# SoCureLLM: An LLM-driven Approach for Large-Scale System-on-Chip Security Verification and Policy Generation

Shams Tarek, Dipayan Saha, Sujan Kumar Saha, Mark Tehranipoor, Farimah Farahmandi
University of Florida {shams.tarek, dsaha, sujansaha}@ufl.edu, {tehranipoor, farimah}@ece.ufl.edu

***Abstract*—Contemporary methods for hardware security verification struggle with adaptability, scalability, and availability due to the increasing complexity of the modern system-on-chips (SoCs). Large language models (LLMs) have emerged as a viable approach to address these shortcomings in security verification because of their natural language understanding, advanced reasoning, and knowledge transfer capabilities. However, their application to large designs is limited by inherent token limitation and memorization constraints. In this paper, we introduce SoCureLLM, an LLM-based framework that excels in identifying security vulnerabilities within SoC designs and creating a comprehensive security policy database. Our framework is adaptable and adept at processing varied, large-scale designs, overcoming the abovementioned issues of LLM. In evaluations, SoCureLLM detected 76.47% of security bugs across three vulnerable RISC-V SoCs, outperforming the state-of-the-art security verification methods. Furthermore, assessing three additional large-scale RISC-V SoC designs against various threat models led to the formulation of 84 novel security policies, enriching the security policy database. Previously requiring extensive manual effort to craft, these newly generated security policies can be used as guidelines for developing secured SoC designs.**

***Index Terms*—LLM, Hardware Security, Verification, Vulnerability Detection**

## I. Introduction

Modern System-on-Chips (SoCs) integrate third-party intellectual property (3PIP) from global vendors, making security verification crucial due to their complexity and the horizontal development process. Hardware security vulnerabilities may occur due to unintentional design mistakes, deliberate actions by malicious insiders, security-unaware optimization by CAD tools, or flaws within test and debug infrastructures. The rise in advanced microarchitectural attacks highlights the need for robust SoC security verification to protect security assets such as sensitive user data, cryptographic keys, and configuration information. Hardware vulnerabilities can cause massive financial losses, and product recalls in the semiconductor industry [1]. To address this, there is an industry shift towards a security development lifecycle (SDL) for SoCs [2]. Nonetheless, current verification tools focus more on functional than security aspects, often missing critical vulnerabilities in the pre-silicon phase.

Hardware security vulnerabilities can be of different types, such as information leakage, access control violation, unauthorized memory access, fault injection attacks, and side-channel attacks. To address these vulnerabilities, recent research predominantly targets establishing security verification frameworks for SoC designs, employing methods such as information flow tracking [3], assertion-based security property verification [4], fuzzing [5], runtime verification monitor [6], and Concolic testing [7]. However, these methods fall short of scalability for large designs. They are not adaptable enough to use in other designs without notable modifications, have limited coverage, and demand significant computational resources [8]. These approaches also necessitate extensive manual analysis of designs, identifying threat models and vulnerabilities to formulate effective security policies for thorough verification. Furthermore, the commercial tools available for security verification often require in-depth knowledge to operate effectively.

To address such limitations of contemporary verification methods, there is a pressing need for a solution that possesses a nuanced understanding of complex SoC designs and can transfer knowledge of one design to another to uncover vulnerabilities effectively but requires the least manual intervention. Large language models (LLMs), with their advanced pattern recognition, natural language understanding, advanced reasoning, and knowledge transfer capabilities, stand out as promising candidates for this task. LLMs can analyze design documentation and codebases to learn the specific characteristics of one design and apply this knowledge to another, potentially revealing security weaknesses that are not immediately apparent. Moreover, LLMs can potentially automate parts of the verification process, significantly reducing the need for manual analysis. However, existing LLMs maintain a fixed context length[1], which is not long enough to accommodate a large design. Hence, current LLM-based hardware security solutions focus only on small hardware designs [9] because of token[2] limitation and limited memorization[3] issues.

To put a solution together, we introduce SoCureLLM, an LLM-based hardware security verification technique that solves the abovementioned problems by integrating the methodical partitioning of complex designs, enriching the analysis with contextual summarization for continuity, and employing refined prompting strategies for targeted exploration. Along with the potential to detect hardware vulnerabilities,

[1]Context Length: The number of tokens an LLM can process at once.
[2]Token: The smallest unit of text LLM processes.
[3]Lack of Memorization: Limitation of LLMs in retaining and recalling information over an extended period.

SoCureLLM also creates a comprehensive security policy database, which is essential for ensuring the security robustness of hardware designs.

The key contributions of our work are summarized as follows:

- SoCureLLM is the first LLM-based hardware security verification framework to handle large-scale SoC designs.
- The framework exhibits scalability and flexibility, effectively addressing the token limitation and memorization problems typically associated with LLMs.
- We introduce an automated vulnerability detection method within SoCureLLM, which thoroughly evaluates designs for potential security flaws.
- SoCureLLM integrates a procedure for creating an extensive security policy database through threat model-based assessments of the large designs.
- Our framework's performance is demonstrated through rigorous evaluation of various buggy SoC designs, and we compare the performance with existing security verification methods.

In the remainder of this paper, Section II narrates the preliminaries and related works. Later, Section III describes the proposed methodology. Afterward, the experimental setup, a case study, and results are discussed in Section IV. Finally, Section V concludes the paper.

## II. Background & Prior Works

In Section II-A, we initially provide an overview of the foundational concepts related to LLMs and SoC security to familiarize the reader with the basics. This foundational knowledge is essential for comprehending the proposed methodology and discussions presented in this paper. Subsequently, Section II-B explores the relevant research works.

### A. Preliminaries

#### 1) Large Language Model (LLM)

LLMs, a subset of generative artificial intelligence (GAI), specialize in processing and generating natural language. They are trained on extensive text datasets, which enables them to perform a wide range of linguistic tasks, such as extracting knowledge from the provided information. The transformer architecture [10] serves as the foundation for the most advanced LLMs such as GPT-3 [11], and GPT-4 [12]. However, variations in model architecture can lead to differences in their operational principles and functions. Specifically, generative pre-trained transformers (GPTs) employ a decoder-only architecture that follows an autoregressive process, where the model uses the tokens it has already generated as context for making subsequent predictions. This architecture allows GPT models to generate coherent and contextually relevant text by building upon each token it produces, iteratively refining the output as it progresses through the sequence. Recently, with the increase in model size, such models have shown advanced reasoning and coding abilities.

