IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received 5 May 2024, accepted 7 July 2024, date of publication 12 July 2024, date of current version 31 October 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3427369

PERSPECTIVE

LLM for SoC Security: A Paradigm Shift

**DIPAYAN SAHA, (Member, IEEE), SHAMS TAREK, (Member, IEEE),
KATAYOON YAHYAEI, (Student Member, IEEE), SUJAN KUMAR SAHA, (Member, IEEE),
JINGBO ZHOU, (Member, IEEE), MARK TEHRANIPOOR, (Fellow, IEEE),
AND FARIMAH FARAHMANDI, (Member, IEEE)**
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA

Corresponding author: Dipayan Saha (dsaha@ufl.edu)

This work was supported by the U.S. National Science Foundation (NSF) through Faculty Early Career Development Program (CAREER) Award under Grant 2339971.

ABSTRACT As the ubiquity and complexity of system-on-chip (SoC) designs increase across electronic devices, incorporating security into an SoC design flow poses significant challenges. Existing security solutions are inadequate to effectively verify modern SoC designs due to their limitations in scalability, comprehensiveness, and adaptability. On the other hand, large language models (LLMs) are celebrated for their remarkable success in language understanding, advanced reasoning, and program synthesis tasks. Recognizing an opportunity, our research explores leveraging the emergent capabilities of generative pre-trained transformers (GPTs) to address the existing gaps in SoC security, aiming for a more efficient, scalable, and adaptable methodology. By integrating LLMs into the SoC security verification paradigm, we open a new frontier of possibilities and challenges to ensure the security of increasingly complex SoCs. This paper offers an in-depth analysis of existing works, presents practical case studies, and demonstrates comprehensive experiments. We also present the achievements, prospects, and challenges of employing LLM in different SoC security verification tasks.

INDEX TERMS Hardware security, SoC security verification, hardware vulnerability, large language model.

I. INTRODUCTION

The recent rise of large language models (LLMs) has profoundly impacted the field of natural language processing (NLP), ushering in a new era of capabilities and applications. As the size and complexity of these models increase, they consistently improve in performance and efficiency on numerous NLP tasks. Specifically, their mastery is evident in fields such as text generation [1], summarization [2], machine translation [3], classification [4], sentiment analysis [5], and question answering [6], to name a few. Beyond their efficacy in such linguistic tasks, LLMs are increasingly showcasing incredible proficiency in complex reasoning tasks. This encompasses arithmetic reasoning [7], commonsense, symbolic, and logical deliberations [8], analogical reasoning [9], and even multimodal reasoning [10]. Such emergent abilities [11], more pronounced in larger models such as GPT-3 [12], GPT-4 [13], PaLM [14], etc., provide a captivating insight into the unforeseen potential of scaled-up language models. In addition, because of zero-shot and few-shot learning capabilities, these models could be applied in a wide range of applications: healthcare [15], legal professions [16], creative works [17], and robotics [10]. The remarkable success of these models has catalyzed the development of fine-tuned domain-specific LLMs such as Med-PaLM [18], BloombergGPT [19], AugGPT [20], BioBERT [21], SciBERT [22], etc. For example, Code-centric LLMs [23], [24], with their deep understanding of various code programming languages, are being deployed to assist code generation.

The widespread presence of system-on-chip (SoC) in modern computing systems emphasizes its critical importance. SoCs are now integrated into diverse devices, including smartphones, tablets, IoT devices, and autonomous vehicles, showcasing their significance in the technology landscape. With such an increase in their use, security has become an increasing concern as SoCs collect, analyze, and store users' personal information. Multiple intellectual property

The associate editor coordinating the review of this manuscript and approving it for publication was Derek Abbott.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

(IP) cores with unique functionality and security challenges come together to make an SoC. The extensive functionality along with complex interactions among the IPs, leaves SoCs susceptible to a plethora of security vulnerabilities. From these vulnerabilities, adversaries can exploit information leakage [25], [26], side-channel leakage [27], [28], [29], access control violations [30], etc. The situation is further complicated when considering third-party IPs, which are particularly prone to issues such as hardware Trojans [31]. These issues highlight the importance of thorough security verification in system design. This rigorous and time-intensive process is at odds with the escalating demand for producing billions of computing devices and the corresponding pressure to reduce time-to-market [32]. The tension between these opposing factors makes effective functionality and security verification increasingly difficult, potentially leading to costly spin-offs if issues are discovered post-production. Unfortunately, the existing SoC security solutions [30], [33], [34], [35], [36], [37] are not scalable for handling the increasing complexity and diversity of modern hardware designs, adaptable to new designs and rapidly evolving threat landscape, and comprehensive in addressing hardware vulnerabilities [26], [38].

Given the complexity and diversity of SoC security issues and the proven capabilities of LLM in language understanding, coding, and advanced reasoning activities, the idea of integrating LLM into the SoC security paradigm appears promising. Such an interaction has the potential not only to address existing challenges in SoC security but also to pioneer innovative solutions for the future with the help of the emerging capabilities of LLMs. Moreover, the hardware security community has recently begun to explore the potential of LLMs for SoC security [39], [40], [41], [42], [43], [44], [45]. These efforts, which target specific individual challenges within hardware security, highlight the promise of LLMs in this domain. Nevertheless, the amount of existing research in this domain is inadequate and the real potential of LLMs in different SoC security tasks is untapped. This is the first of its kind work that addresses this research gap, thoroughly investigating the potential of LLMs in SoC security verification.

Figure 1 presents a comprehensive illustration of the potential applications of LLM in SoC security, as addressed in this work. The potential of LLM, combined with the proper selection of learning paradigm, the finesse of prompt engineering, and the rigor of fidelity checks, holds the promise of redefining security tasks across domains. Within this context, we explore the following four different security tasks.

1) **Vulnerability Insertion**: We show how adeptly LLM can introduce potential vulnerability and weakness into register-transfer level (RTL) design following natural language description through the guidance of a well-crafted prompt.
2) **Security Assessment**: Through security assessment, we harness the prowess of LLMs to critically evaluate the security landscape of hardware designs to identify vulnerabilities, weaknesses, and threats through LLM. We also examine the ability of LLM to pinpoint simple coding issues that can turn into security bugs.
3) **Security Verification**: In this scenario, we use LLM to verify if the design meets specific security rules or policies. Furthermore, we check the proficiency of LLM in calculating security metrics, understanding security properties, and generating functional testbenches to identify weaknesses.
4) **Countermeasure Development**: In this scenario, we analyze how effectively LLM can mitigate existing vulnerabilities embedded in the design.

In each of the outlined scenarios, we provide a comprehensive demonstration of executing the tasks using LLMs, with an emphasis on the strategic use of prompt engineering. In addition, by conducting extensive evaluations, we investigate the proficiency of specific LLMs—particularly GPT-3.5 and GPT-4—in undertaking these four critical security tasks.

