additionally, it has provided a high-level overview of these techniques, illustrating their fundamental working principles from a bird's eye view. It includes an analysis of static and dynamic verification methods, highlighting how each approach addresses different aspects of security assurance for SoC designs. The discussion concludes with the advantages and disadvantages of these techniques. Hence, it provides deeper insights into their applicability and effectiveness in achieving the desired security objectives.

REFERENCES

[1] 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.

[2] N. F. Dipu, F. Farahmandi, and M. Tehranipoor, "SoC security properties and rules," Cryptology ePrint Archive, Paper 2021/1014, 2021. [Online]. Available: https://eprint.iacr.org/2021/1014

[3] R. Kibria, N. Farzana, F. Farahmandi, and M. Tehranipoor, "Fsmx: Finite state machine extraction from flattened netlist with application to security," in *2022 IEEE 40th VLSI Test Symposium (VTS)*. IEEE, 2022, pp. 1–7.

[4] R. Kibria, F. Farahmandi, and M. Tehranipoor, "Fsmx-ultra: Finite state machine extraction from gate-level netlist for security assessment," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 3613–3627, 2023.

[5] M. Tehranipoor and C. Wang, "Introduction to hardware security and trust," 2011.

[6] R. Kibria, M. Sazadur Rahman, F. Farahmandi, and M. Tehranipoor, "Rtl-fsmx: Fast and accurate finite state machine extraction at the rtl for security applications," in *2022 IEEE International Test Conference (ITC)*. IEEE, 2022, pp. 165–174.

[7] 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, Paper 2022/092, 2022. [Online]. Available: https://eprint.iacr.org/2022/092

[8] M. M. M. Rahman, M. S. Rahman, R. Kibria, M. Borza, B. Reddy, A. Cron, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Capec: A cellular automata guided fsm-based ip authentication scheme," in *2023 IEEE 41st VLSI Test Symposium (VTS)*. IEEE, 2023, pp. 1–8.

[9] 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.

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

[11] A. Jain, Z. Zhou, and U. Guin, "Survey of recent developments for hardware trojan detection," in *2021 ieee international symposium on circuits and systems (iscas)*. IEEE, 2021, pp. 1–5.

[12] R. Kibria, F. Farahmandi, and M. Tehranipoor, "Arc-fsm-g: Automatic security rule checking for finite state machine at the netlist abstraction," in *2023 IEEE International Test Conference (ITC)*. IEEE, 2023, pp. 320–329.

[13] 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.

[14] 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, 2019.

[15] P. Mishra, S. Bhunia, and M. Tehranipoor, *Hardware IP security and trust*. Springer, 2017.

[16] J. Da Rolt, A. Das, G. Di Natale, M.-L. Flottes, B. Rouzeyre, and I. Verbauwhede, "Test versus security: Past and present," *IEEE Transactions on Emerging topics in Computing*, vol. 2, no. 1, pp. 50–62, 2014.

[17] A. Basak, S. Bhunia, and S. Ray, "Exploiting design-for-debug for flexible soc security architecture," in *Proceedings of the 53rd Annual Design Automation Conference*, 2016, pp. 1–6.

[18] J.-M. Dutertre, V. Beroulle, P. Candelier, S. De Castro, L.-B. Faber, M.-L. Flottes, P. Gendrier, D. Hély, R. Leveugle, P. Maistri *et al.*, "Laser fault injection at the cmos 28 nm technology node: an analysis of the fault model," in *2018 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 2018, pp. 1–6.

[19] M. Ebrahimabadi, S. S. Mehjabin, R. Viera, S. Guilley, J.-L. Danger, J.-M. Dutertre, and N. Karimi, "Detecting laser fault injection attacks via time-to-digital converter sensors," in *2022 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2022, pp. 97–100.

[20] A. Das, G. Memik, J. Zambreno, and A. Choudhary, "Detecting/preventing information leakage on the memory bus due to malicious hardware," in *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, 2010, pp. 861–866.

[21] D. Tychalas, A. Keliris, and M. Maniatakos, "Stealthy information leakage through peripheral exploitation in modern embedded systems," *IEEE Transactions on Device and Materials Reliability*, vol. 20, no. 2, pp. 308–318, 2020.

[22] I. Giechaskiel and J. Szefer, "Information leakage from fpga routing and logic elements," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–9.

[23] T. Bonny and Q. Nasir, "Clock glitch fault injection attack on an fpga-based non-autonomous chaotic oscilla-