#### 2) In-context Learning & Prompting

In-context learning (ICL) [11] leverages the innate capability of LLMs to adapt their outputs based on given examples or instructions within the input prompt. This allows the models to perform tasks without explicit training or fine-tuning for each new task — a process that would typically require substantial computational resources and time.

Prompting, in turn, is the mechanism by which users communicate with LLMs, providing them with the context or examples they need to learn in-context. The quality of prompting directly influences the success of in-context learning and, thus, the performance of the model. A well-engineered prompt can succinctly convey the task requirements and guide the LLM toward the desired output. With this importance in mind, a variety of prompting techniques, such as chain of thought (CoT) [13], have emerged. These techniques aim to optimize how LLMs are guided during inference to produce the most accurate and relevant responses possible.

#### 3) SoC Security Vulneabilities

Hardware security vulnerabilities are weak points in the design of a system that attackers can exploit. For example, during interrupt handling, if the processor is not running at the highest privilege level, any attacker can exploit the interrupt service routine (ISR), potentially gaining control over the entire system. The vulnerabilities might arise unintentionally due to design mistakes or intentionally through rogue employees or compromised 3PIPs, often revealed during the transition from models to physical implementations. CAD tools may inadvertently contribute to these bugs during synthesis as they have not been designed with security in mind. Test and debug infrastructures could also expose the system to post-manufacture security breaches. The common vulnerabilities and exposures (CVE) and common weakness enumeration (CWE) by MITRE are the primary references for classifying these issues, offering a comprehensive list of prevalent hardware and software vulnerabilities. Table I shows a portion of the vulnerabilities considered in this paper.

#### 4) Security Policy

Modern SoC designs contain many sensitive information that requires protection from unauthorized access, guided by principles of confidentiality, integrity, and availability [14]. Security policies translate these principles into practical design constraints, helping IP designers and design integrators in establishing and executing protective measures. Security policies vary, with some ensuring access control, protocol verification, and memory defense, while others guard against information leakage, fault injection, and denial of service. However, formulating these policies for modern SoCs is complex and makeshift, demanding considerable manual labor based on specific customer needs and design details. To address the aforementioned privilege escalation issue, a verification engineer must understand the ISR's specific implementation in the design under test, as well as the potential consequences of privilege escalation. Crafting such a security policy will be time-intensive due to these detailed requirements.

TABLE I: A PORTION OF THE HARDWARE VULNERABILITIES USED IN THIS WORK

| CWE ID | Vulnerability Type | Vulnerability Description | CIA Violation | Required Security Policy | Security Implication |
|---|---|---|---|---|---|
| CWE-1198 | Privilege level | Improper privilege assignment | Confidentiality | The privilege level must revert to the highest level upon returning from a debug session | Unauthorized access control |
| CWE-269 | Privilege level | Improper interrupt handling | Confidentiality | The process of interrupt handling should only occur in the highest privilege level of a processor core | Unauthorized access control |
| CWE-269 | Privilege level | Illegal changes in enable signals | Confidentiality, integrity | Privilege levels should remain unchanged during instruction execution, and updates to read/write enable for control and data registers should not occur during privilege transfers | Unauthorized access control |
| CWE-250 | Privilege level | Illegal virtual page access request | Integrity | The Memory Management Unit (MMU) must consistently uphold the appropriate privilege levels for accessing virtual memory pages | Unauthorized access control |
| CWE-250 | Privilege level | Illegal instruction execution | Confidentiality, integrity | Instructions with the highest privilege level of a process should maintain their privilege status throughout the execution | Unauthorized access control |
| CWE-1260 | Memory access | Memory range overlapping | Confidentiality | The memory address ranges for all security-critical IPs within an SoC need to be distinct and clearly defined | Unauthorized memory access |
| CWE-284 | Memory access | Illegal DMA access | Confidentiality | The DMA controller must be monitored while accessing the protected memory region | Unauthorized memory access |
| CWE-1245 | FSM | Vulnerable FSM encoding | Integrity | When transitioning between two consecutive unprotected states, the Hamming distance between them must be 1 | Access to protected states |
| CWE-1271 | Reset related | Important register values were inaccurately cleared during reset | Confidentiality | Upon reset, all critical registers must undergo initialization | Information leakage |
| CWE-506 | Hardware Trojan | A Trojan in the CSR module, causing the Supervisor User Memory Access (SUM) bit of the MSTATUS register to be set to 1, enabling access to user-level virtual pages from supervisor mode | Confidentiality | When supervisor mode does not have access to user-level pages, the SUM bit must be set to 0 | Information leakage |
| CWE-506 | Hardware Trojan | Trojan implanted in decoder module for tracking hardware exceptions, causing CPU to halt at threshold | Availability, integrity | The processor needs to monitor unauthorized hardware exceptions | Denial of service (DoS) |
| CWE-310 | Crypto IP | A Trojan within the AES engine, resulting in the encryption process being halted for an uncommon plaintext byte, which acts as a trigger. | Availability, integrity | The AES engine should signal DONE after completing the 10th round of operations for a 128-bit AES cipher key | Denial of service (DoS) |
| CWE-310 | Crypto IP | A Trojan leading to the leakage of the encryption key when encountering a rare plaintext byte | Confidentiality | The AES encryption key must not be exposed | Information leakage |
| CWE-1244 | Debug Module | Illegal JTAG access | Confidentiality | Every debugging session should demand a password, with the password verification system correctly implemented | Information leakage |
| CWE-1244 | Debug Module | Illegal JTAG access | Confidentiality | For every, debug request, there should be a bitwise check after every reset | Information leakage |

### B. Prior Works

As security verification techniques in the SoC designs evolve, H. Witharana *et al.* [4] introduced an automated framework for generating security assertions tailored to specific vulnerabilities. However, the effort is limited to a specific set of vulnerabilities. B. Ahmad *et al.* [15] developed security-specific scanners to identify and analyze hardware CWEs and evaluated their performance in different open-source designs. This method still involves manual intervention. Machine learning-based hardware verification methods [16] face challenges such as design dependency, data scarcity, scalability, and efficiency problems, limiting their broader applicability and effectiveness. Recent research efforts [9], [17], [18] using LLM have begun to address the detection and mitigation of hardware vulnerabilities. The study in [19] used LLMs to identify and map security vulnerabilities in SoC designs to relevant CWEs, generate assertions, and enforce security policies, with demonstrated efficacy in open-source SoC benchmarks. Yet, the full potential of LLMs in this domain remains largely unexplored.