This is the first of its kind to thoroughly investigate the potential of LLMs in different SoC security-related tasks. Our exhaustive survey of existing LLMs and related security works not only informs readers of the current state-of-the-art but also explains the evolutionary trajectory of these models, aiding in understanding their capabilities and potential applications in the above-mentioned tasks. Our exhaustive discussions and empirical findings not only note the successes of LLMs in the SoC security landscape thus far, but also pinpoint the prospects and prevailing challenges of employing LLMs in SoC security.

In the remainder of this paper, Section II narrates the preliminaries on SoC security. Later, Section III describes the interaction between LLM and SoC security by providing a comprehensive survey of existing LLMs and different related aspects. Afterward, investigations of different LLM capabilities in SoC security-related tasks are discussed in Section IV. Next, Section V examines the accomplishments, difficulties, and potential developments in applying LLM to tasks associated with SoC security. Finally, Section VI concludes the paper.

II. SOC SECURITY

Contemporary SoCs are progressively reaching higher levels of advancement and intricacy. However, this increased complexity also opens up concealed vulnerabilities that attackers can take advantage of. Any bug inside the SoC should be caught and fixed at the earliest stage of the design flow; otherwise, it will be 10 times more expensive than the previous stage. There are rules and guidelines to address the problem of functional bugs in the design-time [46]. The RTL design is the first stage of the design flow in modern ICs. Therefore, detecting and mitigating all the possible

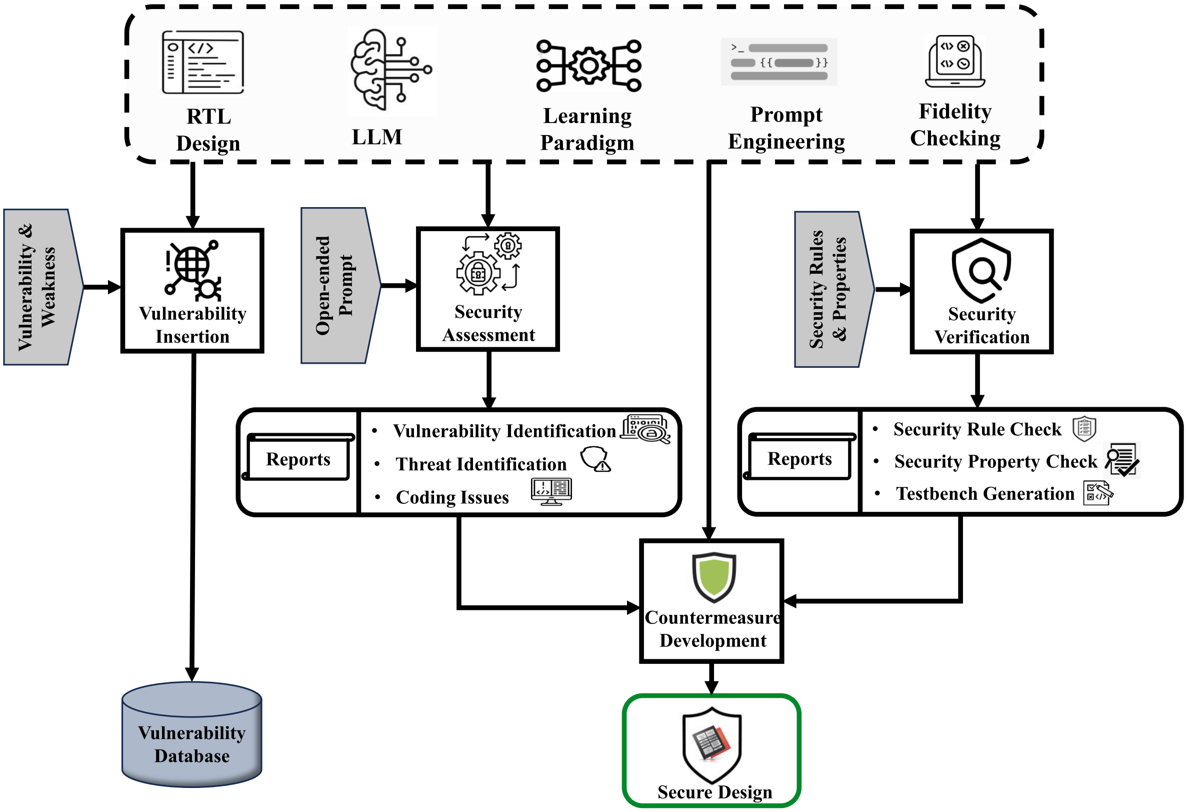


FIGURE 1. Potential applications of LLM in SoC security.

vulnerabilities at this stage will significantly decrease the security verification time, effort, and cost.

1) SECURITY VULNERABILITIES

Hardware security vulnerabilities can emerge at various points in the design process and can be exploited by attackers. These vulnerabilities may result from designers' errors, where the implications of design choices are not fully comprehended, leading to unintended security gaps. Furthermore, computer-aided design (CAD) tools can inadvertently introduce vulnerabilities during the transition from abstract models to specific implementations [27]. Malicious actors within the design process or third-party IP providers can also intentionally insert vulnerabilities, creating backdoors for unauthorized system access [27]. Furthermore, test and debug infrastructures, intended to improve post-silicon diagnostics, can unintentionally offer means to compromise system confidentiality and integrity [26].

Figure 2 illustrates potential vulnerability points within a standard SoC. Many hardware vulnerabilities are preventable through proper hardware description language (HDL) code structures and design practices. The common weakness enumeration (CWE) [47] by MITRE is a key initiative that categorizes hardware vulnerabilities by examining the reasons for their inclusion, their insertion timing, and location within the hardware design [47]. Although insightful, CWE does not provide benchmarks for validating vulnerable designs. Other resources like the Trust-Hub property database [48], Hack@Dac [49], and the HOST Microelectronics Challenge [50] contribute to vulnerability identification, but often lack open-source access or sufficient documentation. The research in [51] advances this field by offering a comprehensive SoC vulnerability benchmark database, which supports the verification of various designs. Despite these developments, the constant emergence of new vulnerabilities necessitates ongoing enhancement of security measures in hardware design.