tor," *Nonlinear Dynamics*, vol. 96, pp. 2087–2101, 2019.
[24] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against intel sgx," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1466–1482.
[25] A. M. Shuvo, T. Zhang, F. Farahmandi, and M. Tehranipoor, "A comprehensive survey on non-invasive fault injection attacks," *Cryptology ePrint Archive*, 2023.
[26] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems," in *Advances in Cryptology—CRYPTO'96: 16th Annual International Cryptology Conference Santa Barbara, California, USA August 18–22, 1996 Proceedings 16*. Springer, 1996, pp. 104–113.
[27] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology—CRYPTO'99: 19th Annual International Cryptology Conference Santa Barbara, California, USA, August 15–19, 1999 Proceedings 19*. Springer, 1999, pp. 388–397.
[28] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, "Introduction to differential power analysis," *Journal of Cryptographic Engineering*, vol. 1, pp. 5–27, 2011.
[29] 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.
[30] D. M. Anderson, *Design for manufacturability: how to use concurrent engineering to rapidly develop low-cost, high-quality products for lean production*. Productivity Press, 2020.
[31] F. Farahmandi, Y. Huang, P. Mishra, F. Farahmandi, Y. Huang, and P. Mishra, "Soc security verification challenges," *System-on-Chip Security: Validation and Verification*, pp. 15–35, 2020.
[32] E. El Mandouh and A. G. Wassal, "Automatic generation of hardware design properties from simulation traces," in *2012 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2012, pp. 2317–2320.
[33] S. Lagraa, A. Termier, and F. Pétrot, "Data mining mpsoc simulation traces to identify concurrent memory access patterns," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 755–760.
[34] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1691–1696.
[35] W. Hu, A. Ardeshiricham, and R. Kastner, "Hardware information flow tracking," *ACM Computing Surveys (CSUR)*, vol. 54, no. 4, pp. 1–39, 2021.
[36] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner, "Gate-level information flow tracking for security lattices," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 1, pp. 1–25, 2014.
[37] K. Xiao, A. Nahiyan, and M. Tehranipoor, "Security rule checking in ic design," *Computer*, vol. 49, no. 8, pp. 54–61, 2016.
[38] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721.
[39] W. Webber, "Re-examining the effectiveness of manual review," in *Proc. SIGIR Information Retrieval for E-Discovery Workshop*, vol. 2, 2011.
[40] P. Louridas, "Static code analysis," *Ieee Software*, vol. 23, no. 4, pp. 58–61, 2006.
[41] I. Gomes, P. Morgado, T. Gomes, and R. Moreira, "An overview on the static code analysis approach in software development," *Faculdade de Engenharia da Universidade do Porto, Portugal*, vol. 16, 2009.
[42] W. Hu, A. Ardeshiricham, M. S. Gobulukoglu, X. Wang, and R. Kastner, "Property specific information flow analysis for hardware security verification," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2018, pp. 1–8.
[43] F. Farahmandi, M. S. Rahman, S. R. Rajendran, and M. Tehranipoor, "Metrics for soc security verification," in *CAD for Hardware Security*. Springer, 2023, pp. 37–79.
[44] F. Farahmandi, Y. Huang, and P. Mishra, "System-on-chip security," *Cham, Switzerland: Springer*, 2020.
[45] N. Farzana, A. Ayalasomayajula, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Saif: Automated asset identification for security verification at the register transfer level," in *2021 IEEE 39th VLSI Test Symposium (VTS)*. IEEE, 2021, pp. 1–7.
[46] Y. Huang and P. Mishra, "Trace buffer attack on the aes cipher," *Journal of Hardware and Systems Security*, vol. 1, pp. 68–84, 2017.
[47] B. Ege, A. Das, S. Gosh, and I. Verbauwhede, "Differential scan attack on aes with x-tolerant and x-masked test response compactor," in *2012 15th Euromicro Conference on Digital System Design*. IEEE, 2012, pp. 545–552.
[48] A. A. Pammu, K.-S. Chong, W.-G. Ho, and B.-H. Gwee, "Interceptive side channel attack on aes-128 wireless communications for iot applications," in *2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2016, pp. 650–653.
[49] M. Bozdal, M. Randa, M. Samie, and I. Jennions, "Hardware trojan enabled denial of service attack on can bus," *Procedia Manufacturing*, vol. 16, pp. 47–52, 2018.
[50] W. Hu, C.-H. Chang, A. Sengupta, S. Bhunia, R. Kastner, and H. Li, "An overview of hardware security and trust: Threats, countermeasures, and design tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 40, no. 6, pp. 1010–1038, 2020.
[51] H. D. Foster, A. C. Krolnik, and D. J. Lacey, *Assertion-based design*. Springer Science & Business Media, 2004.
[52] M. Boule and Z. Zilic, "Automata-based assertion-checker synthesis of psl properties," *ACM Transactions*

on Design Automation of Electronic Systems (TODAES), vol. 13, no. 1, pp. 1–21, 2008.

[53] 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.

[54] H. Al-Shaikh, A. Vafaei, M. M. M. Rahman, K. Z. Azar, F. Rahman, F. Farahmandi, and M. Tehranipoor, "Sharpen: Soc security verification by hardware penetration test," in *2023 28th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2023, pp. 579–584.

[55] (2018) American fuzzy lop (afl) fuzzer. [Online]. Available: http://lcamtuf.coredump.cx/afl/

[56] (2018) Honggfuzz. [Online]. Available: http://honggfuzz.com/

[57] 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.

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

[59] (2024) Verilator. [Online]. Available: https://www.veripool.org/verilator/

[60] K. Z. Azar, M. M. Hossain, A. Vafaei, H. Al Shaikh, N. N. Mondol, F. Rahman, M. Tehranipoor, and F. Farahmandi, "Fuzz, penetration, and ai testing for soc security verification: Challenges and solutions," *Cryptology ePrint Archive*, 2022.