## III. Proposed SoCureLLM Framework

Identifying vulnerabilities early in the design process is paramount for upholding the integrity, confidentiality, and availability of the designs. A comprehensive database of security policies greatly enhances the validation of any security verification framework, including the detection of vulnerabilities. Our proposed framework intends to harness the capabilities of LLMs to fulfill these objectives. Within this framework, the LLM adopts different names depending on its function: it is the "Summarizer LLM" when condensing information, the "Detector LLM" when scanning for security policy violations, the "Finder LLM" when locating potential attack points, and the "Generator LLM" when crafting security policies for IPs. The important steps of this framework are presented in Figure 1. The following sections give detailed explanations of each step.

Step ①: Partitioning of Designs into Smaller Code Snippets: In this framework, we employ large-scale, open-source buggy SoC designs as our initial input. Since LLM cannot handle such a large volume of tokens simultaneously, our framework adopts a design partitioning approach. A Python-based Verilog code divider splits the design into a series of smaller code snippets based on various rules. Initially, the individual IP modules are separated from each other. Subsequently, each IP is further subdivided into smaller code segments. These code snippets are then stored as text files in a local repository. While the framework affords the flexibility to segment the design in a random manner without necessarily encompassing complete functionality, we have established a set of rules guided by multiple Verilog constructs to dictate the segmentation. These rules are based on factors such as the number of lines of code, the quantity of "always" blocks within the module, the presence of case statements within the module, and any instantiation of IP within the module. For example, consider a control & status register (CSR) IP module comprising 1500 lines of code; it can be divided into 15 distinct code snippets. The first snippet contains all inputs, outputs, and parameters. The second and third snippets correspondingly encapsulate the CSR Read logic and the CSR Write & update logic. The rest of the code is divided into code snippets similarly, depending on the other rules.

Step ②: Generation of Additional Context through Summarization: LLMs often struggle with context retention in interactions. Apart from creating smaller code snippets, we need to preserve and provide the information from earlier segments during the complete analysis. To capture the most significant information from each code snippet, we generate a concise technical natural language summary for every snippet. This stage employs the "Summarizer LLM" to craft these summaries. A carefully constructed context prompt aids the LLM in producing a thorough summary that includes the signal, security, and functional information from the code snippets. These summaries are subsequently stored as text files within a local storage repository. Notably, these summaries are rigorously constrained to a concise fixed length. These condensed summaries play an important role in step 3 as additional context in the framework.

The following textbox presents an example summary derived from a segment of code pertaining to the CSR module. This summary contains essential functional elements, such as procedural blocks, as well as security-related details, including privilege levels. Consider a scenario where the security policy prompt requires verification of a particular policy: after a debug session, the processor must revert to the highest privilege level. When the code snippet is fed to an LLM along with this summary (of a different snippet), it will work as an additional context regarding the processor's privilege status.

**Example of summary**

This Verilog code snippet consists of three always_comb blocks: privilege_check, exception_ctrl, and wfi_ctrl. The privilege_check block checks for privilege violations when performing operations on the Control Status Register (CSR). For a violation, it sets the privilege_violation flag. The exception_ctrl block handles exceptions, setting the cause to ILLEGAL_INSTR and marking the exception as valid if there's access exception or privilege violation. The wfi_ctrl block controls the Wait for Interrupt (WFI) instruction. It sets the wfi_d flag based on the status of mip_q, debug_req_i, irq_i[1], debug_mode_q, csr_op_i, and ex_i.valid. Machine (M) mode is the highest privilege level for the CSR module.

Step ③: Detection of Security Policy Violation: Following the generation of summaries and code snippets, in this phase, our framework proceeds to employ the "Detector LLM". In this particular step, we employ a predefined policy prompt that contains the security policies to be checked, with the aim of identifying a spectrum of hardware vulnerabilities within