2) CHALLENGES IN SECURITY VERIFICATION

The verification process in chip design is a major bottleneck, taking up over 70% of resources and time [52]. Recently, a noticeable increase has also been observed in the instances where the verification process has contributed more than 80% of the total duration of the project [32]. Ensuring secure and compliant SoCs for diverse applications becomes even more challenging, necessitating the identification and addressing of security vulnerabilities in pre-silicon stages. These challenges include the globalized nature of the development lifecycle, complex interactions among hardware, software, and firmware layers leading to unforeseen threats,

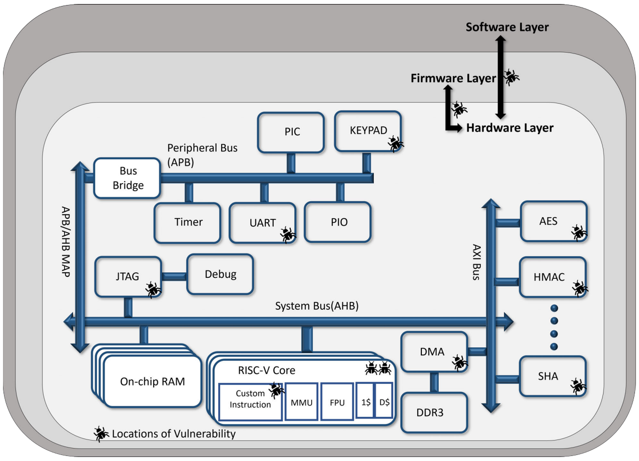


FIGURE 2. **A typical SoC with probable locations of hardware bugs.**

increased design complexity that limits verification coverage, and the lack of standardized benchmarks for comparing emerging verification techniques. Currently, security verification and validation suffer from limited success due to inadequate prioritization of security in design, lack of suitable threat models and vulnerability databases, and reliance on mostly manual and ad-hoc security analysis and mitigation methods. The state-of-the-art verification methods for security focus on assertion-based security property verification [30], information flow tracking [34], fuzz testing [37], concolic testing [53], penetration testing [54], and ML-based hardware verification [55]. The following discussion describes recent trends in hardware design verification approaches.

a: ASSERTION-BASED SECURITY PROPERTY VERIFICATION:

Assertion-based security property verification is a vital method for accelerating the hardware design validation process. It involves incorporating security-related logical statements, or property-based assertions, into the design of a system to formally define security requirements and constraints during development and testing [30], [56], [57]. This technique addresses a key challenge in hardware validation: the limited observability of designs during testing. Observability deals with the capacity to monitor different states within the design. By integrating these properties, we can enhance the observability of design components, enabling the detection of anomalous behavior during simulation. Moreover, assertions based on these properties can identify errors during simulation, significantly reducing the time and resources required for verification. Two widely used assertion specification languages for this purpose are SystemVerilog assertions (SVA) [58] and property specification language (PSL) [59], typically applied at the RTL abstraction level of the design. Automated assertion generation techniques have gained popularity to streamline the process of generating assertions and reduce manual effort [60], [61], [62]. However, a key challenge with these methods is ensuring the functional accuracy and coverage of the generated assertions. Additionally, these generated assertions operate at the bit level, which can noticeably extend simulation runtimes.

b: INFORMATION FLOW TRACKING

Information flow tracking (IFT) within hardware verification is a method utilized for monitoring and regulating the transfer of data or signals within a hardware design or system. Its primary application lies in the domain of security and safety verification, ensuring that sensitive or critical information is managed appropriately and does not unintentionally leak or lead to undesired behavior. The most commonly employed IFT techniques include SecVerilog [63], Sapper [64], Caisson [65], and VeriCoqIFT [66]. Although these approaches excel at detecting hardware bugs associated with information leakage, they are limited by certain problems, such as the need to learn a new language, the requirement for manual annotation, and the inability to distinguish between implicit and explicit information flows. The RTLLIFT approach, as detailed in [34], addresses these challenges by operating directly within existing HDLs and by enabling the differentiation between implicit and explicit flows. Nevertheless, this approach demonstrates limitations in terms of performance when dealing with complex designs, compelling designers to make trade-offs between precision and computational complexity. In addition, these approaches are not comprehensive enough to detect most of the security vulnerabilities.

c: FUZZING

It is a popular testing method in software and has recently attracted a lot of attention in the hardware security verification domain. “Fuzz testing” or “fuzzing” denotes a method involving randomized testing of software programs to identify irregularities and weaknesses [67]. Fuzzing typically involves an automated or semi-automated process designed to check a wide range of predefined (instrumented) scenarios involving invalid inputs, to trigger any existing vulnerabilities within a program. In the hardware domain, fuzzing has been introduced primarily as a solution to address the scalability challenges associated with formal verification methods [68]. Fuzzing can be classified into black-box fuzzing, gray-box fuzzing, and white-box fuzzing according to the information available during the verification phase. Black-box fuzzing relies on the design specifications and is effective for designs with limited information about internal signals [69]. White-box fuzzing is used when design information is completely available [70], and gray-box fuzzing is a hybrid framework to use the best of both white-box and black-box fuzzing techniques [71]. However, these solutions suffer from several drawbacks, such as limited vulnerability coverage, low accuracy due to limited visibility, and poorly defined coverage metrics [37].

d: PENETRATION TESTING

Penetration testing involves actively simulating potential attacks to assess the security of a hardware system or device, aiming to discover vulnerabilities and weaknesses [72]. Its primary goal is to evaluate how well hardware components and systems can withstand potential threats and unauthorized accesses. Similar to fuzzing, penetration testing can adopt black-box, white-box, or gray-box approaches based on the specific threat and available resources [73]. This method comprises various stages, ranging from assessing hardware design to exploiting a specific vulnerability that needs resolution. In contrast to random test patterns, penetration testing relies on precise information about security properties, vulnerabilities, and established threat models. However, applying penetration testing in the hardware domain poses greater challenges compared to the software domain because hardware vulnerabilities are more diverse, requiring different strategies for different vulnerabilities in each targeted penetration testing scenario [35].

e: CONCOLIC TESTING

Concolic testing, as an automated test vector generation approach, combines concrete execution with symbolic execution (concolic) to analyze and validate the behavior of a system [74], [75]. This method blends the execution of a program or hardware design, involving actual (concrete) input values, with abstract symbolic representations of input values. The primary objective of Concolic testing is to systematically explore different execution paths within a program or hardware design. By incorporating symbolic inputs, it can simultaneously investigate multiple paths, including those that might be challenging to reach using traditional testing methods. Concolic testing can be resource intensive, particularly when applied to complex and sizable hardware designs or software programs. In recent times, Concolic testing has found application in hardware security verification, serving purposes such as the detection of hardware Trojans [75], the identification of bugs within the CPU core of SoC [76], and firmware validation [77]. Unfortunately, these methods either only work for a certain part of the SoC due to scalability issues or are limited to detecting a few hardware vulnerabilities.

f: ML-BASED VERIFICATION

Machine learning (ML) is an emerging tool that has recently gained significant attention within the field of hardware verification. ML techniques find applications in diverse verification processes, including the generation of challenging test cases that are hard to achieve and the validation of test results to enhance coverage. Current research trends in this area include creating constraint-random test vectors through supervised and reinforcement learning [78], fine-tuning decision-making procedures for SAT solvers [55], the identification of hardware Trojans [79], and in-depth debugging of system failure analysis [80]. While the prospects are undoubtedly promising, the integration of ML into hardware verification is not without its set of challenges. These hurdles come from the following factors.