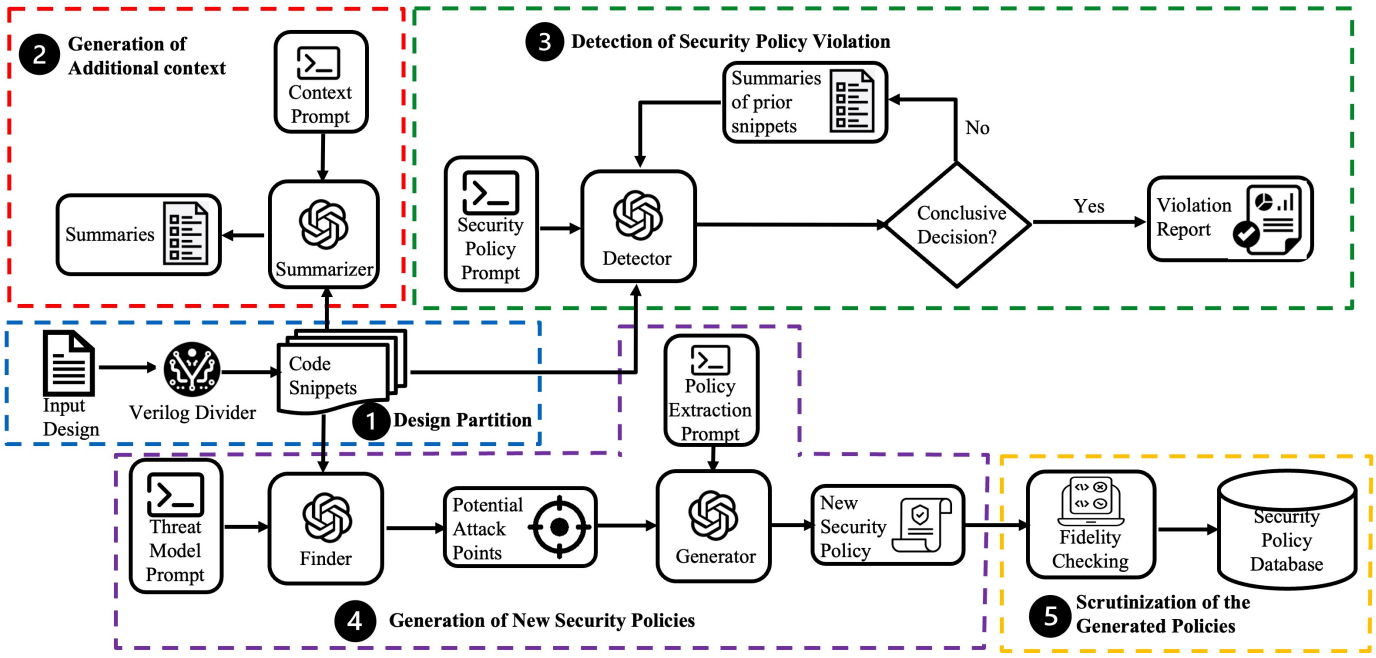



Fig. 1: Overall flow of the proposed SoCureLLM framework

the code snippets. To ensure the correctness of the decisions made by the LLM, we follow a specific prompting strategy while crafting the security policy prompt. The details of this prompting strategy are comprehensively described in Section III-A.

In this recursive step, the Detector LLM analyzes each code snippet against a security policy prompt. The LLM then produces a response that may confirm or deny violations or seek more context for a definitive ruling on policy compliance. Outcomes are marked as conclusive if a clear affirmation or negation of policy violation is confirmed. Otherwise, the decisions are inconclusive. For conclusive findings, the LLM generates an in-depth violation report detailing the findings and explanations.

In contrast, when the decision remains inconclusive, we need to provide additional context from the prior code snippets. Hence, the current code snippet, along with the policy prompt and the summary of the preceding code snippet, are resubmitted to the LLM. This process repeats until the LLM gives a definite answer on policy violations or until it has reviewed all prior summaries. If still undecided, the LLM creates a report stating it is inconclusive and moves on to the next code snippet. For instance, if our design is divided into 100 code snippets and we are examining the 4th snippet using the LLM with the policy prompt, it will either produce a clear violation report or, if the decision is inconclusive, we provide the LLM with the 4th snippet, policy prompt, and a summary of the 3rd snippet for further analysis. This step is repeated, incorporating summaries of the 2nd and 1st snippets, if needed, until the LLM reaches a conclusive decision. If it remains unresolved after all summaries are reviewed, the LLM compiles an inconclusive report for the 4th snippet before moving on to the 5th.

Given the variability in outcomes across different code snippets—where some may comply with security policies and others may not—the framework adopts a cautious approach. The framework prioritizes critical findings; thus, if even one snippet exhibits non-compliance or a security flaw, the whole design is marked as vulnerable. In the context of our previous example, even if 1 out of 100 code snippets is found to have issues after a thorough iterative assessment, the framework would still categorize the overall design as potentially vulnerable. Algorithm 1 concisely depicts the hardware vulnerability detection flow (step ①, ② and ③) of the proposed framework.

Step ④: Generation of New Security Policies: In this stage, we require the inclusion of code snippets as input to the "Finder LLM". Distinct from the policy prompt approach, we employ an alternative strategy. We craft a threat model prompt containing a curated selection of threat models capable of investigating breaches in integrity, confidentiality, or availability within the design. This prompt is meticulously designed to encompass various scenarios stemming from the chosen threat models. The threat models in consideration include the following categories: information leakage, denial of service, confidentiality attack, privilege escalation, access control violation, and unauthorized memory access. Upon submitting these inputs, the Finder LLM formulates a response that identifies potential points of vulnerability within the design. These vulnerabilities can manifest as sensitive signals, specific conditions, or distinct case statements. The "Generator LLM" is supplied with potential attack points in conjunction with the policy extraction prompt. This policy

**Algorithm 1** Detection of Security Policy Violation

**Require:** $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$: Set of code snippets, $V$: Set of vulnerabilities, $P_v$: Security policy prompt for $v \in V$, $P_c$: Context prompt, $\mathcal{L}_{\text{summary}}$: Summarization LLM, $\mathcal{L}_{\text{detection}}$: Decision-making LLM, $\mathcal{C}$: Conclusive decisions, $O = \{o_{1,v}, o_{2,v}, \ldots, o_{n,v}\}$: Intermediate detection outcomes, $R_v$: Vulnerability report.
**Ensure:** $\{R_v \mid v \in V\}$: Set of reports for each vulnerability.

$\mathcal{S} \leftarrow \mathcal{D}$
Store $\mathcal{S}$ in memory
**for all** $v \in V$ **do**
  **for all** $s_i \in \mathcal{S}$ **do**
    $r_i \leftarrow \mathcal{L}_{\text{summary}}(P_c, s_i)$ /* LLM creates summary */
    $o_{i,v} \leftarrow \mathcal{L}_{\text{detection}}(s_i, P_v)$ /* LLM performs security check */
    **if** $o_{i,v} \notin \mathcal{C}$ **then**
      **for** $j = i - 1$ downto 1 **do**
        $o_{i,v} \leftarrow \mathcal{L}_{\text{detection}}(s_i, P_v, r_j)$ /* Includes summary */
        **if** $o_{i,v} \in \mathcal{C}$ **then**
          break
  $R_v \leftarrow \max_{\text{severity}}(O)$ /*Reports as buggy, if detected in any snippet*/
**return** $\{R_v \mid v \in V\}$

**Algorithm 2** Generation of Security Policy

**Require:** $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$: Set of code snippets, $T$: Set of threat models, $\mathcal{L}_{\text{finder}}$: Finder LLM, $\mathcal{L}_{\text{generator}}$: Generator LLM, $P_t$: Threat model prompt, $P_e$: Policy extraction prompt, $\mathcal{F}_c$: Fidelity Checking.
**Ensure:** $SP = \{sp_{1,1}, sp_{1,2}, \ldots, sp_{n,m}\}$: Set of security policies approved after fidelity check.

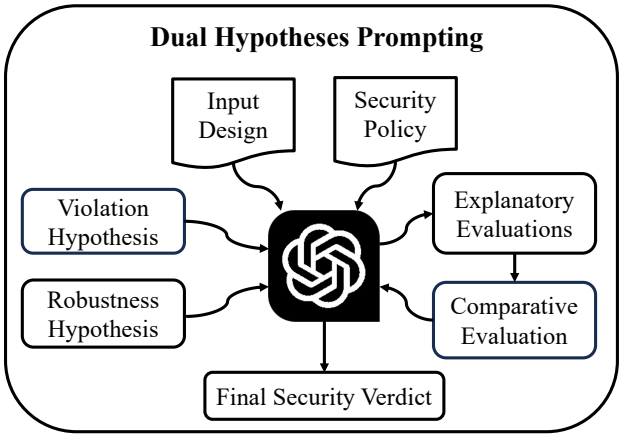
Initialize $SP = \emptyset$
**for all** $s_i \in \mathcal{S}$ **do**
  Initialize $AP = \emptyset$
  **for all** $t \in T$ **do**
    $AP_i \leftarrow \mathcal{L}_{\text{finder}}(s_i, P_t)$ /* LLM identifies potential attack points*/
    $AP \leftarrow AP \cup AP_i$ /* Aggregate attack points */
  **for all** $ap \in AP$ **do**
    $sp_{i,ap} \leftarrow \mathcal{L}_{\text{generator}}(s_i, ap, P_e)$ /* LLM creates security policy */
    $passed \leftarrow \mathcal{F}_c(sp_{i,ap})$ /* Manual scrutiny evaluates the policy */
    **if** $passed$ **then**
      $SP \leftarrow SP \cup \{sp_{i,ap}\}$ /* Add only approved policies to SP */
**return** $SP$

extraction prompt directs the Generator LLM to transform these potential attack points into actionable security policies.

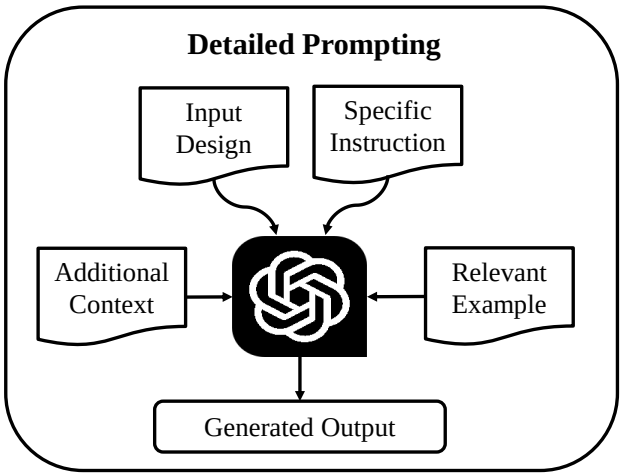
Step ⑤: Scrutinization of Generated Policies: The generated policies are then subjected to thorough examination. It is possible that certain policies derived may coincide with the security policies mentioned in the security policy prompt. Additionally, some of the policies might be impractical to implement. Therefore, this process of scrutinization filters out such policies, retaining only the viable ones in an extensive security policy database. Algorithm 2 concisely represents the security policy generation aspect (step ①, ④ and ⑤) of the proposed framework.

### A. Prompting Strategies

As discussed in Section II-A2, since the performance of an LLM is significantly influenced by the prompting technique employed, our framework is designed with tailored prompting strategies to enhance the performance.




(a) Overview of dual hypotheses prompting




(b) Overview of detailed prompting

Fig. 2: Different prompting strategies used in SoCureLLM

While formulating the security policy prompt in step ③ of the proposed framework, we adopt a novel prompting strategy named the "dual hypotheses prompting approach", shown in Figure 2a. In this prompting strategy, the LLM is tasked first to consider two assumptions: violation and robustness hypotheses. In the 'violation hypothesis', the LLM is asked to assume that there is a violation of the given security policy within the design and generate plausible explanations for such violation. Then, in the 'robustness hypothesis', the LLM is prompted to consider the contrary, focusing on the absence of vulnerabilities by highlighting robust security features of the design. By contrasting these two scenarios, the strategy facilitates a comprehensive evaluation, comparing the explanations to ascertain which is more convincing. The strategy concludes by evaluating the most logical explanation to determine the presence or absence of the violation of the security policy.

This approach essentially mirrors scientific hypothesis testing, adapted for the domain of SoC security. It enhances critical evaluation by contrasting scenarios of risk, leading to more informed and rational decision-making in SoC security.

Furthermore, we follow another prompting technique (shown in Figure 2b) in designing the context, threat model, and policy extraction prompts. In thsis method, the LLM is guided by specific instruction, additional context, relevant examples, and self-scrutiny instruction along with the input design to produce enriched output. In this case, specific instructions ensure the process adheres to defined criteria. Additional context and relevant examples enrich decision-making by providing depth and situational awareness, while self-scrutiny instruction is a critical introspective step where the LLM reviews and optimizes its own processes to ensure accuracy and relevance.

## IV. Experiments

In order to confirm the efficacy of the proposed framework SoCureLLM, an extensive experiment has been performed on multiple SoC designs. Section IV-A narrates the experimental setup. Next, Section IV-B explains the methodology through a case study and Finally, Section IV-C describes the experimental results with analysis.

### A. Experimental Setup

Although any LLM can be used in the SoCureLLM framework, we applied the GPT-4 [12] API within our framework as the core LLM. We focused on detecting security policy violations in three vulnerable RISC-V SoCs: hack@dac 2018 (PULPissimo core [20]), TrustHub benchmark, and HOST 2022 (Ariane core). The hack@dac 2018 SoC is a renowned SoC with known bugs, often used as a benchmark in various verification methods. Since the GPT-4 model was trained with data up to September 2021, it might be familiar with the hack@dac 2018 SoC. To circumvent issues related to the availability of design information, we have opted for two other SoCs, HOST 2022 and the TrustHub benchmark, as they are mainly closed-source. We utilized three additional open-source RISC-V SoCs for the security policy generation flow: CVA6, lowRISC(ibex), and CV32E40P. A Python-based tool was developed to segment Verilog code for analysis. Simulations were conducted using Verilator and Cadence Incisive, while Cadence Jaspergold SPV and FPV tools were employed for formal verification.