- Design Dependency: A critical overlooked issue is the design dependency of ML-based solutions. Due to the lack of a vast and rigorous dataset that involves all corner cases and types of designs, these solutions are often tailored to specific designs and lack adaptability. This limitation hinders their applicability across diverse hardware designs, making them less versatile and effective in a broader context
- Data Management: There exists a scarcity of datasets in the hardware security domain that encompass all potential scenarios and corner cases. This data scarcity can lead to the underperformance of ML models that cannot fully optimize the verification process. The lack of a benchmark is also a critical challenge for the proper evaluation of ML-driven security verification methods.
- Scalability and Efficiency: ML-based solutions are resource-intensive, raising concerns about scalability and efficiency. The time required to train and validate models can be significant, impacting the verification timeline.
- Feature Selection and Objective Function Design: There exists a difficulty in identifying and selecting the most relevant features and designing appropriate objective functions for security verification tasks.

In summary, existing verification methods require significant manual labor for security verification, and also suffer from the problems of adaptability and scalability. A notable issue lies in the scarcity of reliable databases for the development of effective techniques and the proper evaluation of performance. LLMs have the potential to introduce creative solutions to address these prevailing challenges in hardware security verification. With proper prompt engineering, LLMs can prove to be highly useful for identifying and mitigating vulnerabilities in complex hardware designs that can reduce a lot of manual effort. Furthermore, the inferential capabilities of LLMs can be harnessed to construct comprehensive databases in the domain of hardware security, which can help solve the problem of lack of data and adaptability in existing approaches. Incorporation of LLMs can enhance the precision, efficiency, and adaptability of hardware security verification, marking a significant step toward overcoming existing limitations.

III. LLM IN SOC SECURITY VALIDATION

This section ventures into the intersection of LLMs and SoC security. At first, we narrate three basic concepts, namely learning paradigms, model architectures, and control parameters, that are necessary to understand how LLMs can be incorporated into SoC security validation. After setting this foundation, we perform a thorough survey of the established and specialized LLMs to offer insights into their learning settings, architectures, capabilities, and current state. Later,

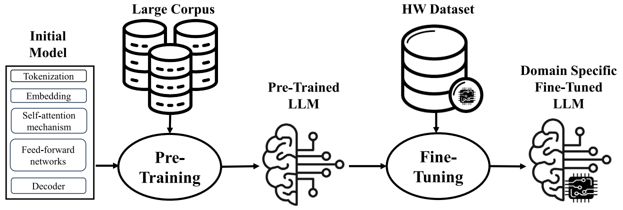


FIGURE 3. Pre-training and fine-tuning approached for hardware design.

we explore the art of prompt engineering, highlighting its significance in refining user-model interactions for security validation. Afterward, we describe the importance of fidelity checking in LLM-based solutions. Concluding the section, we turn our attention to the existing works adopting LLM for software and hardware security solutions.

A. PRELIMINARIES

1) LEARNING PARADIGMS IN LLM

As these models evolve, various learning paradigms have been developed to optimize their performance, each with its own set of advantages and challenges. Table 1 summarizes these methods in terms of data requirement, cost, and complexity, and also discusses their role in the context of SoC security. Here, we discuss these learning paradigms and also make our observations in the context of SoC security.

a: PRE-TRAINING

Pre-training is crucial for LLMs, as it trains the model on a vast text corpus to predict the next word, thereby understanding language structures and semantics. While pre-training equips the model with a broad understanding of language, it does not necessarily make the model an expert in specific domains. Unfortunately, very few existing pre-trained LLMs used HDL codes in their training corpus which narrows the scope of performing security tasks with LLMs. Pre-training LLMs on security-focused HDL codes could enhance their hardware security knowledge, but this is resource-intensive and costly due to the lack of extensive HDL code databases. Additionally, since pre-training relies on historical data, LLMs may not recognize or respond effectively to the most recent security threats that it has not been trained on.

b: FINE-TUNING (FT)

Pre-trained LLMs have a comprehensive grasp of language, yet they often lack the specialized knowledge for specific domains, necessitating fine-tuning. This process, which can involve supervised fine-tuning (SFT), instruction tuning, and reinforcement learning with human feedback (RLHF), refines the model's parameters to improve its performance on specialized tasks [81]. Fine-tuning has proven effective for models such as InstructGPT and ChatGPT, improving accuracy and reducing biases [82], [83]. While less costly than pre-training, fine-tuning still demands substantial resources.

For SoC security, fine-tuning can significantly enhance a model's capabilities in tasks like vulnerability detection and mitigation. However, the process is costly, and models face a ''knowledge cut-off'', becoming outdated as new threats emerge post-update. This issue requires continuous fine-tuning to keep up with the evolving security landscape, as exemplified by the growing list of hardware CWEs maintained by The MITRE Corporation [47]. Furthermore, the scarcity of large, task-specific datasets and the lack of standardized benchmarks in SoC security pose additional challenges to the effective fine-tuning and evaluation of these models. The concepts of pre-training and fine-tuning in the context of hardware design are illustrated in Figure 3.

c: IN-CONTEXT LEARNING (ICL)

In-context learning (ICL), as highlighted in GPT-3 [12], is one of the game-changing capabilities of LLM. ICL includes zero-shot, one-shot, and few-shot learning. Zero-shot learning involves the model performing tasks without prior specific examples. One-shot learning gives the model a single example to learn a new task, and few-shot learning provides a few examples for the model to understand and perform the task. In this way, ICL allows GPT models to adapt to new tasks and bypasses the need for traditional fine-tuning by generating responses based on the instructions supplemented with or without examples. ICL offers significant advantages, such as expanding capabilities to new tasks with few examples and reducing the need for extensive computational resources, enhancing learning efficiency. In SoC security, ICL offers a unique blend of adaptability and specificity. By leveraging the ability of the model to understand and respond based on the provided context, it can offer solutions tailored to specific security challenges. This is particularly beneficial for dynamic security analysis, where the context can vary based on the design, the threat landscape, or the specific security protocols in place. However, one of the primary challenges is the ability of the model to maintain a long-term context. In SoC security, scenarios often span complicated designs and complex threat landscapes, requiring a deep and prolonged understanding of the context. If the model struggles to retain or comprehend this extended context, it might offer solutions that are fragmented or lack depth. This limitation could prevent the model from effectively handling ongoing security scenarios, especially those that require a holistic understanding of the system, its vulnerabilities, and potential mitigation strategies.

d: RETRIEVAL-AUGMENTED IN-CONTEXT LEARNING (RA-ICL)

Retrieval-augmented in-context learning (RA-ICL) enhances GPT models by integrating external knowledge sources to provide up-to-date information, addressing the limitations of the last training update of the models [84], [85]. RA-ICL employs a knowledge retriever to source current data and a knowledge generator to incorporate this information into responses. This approach enriches response generation

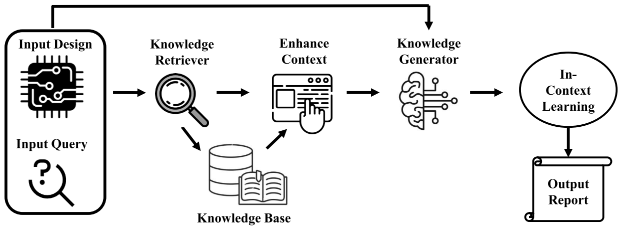


FIGURE 4. Concept of retrieval-augmented in-context learning.

TABLE 1. Prospects of SoC security tasks for different learning settings of GPT.

Learning Setting	Data & Cost	Complexity of task	Prospect of SoC Security
Pre-Training	Very High	High	Not feasible due to high cost and lack of data
SFT	High	Medium	Good for all tasks Limited training data
RLHF	High	Very High	Good for vulnerability insertion, detection & mitigation
ICL	Low	Low	Decent at all tasks. Lack of long term context
RA-ICL	Medium	High	Decent at vulnerability detection & mitigation.

but might also increase computational complexity and costs. In SoC security, RA-ICL can allow GPT models to pull relevant security information from external databases, enhancing the comprehensiveness and relevance of security analysis. In a scenario described in Section III-A1b, RA-ICL can solve the problem of ''knowledge cut-offs'' in the task of detecting and mitigating hardware vulnerability. For example, the knowledge retriever component actively can scan and retrieve new CWE data, ensuring that the responses of the model are informed and current. The The process of RA-ICL in shown in Figure 4.

2) MODEL ARCHITECTURE

Based on the model architecture, we discuss three major categories of existing LLMs: decoder-only, encoder-only, and encoder-decoder. A detailed discussion of their structure, working principles, training objectives, functions, and role in the context of hardware design and SoC security is given below.

a: DECODER-ONLY

Decoder-only LLMs have established impressive benchmarks in numerous NLP tasks, especially in the generation of free-form text. In a decoder-only model, a sequence is fed into the model, which then directly predicts the next token or word in the sequence. It operates autoregressively, using its generated tokens as context for subsequent predictions. It has two variants: causal decoder and prefix decoder. In a standard decoder (causal decoder), the unidirectional attention masking ensures that a token attends only to previous tokens and itself. Prefix decoders permit bidirectional attention over prefix tokens while maintaining unidirectional attention to generated tokens. These models excel in tasks like dialog and story generation that require deep input understanding and coherent output generation. Decoder-only architectures, such as the GPT series [12], [13], have gained popularity due to their parameter efficiency, simplicity, generalization, and versatility.

TABLE 2. Performance of different model architectures across various security tasks. The "Best" column indicates the model type that is most suited for the task, offering optimal performance. The "Decent" column lists model types that can perform the task reasonably well but might not be the optimal choice. The "Poor" column indicates model types that are least suited for the task and might not deliver satisfactory results.

Security Task	Performance		
	Best	Decent	Poor
Vulnerability Insertion	Decoder-only	Encoder-decoder	Encoder-only
Security Verification	Encoder-only	Encoder-decoder Decoder-only	-
Security Assessment	Encoder-only	Encoder-decoder Decoder-only	-
Security Policy & Property Generation	Decoder-only	Encoder-decoder	Encoder-only
Testbench Generation	Decoder-only	Encoder-decoder	Encoder-only
Vulnerability Mitigation	Encoder-decoder	Decoder-only	Encoder-only

Decoder-only models, known for their strength in unconditional generation tasks, shine in areas of SoC security that are predominantly generative. They are tailored for tasks like vulnerability insertion, security policy and property generation, and testbench generation, where the model needs to produce new content based on a given prompt or context. For tasks demanding a deeper understanding before generation, like vulnerability mitigation, they can still offer decent performance but might not be the primary choice.

b: ENCODER-ONLY MODEL

Encoder-only models process input sequences and output a fixed-size context for each token or the entire sequence. These models are adept at distilling information from input sequences into fixed representations, making them suitable for tasks like classification where the aim is to derive a condensed understanding from the input. These models are referred to as ''encoder-only'' because they prioritize encoding input sequences into meaningful embeddings. The ''decoding'' they do is not about generating novel sequences (as with autoregressive models), but rather about producing specific outputs from the learned embeddings, such as masked token predictions during pretraining. BERT [86] developed by Google and its variants: RoBERTa [87], DistilBERT [88], etc are popular examples.

Encoder-only models, with their inherent design to understand and represent data, align well with tasks that require profound analysis. In the SoC security landscape, they are best suited for tasks like security verification and assessment, which demand an in-depth comprehension of the given data without extensive generation. However, when the task requires subsequent generative actions based on the understood context, encoder-only models might not be the ideal choice.

c: ENCODER-DECODER MODEL

The most well-known implementation of the encoder-decoder architecture is the transformer [89]. This model is a two-part architecture. The encoder processes the input sequence and compresses it into a context or an intermediate representation. The encoder component transforms input tokens into vectors using embeddings and positional encodings, then applies multihead self-attention and feedforward networks. The decoder, starting similarly, incorporates masked self-attention and cross-attention with the output of the encoder, ensuring alignment and preventing future word prediction. These models such as BART [2], T5 [90], and UL2 [91] shine in tasks where there is a direct and complex transformation between inputs and outputs. The encoder-decoder architecture, renowned for its ability in natural language understanding tasks, exhibits versatility in SoC security. Its two-stage process of encoding the input data and then decoding it to produce an output makes it suitable for tasks that require both comprehension and generation. This model is particularly adept at vulnerability mitigation, where understanding the context (encoder) and generating a solution (decoder) are both crucial. However, while it is also a good fit for tasks like vulnerability insertion, security verification, and assessment, it may not always be the optimal choice when the task leans heavily toward comprehension or generation.

The potential roles of different model architectures in SoC security tasks are summarized in Table 2.

3) CONTROL PARAMETERS

One of the benefits of LLM or GPT over other traditional deep learning approaches is that the nature of the generated output can be controlled through several parameters [92]. Temperature is an example parameter that controls the randomness of predictions in the output sequence. When an LLM generates text, it predicts the next word based on a probability distribution over all possible words in its vocabulary. This distribution is derived from the model's training, where it has learned the likelihood of each word following a given sequence of words. The temperature parameter adjusts output randomness by modifying this probability distribution. A high temperature makes the probability distribution uniform and leads to more diverse output tokens. While a low temperature makes the results more deterministic and leads to potentially less creative output tokens. When the goal is to generate alternative versions of a given code segment, a high temperature is needed for diverse and innovative solutions. On the other hand, a lower temperature is needed to generate code following a specific pattern or protocol to ensure that the resulting code adheres to the desired structure. The hardware security community needs to do a comprehensive investigation of how these parameters should be selected for a particular security task in order to better enhance SoC security.