### B. Case Study

To evaluate the effectiveness of our proposed framework for detecting vulnerabilities in SoCs, we selected three RISC-V-based SoCs as detailed in the preceding section. This case study illustrates the process of identifying a vulnerability within the debug module of the TrustHub benchmark and the hack@dac 2018 SoC. The bug is located within the JTAG TAP controller module and is characterized by a flawed password verification mechanism. This mechanism is crucial for any debugger utilizing the JTAG module, with the presumption that an attacker could mimic a debugger to bypass this password verification. The vulnerability lies in the "`correct`" counter, which tallies the number of correct password bits during each attempt. The counter increases when the input password bit aligns with the actual password bit, unlocking the module when the counter matches or exceeds the total number of password bits. However, during the reset state, this "`correct`" counter was not cleared to zero due to the bug, leaving the module perpetually unlocked after a single successful entry, thereby creating a potential backdoor for attackers.

SoCureLLM initially segments the entire SoC into multiple code snippets based on the integrated IPs and peripherals. Subsequently, these IPs are broken down into smaller code snippets, which are saved as local text files alongside a technical summary for each. The process is not repeated for the same SoC for other security policies. To identify security policy violations (Step 3 in our framework), each snippet is analyzed by the detector LLM with a security policy prompt, employing the dual hypothesis prompting technique. The specific prompt used in the proposed framework for detecting the above-mentioned JTAG module vulnerability is

**Dual Hypothesis Prompting (C.1)**

**//Input Code Snippet & Summaries will be added to the prompt automatically//**

*Security Policy: The debug module's password-checking mechanism must be logically correct, and all related signals should be cleared during the reset state.*

Consider two scenarios: First, assume that the security policy is breached in the provided code snippet and the technical summaries. Write an analysis based on this assumption.
Second, assume that the security policy is adhered to in the provided code snippet and the technical summaries. Write an analysis based on this assumption.

Compare the two analyses and identify the more logical one. If you reach to a conclusive decision, provide a detailed explanation based on your analysis. If your decision is inconclusive, you will request additional summaries from the previous snippets sequentially.

Your response should strictly maintain the following structure:

**Analysis under Breach of Security Policy Assumption:**
# add your response here

**Analysis under Adherence to Security Policy Assumption:**
# add your response here

**Final Verdict:**
# add your response here
# "The decision is conclusive" or "The decision is inconclusive"

**Response from GPT-4 without correct Summary (C.2)**

**Analysis under Breach of Security Policy Assumption:**
Retention of previous state information (`passchk`, `correct`, `bitmask`) in other non-reset states can be exploited if these variables are not appropriately re-initialized when expected. An attacker could manipulate the transition conditions by ensuring that `correct` hits the specific value, thereby setting `passchk` to 1 without a legitimate authentication sequence.

**Analysis under Adherence to Security Policy Assumption:**
If the system enters the reset state, it should explicitly clear signals like `passchk`, `correct`, `bitmask`, and others to adhere to the policy that all related signals should be reset. This ensures that any remaining or leftover signal values from prior operations that could jeopardize security are eliminated each time the machine is reset.

**Final Verdict:**
A definitive conclusion on vulnerability cannot be reached without further details on other parts of the system. Additional snippets or a broader view of the full implementation would be necessary to determine compliance or breach of the stated security policy conclusively.

**The decision is inconclusive.**

**Response from GPT-4 with correct Summary (C.3)**

**Analysis under Breach of Security Policy Assumption:**
Because `correct` is not cleared, an attacker who successfully unlocks the JTAG module once can bypass the password-checking mechanism in subsequent attempts. The counter value remains above the conditional check (`correct == 32'h8000_FFFF`), automatically setting `passchk` to 1, which should only happen after a successful password check.
This persistent `correct` signal effectively disables the password-checking mechanism after one successful entry, making the module vulnerable to unauthorized access without further password verification. The module stays unlocked indefinitely, a severe security risk allowing persistent, unauthorized access.

**Analysis under Adherence to Security Policy Assumption:**
With the additional context provided, it becomes evident that despite resetting `passchk`, failing to reset `correct` is a significant oversight. This non-compliance with the security policy undermines any other security measures implemented in the code.
The initial analysis assumed the other signals might be correctly managed based on the snippet. However, knowing now that `correct` is not reset, it is clear that the implementation does not adhere to the essential security policy of clearing all sensitive signals during a reset.

**Final Verdict:**
Based on the combined analysis of both code snippets and the new summary provided, the code is definitively vulnerable. The failure to reset the `correct` signal allows for an easily exploitable loophole that can leave the module unlocked indefinitely, providing unauthorized access without subsequent password verification. This represents a serious security flaw requiring immediate correction to comply with security policies and ensure the integrity of the module.

**The decision is conclusive.**

shown in Textbox C.1. This prompt instructs the detector LLM to evaluate based on two hypotheses: detecting security policy violations and ensuring compliance with the security policy. From these assessments, the detector LLM determines whether it can make a definitive decision. If the decision is conclusive, it produces a detailed report of the violation. If the outcome is inconclusive, the SoCureLLM framework supplies the summaries of previous code snippets sequentially until a definitive conclusion is achieved. The prompt additionally ensures that the detector LLM consistently delivers structured responses. In the case of this vulnerability, one snippet encompasses the actual implementation of the password-checking logic, and a separate snippet contains the reset state that includes the bug. When the snippet with the password-checking logic is analyzed without any summary of the previous code snippets, the response from the detector LLM, as displayed in Textbox C.2, does not yield a definitive conclusion. Nonetheless, it outlines the potential implications of both violating and adhering to the security policy as outlined in the prompt. However when the detector LLM reviews the technical summary describing the reset state, where the bug resides, along with the provided code snippet, it conclusively identifies the snippet as vulnerable, as indicated in Textbox C.3. Notably, the security policy violation was so evident that the detector LLM found no grounds to support policy adherence. This case study emphasizes the importance of the technical summaries in pinpointing security policy violations and mitigating the limitations of LLM memorization, affirming the value of feedback and segmentation-based examination within SoC designs through SoCureLLM, thus enhancing the accuracy of security assessments.