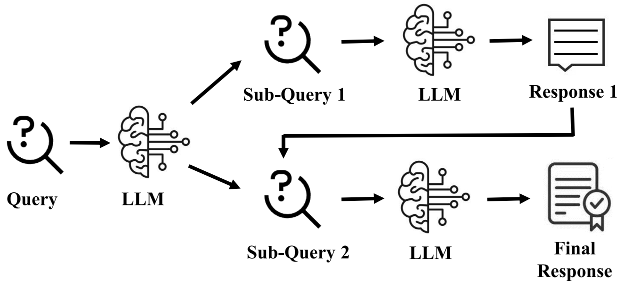


FIGURE 5. Least-to-most prompting methodology.

B. LLM FOR CODING TASK

LLMs have shown impressive capabilities in coding tasks, bridging the gap between natural language processing and coding. From code generation, defect detection, and auto-documentation, to assisting in debugging and even predicting potential software vulnerabilities, LLMs have transformed traditional coding paradigms. Various fine-tuned LLMs specially dedicated to coding tasks have been released by fine-tuning pre-trained models. Table 3 lists all such code LLMs developed in recent years. From the table, it is obvious that the LLMs have predominantly been developed with a focus on mainstream programming languages like Python. This emphasis on Python and similar languages is understandable given their widespread use in software development and data science. However, this has inadvertently led to a gap in the landscape of LLMs specifically fine-tuned for HDL such as Verilog and VHDL. In the context of SoC security, the lack of comprehensive LLMs fine-tuned for Verilog suggests an opportunity to enhance their capabilities to handle HDLs, capitalizing on their underlying coding knowledge. Adapting LLMs for HDLs could effectively help with the security concerns incurred in SoC designs.

C. PROMPT ENGINEERING

Prompt engineering [114] refers to the process of crafting natural language prompts that guide an LLM during inference. It has been observed that prompting techniques can significantly influence the behavior of LLM. This has led to the emergence of several prompting techniques [115], [116], [117], [118], [119], [120], [121], [122], [123], [124], [125], [126], [127], [128], [129]. We categorize the prompting methods for coding and reasoning tasks into the following categories.

- Task Decomposition: Task decomposition refers to the breaking down of a larger or more complex problem into smaller, more manageable subtasks or components. This allows for the easier tackling of each smaller task, which when combined, solves the original problem. The least-to-most prompting [115] and the subsequent prompting [116] are two of such

TABLE 3. A comprehensive list of LLMs specifically developed for coding tasks.PT, FT, and NL stand for pre-training, fine-tuning and natural language, respectively.

Code LLM	Model Architecture	# Params	Training	Base Model	FT for HDL?	NL + Code	Open/ Closed	Task Paradigms
AlphaCode [23]	Encoder-Decoder	1B	PT + FT	-	No	Code	Closed	Code generation
CodeBERT [93]	Encoder-only	125M	PT	RoBERTa -base	No	NL + Code	Open	Code search and code-to-documentation generation
CodeGeex [94]	Decoder-only	13B	PT + FT + FS	-	No	NL + Code	Open	Pre-trained, fine-tuned & few-shot prompted model for code generation, translation & explanation, respectively
CodeGen-Mono [95]	Decoder-only	350M-16.1B	PT + FT	CodeGen-Multi [95]	No	NL + Code	Open	Code generation
CodeGen-Multi [95]	Decoder-only	350M-16.1B	PT + FT	CodeGen-NL	No	NL + Code	Open	Code generation
CodeGen2.5-Mono [96]	-	7B	PT + FT	-	No	NL + Code	Open	Code generation and in-filling
Code LLama [97]	Decoder-only	7-34B	FT	LLama-2	No	NL + Code	Open	Code generation and understanding
Code LLama-Python [97]	Decoder-only	7-34B	FT	Code LLama	No	NL + Code	Open	Specialized for Python
Code LLama-Instruct [97]	Decoder-only	7-34B	IT	Code LLama	No	NL + Code	Open	Code generation and understanding
CodeT5 [98]	Encoder-Decoder	60M, 220M	PT + FT	T5	No	Code	Open	Code understanding and generation tasks
CodeT5+ [99]	Encoder-Decoder	220M-16B	PT	T5 & CodeGen -Mono	No	Code	Open	Code understanding (retrieval, defect and clone detection) and summarization, generation
Codex [24]	Decoder-only	12B	FT	GPT-3	No	Code	Closed	Code Generation and others
Incoder [100]	Mixture-of-Experts (MoE) [101]	6.7 B	PT	FairSeq	No	NL+ code	Open	Generation, masking and infilling
JuPyT5 [102]	Encoder-Decoder	350 M	PT	BART	No	Code	Closed	Code in-filling
Hardware Phi-1.5B [103]	Decoder-only	1.5B	PT	Phi-1.5 [104]	Yes	NL + Code	Closed	Hardware related text and code (VHDL, Verilog and SystemVerilog) generation
PanGu-Coder [105]	Decoder-only	317M 2.6B	PT	PANGU-	No	Code	Closed	Text-to-code generation
PanGu-Coder2 [106]	Decoder-only	15B	RRTF	-	No	Code	Closed	Code generation
PLBART [107]	Encoder-Decoder	140M	PT +FT	BART	No	NL+ Code	Closed	Summarization, generation, translation and classification
PolyCoder [108]	Decoder-only	2.7B	PT	GPT-2	No	Code	Open	Code generation
RTLCoder [109]	Decoder-only	7B	FT	Mistral & DeepSeek	Yes	NL+ Code	Open	Verilog Code generation
SantaCoder [110]	Decoder-only with FIM & MQA	1.1 B	PT	-	No	Code	Open	Infilling capabilities
StarCoder [111]	Decoder-only with FIM & MQA	15.5 B	PT + FT	StarCoder-Base	Yes	Code	Open	Infilling capabilities
VeriGen [112]	Decoder-only	16B	FT	CodeGen	Yes	Code	Open	Verilog code generation
WizardCoder [113]	Decoder-only with FIM & MQA	15.5B	FT	StarCoder	No	Code	Open	Code generation

methods. The least-to-most prompting [115] (shown in Figure 5) first breaks down a complex problem into a series of simpler sub-problems. After the decomposition, these sub-problems are then solved in sequence. Unlike Least-to-most prompting, in the successive prompting [116] method, the decomposition of the question and the answering stages are interleaved. This means that sub-problem solving does not wait for the decomposition of the whole problem, and decomposed sub-problems are not necessarily solved in a sequence. For example, When a sub-problem is identified, it might be immediately solved before moving on to decomposing the next part of the problem.

- Sequential Reasoning: Sequential reasoning methods [117], [118], [119], [120], with a little difference from task decomposition, follow a chain of logical steps to solve a problem. Chain-of-Thought (CoT) prompting is a popular multi-step reasoning technique that enhances the complex reasoning abilities of LLMs by guiding them through a series of intermediate reasoning steps, often demonstrated via exemplars. Zero-CoT is a variant of CoT that adds phrases like "Let's think step by step", and the LLM is cued to reason sequentially, even without prior examples. However, CoT is limited to the challenges where the model needs to navigate through a diverse range of ideas, concepts, or possibilities to generate thoughtful and varied responses. Tree of thoughts (ToT) is an advanced version of CoT. It generalizes the CoT technique and enables the exploration of different "thoughts" that act as intermediate steps in problem-solving. Other researchers improved the performance of CoT through self-consistency [130] and greater complexity of reasoning [121].
- Self-Evaluation and Refinement: The methods facilitate the LLM to critique, rectify, or refine its own outputs.

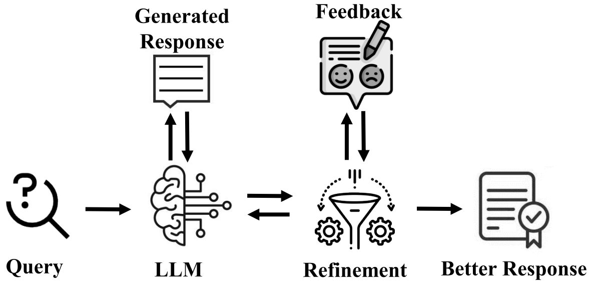


FIGURE 6. Self-refine prompting methodology.

In such methods, the LLM acts as both an executor and an evaluator. Self-Debugging [124] approach prompts the LLM to debug its predicted code. After code execution, the model generates feedback based on code performance, helping toy and correct errors. In Self-Refine [123] (shown in Figure 6), the method generates an initial LLM output which is then refined iteratively based on feedback from the same LLM. Recursive criticism and improvement (RCI) [125] is another technique that starts with a zero-shot prompted LLM output. Then, the LLM is asked to identify problems with that output, and based on the detected issues, the LLM generates an updated solution.

- Multi-LLM Collaboration: Focused on the cooperative use of multiple LLMs, these methods leverage the strengths of different models for better problem-solving. For example, [127] uses two fine-tuned LLMs in tandem, one dedicated to selection and the other to inference, fostering a cohesive reasoning process.

SoC security-related tasks, i.e., security vulnerability injection, detection, and countermeasure development are inherently complicated. The traditional prompting approach - typically by framing queries in a standard question-answer format— does not always give the intended performance. In this work, through meticulously designed case studies and extensive research investigations, we show that the key to unlocking the potential of LLM in SoC security lies in the intricate crafting and calibration of prompts, which can better harness the model's depth of knowledge and reasoning capabilities.

D. FIDELITY CHECKING

When an LLM produces a code, whether in the form of a design file, a testbench, or a SystemVerilog assertion, it is imperative to ensure its accuracy. For example, if the LLM is tasked with introducing a vulnerability into a design, it becomes necessary to verify the presence of that vulnerability within the design. Likewise, when addressing vulnerability mitigation, an evaluation must be conducted to confirm the successful removal of the vulnerability from the design. Additionally, it is prudent to conduct an analysis to determine if the mitigation process has inadvertently introduced new vulnerabilities. Furthermore, assessments should include checks for code quality, ensure syntactical correctness, and confirm that the HDL code of the design accurately represents a valid and functional design.

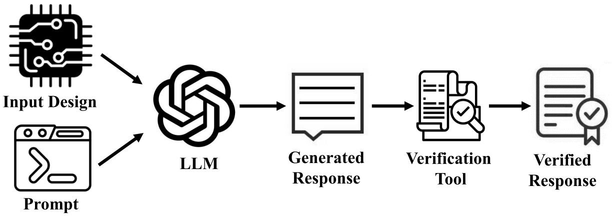


FIGURE 7. Overview of fidelity checking.

Manual code review, although flexible and supporting human-assisted analysis, proves impractical for large and complex designs due to its time and labor intensity. Therefore, it is more suitable as a supplementary method than as the primary verification method. On the other hand, static code analysis automates vulnerability detection and functional correctness checks. It encompasses two key approaches: linting tools and security-aware development tools. Linting tools like Synopsys Spyglass [131] and Cadence JasperGold Superlint [132] rely on coding standards and best practices. However, these tools, primarily designed for non-security purposes, have limitations when used for security analysis. On the contrary, security-aware development tools, such as ARC-FSM [133], are explicitly designed for the detection of security vulnerabilities. Nevertheless, they are relatively new and currently offer limited coverage for vulnerabilities. Formal verification techniques, aided by EDA tools such as JG Superlint, offer automated formal checks alongside lint checks. In addition to the methods mentioned above, additional approaches can also be considered. The flow of fidelity checking is explained in Figure 7.

E. EXISTING WORKS ON LLM IN HARDWARE DESIGN AND SECURITY

Since the advent of LLMs, the software research community has been actively exploring the potential of LLMs in software security. This exploration includes three key areas: the use of LLMs for security assessments in software code [134], [135], [136], [137], [138], [139], [140], the repair of software vulnerabilities [141], [142], [143], and the evaluation of security in software codes generated by LLMs [144], [145], [146], [147]. Research into the skills of LLMs in software is booming, with countless studies highlighting its strengths and possible uses in coding. Yet, when it comes to hardware, there is a noticeable gap. The involvement of LLM in hardware design and analysis has yet to be studied as deeply. In this section, we explore existing work on LLM used in hardware security-related tasks.

1) GENERATION OF HARDWARE DESIGN

The ChipChat study [148], used testbench prompting for their design verification. They chose to employ iVerilog-compatible testbenches, due to the convenience they offered in terms of simulation and testing. This study limited its investigation to only eight benchmarks, assessing four LLMs (GPT-3.5, GPT-4, Bard, HuggingChat) mainly for performance comparisons. Security concerns were overlooked, and the process lacked automation, posing scalability challenges, with manual prompting. Another study called ChipGPT [39] presented an automated design system using LLM to convert natural language specifications into hardware logic designs, smoothly integrating it into the EDA process. They used prompt engineering for HDL to address the limitations of LLM, avoiding manual code editing. If power, performance, or area requirements were not met, they adjusted the prompt and regenerated. However, their framework lacked extensive test cases, with only 8 simple benchmarks, and does not prioritize security.