### C. Evaluation

We examined three buggy SoCs containing 68 hardware vulnerabilities in the security policy violation evaluation. Due to space limitations, we cannot detail each vulnerability, encouraging readers to go through [20], [21] for comprehensive information about the vulnerabilities and the security policies used in the policy prompt. A general overview of the bugs can be found in Section II-A3. Table II highlights the effectiveness of our framework in identifying vulnerabilities through security policy violations compared to a more generic, open-ended, prompt-based security assessment. The comparison indicates that SoCureLLM has a true positive rate (TPR) of 76.47%, significantly outperforming the open-ended method's 35.29%, showcasing its higher efficiency in accurately detecting vulnerabilities present in the design. This emphasizes the importance of specific prompting and additional context through

summarization in uncovering security policy violations. Also, the framework attains a higher true negative rate (TNR) of 96.46%, showing its reliability in confirming the absence of bugs and minimizing false positives. The lines of code (LoC) in the table demonstrate our framework's capability to process large-scale designs, effectively addressing the token limitation and memorization issues in LLMs.

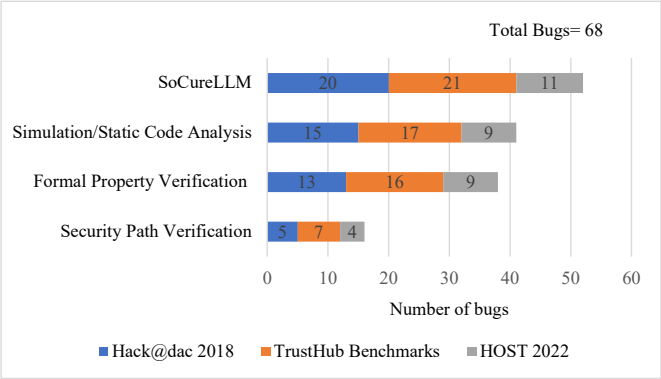



Fig. 3: Comparison of our proposed framework with the contemporary security verification techniques in terms of the number of bugs detected successfully in the SoCs under test

Figure 3 compares traditional security verification methods (formal verification, simulation, static code analysis) and SoCureLLM. The results indicate that SoCureLLM outperforms other verification approaches in bug detection, specifically detecting 16.18% more bugs than the closest competitor (simulation). Although simulation or static code analysis can catch some bugs, they are generally slower than the other methods. Moreover, it takes significant manual effort to prepare the test stimuli for the process. Regarding formal verification, it can pinpoint vulnerabilities when there are clear deviations from specified requirements, such as value mismatches or parameter inconsistencies (i.e., memory range overlaps, privilege escalations, and reset-related issues). SoCureLLM is faster and more user-friendly as it only needs design specifications to find vulnerabilities, unlike formal tools that take more time and need detailed knowledge of the specifications and implementation. For example, autoSVA [22], a formal verification method, takes around one hour to assess the MMU module of the Ariane core, whereas SoCureLLM only takes 15 minutes to analyze the module entirely for verification of all the security policies. Nonetheless, some vulnerabilities, such as hardware Trojans, are challenging to identify using any verification method due to their latent characteristics. Despite detailed context scenarios, these remain a challenge for SoCureLLM to detect, as others. In terms of expenses, a complete analysis of the hack@dac 2018 SoC for vulnerability detection costs just around $7.5, a figure that is significantly economical compared to the costs of time-to-market and commercial licensing tools. Furthermore, this cost is continually decreasing with the advancement of LLM technology.

Figure 4 illustrates the quantity of potential weak points and the security policies derived from chosen threat models for

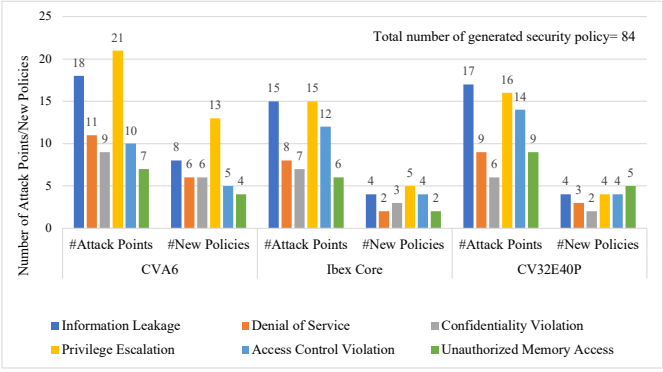



Fig. 4: Number of potential attack points for different threat model analyses in each SoC and newly generated security policies after scrutinization.

three SoCs. The weak points are consistently about twice the number of security policies, attributed to multiple attack points leading to similar types of security policies. The SoCureLLM develops security policies by spotting potential vulnerabilities, but manual review is crucial to refine them. This step is necessary to eliminate overly unfeasible ones like those for brute force attacks and dismiss broad ones such as suggesting the use of physical unclonable functions (PUFs) or cryptomodules. Despite these issues, the manual scrutiny process used here is still more efficient than creating security policies for each threat model from the ground up. A noticeable decrease in the count of security policies across different designs is observed due to the scrutinization process eliminating repetitive policies from each SoC. Eventually, our framework generated 84 new security policies from the designs. A sample list of the generated security policies for the lowRISC (ibex) controller module is shown in the corresponding textbox.

**Sample of generated security policies**

- Monitor the 'debug_mode_o' signal to ensure it does not leak through side channels.
- Review state transitions for potential escalation vectors, especially around 'current_priv_lvl_i'.
- Implement rate-limiting and sanity checks on the 'irq_req_ctrl_i' signal to prevent IRQ flooding.
- Implement strict access control checks on debug and control registers.
- Audit the control flow for any unauthorized bypasses or weak checks.
- Implement bounds checking for memory accesses and handle exceptions for 'data_misaligned_i'.

## V. Conclusion

SoCureLLM offers a flexible, scalable solution to hardware security verification for large-scale SoCs, employing an LLM-based framework to effectively navigate the complexities of modern SoC designs. It outperforms traditional methods by efficiently detecting security vulnerabilities and enriching security policy databases. SoCureLLM not only solves the

TABLE II: Performance comparison between open-ended security assessment and proposed SoCureLLM in the detection of security bugs

| Design | IP | LoC | Open Ended Security Assessment | | | | | Security Policy Violation Assessment (SoCureLLM) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | # bugs in the module | # bugs detected | # bugs detected successfully | TPR | TNR | # bugs detected | # bugs detected successfully | TPR | TNR |
| **Hack@Dac2018** | Debug Unit | 715 | 7 | 5 | 4 | 0.571 | 0.950 | 8 | 6 | 0.857 | 0.900 |
| | GPIO | 408 | 5 | 3 | 2 | 0.400 | 0.955 | 4 | 4 | 0.800 | 1.000 |
| | CSR | 1510 | 3 | 1 | 0 | 0.000 | 0.958 | 2 | 2 | 0.667 | 1.000 |
| | RISC-V Core | 14635 | 7 | 3 | 2 | 0.286 | 0.950 | 9 | 5 | 0.713 | 0.800 |
| | AXI Interface | 810 | 1 | 0 | 0 | 0.000 | 1.000 | 1 | 1 | 1.000 | 1.000 |
| | Crypto Modules | 11606 | 4 | 2 | 1 | 0.250 | 0.957 | 2 | 2 | 0.500 | 1.000 |
| | **Total** | **29684** | **27** | **14** | **9** | **0.333** | **0.963** | **26** | **20** | **0.741** | **0.956** |
| **Trust-Hub Benchmark** | CSR | 1510 | 10 | 4 | 3 | 0.300 | 0.944 | 8 | 8 | 0.800 | 1.000 |
| | RISC-V Core | 14635 | 4 | 5 | 2 | 0.500 | 0.870 | 5 | 4 | 1.000 | 0.957 |
| | Decoder | 1418 | 4 | 2 | 1 | 0.25 | 0.957 | 4 | 3 | 0.750 | 0.957 |
| | MMU | 519 | 2 | 0 | 0 | 0.000 | 1.000 | 1 | 1 | 0.500 | 1.000 |
| | PMP | 278 | 1 | 0 | 0 | 0.000 | 1.000 | 1 | 1 | 1.000 | 1.000 |
| | AES | 12624 | 4 | 2 | 2 | 0.500 | 1.000 | 3 | 3 | 0.750 | 1.000 |
| | AXI Interface | 810 | 2 | 1 | 1 | 0.500 | 1.000 | 2 | 1 | 0.500 | 0.960 |
| | **Total** | **31794** | **27** | **14** | **9** | **0.333** | **0.970** | **24** | **21** | **0.778** | **0.982** |
| **HOST 2022** | CSR | 1510 | 4 | 3 | 2 | 0.500 | 0.900 | 3 | 3 | 0.750 | 1.000 |
| | Crypto Module | 8133 | 5 | 2 | 2 | 0.400 | 1.000 | 5 | 4 | 0.800 | 0.890 |
| | RISC-V Core | 14635 | 4 | 3 | 2 | 0.500 | 0.900 | 5 | 3 | 0.750 | 0.800 |
| | memory unit | 235 | 1 | 0 | 0 | 0.000 | 1.000 | 1 | 1 | 1.000 | 1.000 |
| | **Total** | **24513** | **14** | **8** | **6** | **0.423** | **0.952** | **14** | **11** | **0.786** | **0.923** |

challenges in the existing hardware verification techniques but also addresses the token limitation and memorization constraints of traditional LLMs. The future direction of this research involves fine-tuning an LLM specifically for hardware security verification to enhance its precision and effectiveness within the domain.

## REFERENCES


[1] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi, "V0ltpwn: Attacking x86 processor integrity from software," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1445–1461.

[2] 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*, 2012, pp. 116–121.

[3] 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*, 2017, pp. 1691–1696.

[4] H. Witharana, A. Jayasena, A. Whigham, and P. Mishra, "Automated generation of security assertions for rtl models," *J. Emerg. Technol. Comput. Syst.*, vol. 19, 2023.

[5] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *USENIX Security Symposium*, 2022, pp. 3237–3254.

[6] A. Kassem and Y. Falcone, "Detecting fault injection attacks with runtime verification," in *Proc. of the 3rd ACM Workshop on Software Protection*, 2019, p. 65–76.

[7] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 466–477, 2021.

[8] S. R. Rajendran, S. Tarek, B. M. Hicks, H. M. Kamali, F. Farahmandi, and M. Tehranipoor, "Hunter: Hardware underneath trigger for exploiting soc-level vulnerabilities," in *2023 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2023, pp. 1–6.

[9] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4043–4057, 2024.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[11] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," in *Advances in Neural Information Processing Systems*, vol. 33, 2020, pp. 1877–1901.

[12] OpenAI, "Gpt-4 technical report," 2023. [Online]. Available: https://arxiv.org/pdf/2303.08774.pdf

[13] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Adv. in Neural Information Processing Systems*, vol. 35, pp. 24 824–24 837, 2022.

[14] A. Basak, S. Bhunia, and S. Ray, "A flexible architecture for systematic implementation of soc security policies," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 536–543.

[15] B. Ahmad, W.-K. Liu, L. Collini, H. Pearce, J. M. Fung, J. Valamehr, M. Bidmeshki, P. Sapiecha, S. Brown, K. Chakrabarty *et al.*, "Don't cweat it: Toward cwe analysis techniques in early stages of hardware design," in *Proc. of the 41st IEEE/ACM Int. Conf. on Computer-Aided Design*, 2022, pp. 1–9.

[16] Z. Pan and P. Mishra, "A survey on hardware vulnerability analysis using machine learning," *IEEE Access*, vol. 10, pp. 49 508–49 527, 2022.

[17] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "Llm for soc security: A paradigm shift," *arXiv preprint arXiv:2310.06046*, 2023.

[18] D. Saha, K. Yahyaei, S. Kumar Saha, M. Tehranipoor, and F. Farahmandi, "Empowering hardware security with llm: The development of a vulnerable hardware database," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2024, pp. 233–243.

[19] S. Paria, A. Dasgupta, and S. Bhunia. (2023) Divas: An llm-based end-to-end framework for soc security analysis and policy-based protection.

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

[21] S. Tarek, H. Al Shaikh, S. R. Rajendran, and F. Farahmandi, "Benchmarking of soc-level hardware vulnerabilities: A complete walkthrough," in *2023 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2023, pp. 1–6.

[22] M. Orenes-Vera, A. Manocha, D. Wentzlaff, and M. Martonosi, "Autosva: Democratizing formal verification of rtl module interactions," in *ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 535–540.