The authors in [149] exhaustively investigated the capabilities and challenges of employing existing LLMs in AI accelerator design. Based on the investigation, they made three key observations: the need for decoupling hardware functionalities due to LLMs' struggles with lengthy, dependent codes in rare languages like high-level synthesis (HLS); the effectiveness of combining in-context learning with the logical reasoning of powerful, typically closed-sourced LLMs due to limited annotated data; and the importance of augmenting LLM prompts with high-quality, contextually relevant demonstrations. The authors also presented the GPT4AIGChip framework which overcame limitations in hardware design by incorporating a demo-augmented prompt generator and an LLM-friendly hardware template, enhancing design efficiency and accuracy. Furthermore, M. Liu et al. [150] presented a novel approach to adapt LLMs to the specific domain of chip design. This adaptation involved custom tokenizers, domain-adaptive pretraining, supervised fine-tuning with domain-specific instructions, and domain-adapted retrieval models. The effectiveness of the model was demonstrated through applications in engineering assistant chatbots, EDA script generation, and bug summarization and analysis.

Few researchers have attempted to improve the quality of LLM-generated HDL codes. Tsai et al. [151] proposed RTLFixer, a framework designed to automatically fix syntax errors in Verilog code using LLM, addressing the issue that 55% of the LLM-generated Verilog code contains syntax errors. Incorporating retrieval-augmented generation (RAG) and ReAct [129] prompting, the framework achieved a 98.5% success rate in correcting syntax errors. In another effort, Thakur et al. [152] discussed a new approach named Autochip to generate Verilog HDL code using LLM. The authors used LLMs in combination with feedback from Verilog simulations to iteratively generate and refine Verilog code. The system started with a design prompt and used feedback from compilation errors and debugging messages to improve accuracy.

2) BECHMARKS AND DOMAIN SPECIFIC LLMS FOR RTL GENERATION

Recognizing the necessity of proper benchmarks to evaluate the RTL generated from natural language instructions, recently a few benchmarks have been proposed. An open-source benchmark named RTLLM [153] consisted of 30 designs focused on three progressive goals: syntax, functionality, and design quality. Another open-source benchmark VerilogEval [154] featured a comprehensive dataset of 156 problems, covering a wide range of Verilog tasks, and included an automatic testing method for the functional correctness of the code. The study also explored the improvement of LLMs in Verilog code generation through supervised fine-tuning. Authors in [155] showed that through fine-tuning, LLM produced more consistently correct codes. The study also compared the performance of different LLMs in generating Verilog code.

Verigen [112] was the first published open-source domain-specific LLM for RTL generation. This model, fine-tuned based on CodeGen-16B, outperformed the commercial GPT-3.5-turbo, showing a 1.1% overall improvement and a 41% increase in generating syntactically correct Verilog code. Furthermore, RTLCoder [109], another open-source 7-B parameter fine-tuned LLM, was designed for automatic RTL code generation, achieved outperformed GPT-3.5 and Verigen models on the RTLLM benchmark. The model was further optimized to run on a single laptop by quantizing it to 4 bits, reducing its size to 4GB with minimal accuracy loss. Recently, authors in [103] developed another hardware domain-specific model named Hardware Phi-1.5B. The authors did not specify the performance of this model. It should be noted, however, that while these developments are significant, the performance levels of these models are still not sufficient for completely reliable RTL generation, indicating a need for further advancements in this field.

3) SECURITY EVALUATION OF LLM-GENERATED HARDWARE DESIGN

Another study [40] demonstrated that the way prompts are structured in ChatGPT can inadvertently introduce security vulnerabilities into hardware designs. The authors claimed to develop prompt design techniques to ensure secure design generation, but analyzed only a limited number of examples from only 10 CWEs. Essentially, these techniques, which mainly revolve around adding a few sentences to existing prompts, were rudimentary and should not be truly regarded as comprehensive prompting strategies. Further research is needed to evaluate these techniques comprehensively, considering the extensive range of over 100 remaining hardware CWEs.

4) IDENTIFICATION AND REPAIRMENT OF HARDWARE BUG

In the domain of bug repair, the authors in [156] developed a framework that quantitatively evaluates the performance of LLMs in bug repair with a focus on bugs in Verilog HDL code. They prepared design benchmarks from open-source platforms which include ten hardware security bugs. They concluded that an ensemble of LLMs can detect all considered bugs. They achieved a 31.9% overall success rate, demonstrating the ability of LLM to repair unseen code, outperforming previous methods. However, the study acknowledged the need for designer assistance in bug localization and subjective instruction variation. It also lacked exhaustive functional and security evaluations and did not verify if fixed bugs introduce new issues. Furthermore, the method in [156] was limited to only 5 CWEs, leaving numerous security vulnerabilities unexplored, and further examples are required for a complete analysis. RW. Fu et al. [157] addressed the scarcity of domain-specific data in hardware security by compiling a unique dataset of hardware design defects and their remediation steps from open-source projects such as OpenTitan [158], CV6 [159], Ibex [160], etc. This dataset was used to fine-tune medium-sized LLMs (StableLM-Base-Alpha-7b, Falcon-7b, and Llama-2 7b), enabling them to identify bugs in hardware designs effectively. However, in this work, the authors excluded lengthy files that exceed the models' context limits. The framework was not capable of performing analysis on large hardware design.

5) GENERATION OF SECURITY PROPERTY AND ASSERTION

The study in [41] proposed a novel NLP-based security property generator (NSPG) that utilized hardware documentation to automatically mine security property-related sentences. The authors have fine-tuned the general BERT [86] model with sentences from various SoC documentation and evaluated unseen OpenTitan design documents. However, the framework did not create any security properties specific to a particular language (such as SVAs) that can be directly applied to the design. Additionally, the evaluation was conducted on only five previously unseen documents, resulting in a test set with an extremely limited sample size. In another work, Kande et al. [42] created a framework for generating hardware SystemVerilog assertions, employing two manually crafted designs and eight modules from Hack@DAC [49] and OpenTitan [158]. Although the primary experimentation revolved around OpenAI's code-davinci-002, the scalability of the framework was also demonstrated through experiments involving three other LLMs. The study acknowledged the need for coverage of a broader range of CWEs and highlighted that reference assertions and comments were human-created, focusing on one way to capture security properties. Despite an average security assertion generation accuracy of 9.2% during the experimental variations, indicating room for improvement, the study validated the foundational understanding of security assertions within LLMs. This insight suggested that with precise prompts and careful parameter selection, accuracy can be enhanced.

In another study [44], authors presented an iterative methodology using formal property verification (FPV) and GPT-4 to improve the generation of syntactically and semantically correct SVA from RTL modules, and it was integrated with the AutoSVA framework [161] to enhance its capability in generating safety properties. Experiments demonstrated that this enhanced AutoSVA2 framework could identify bugs in complex systems, such as the RISC-V CVA6 Ariane core, which was previously undetected. This framework was not fully automated and required manual effort from the engineer. Furthermore, Paria et al. [45] introduced an end-to-end automated framework to develop security policies with the help of LLM. This framework involved producing appropriate CWEs and SVAs using LLMs from SoC specifications. Nevertheless, a significant portion of the assertions generated by LLMs were syntactically incorrect, posing challenges for automatic integration into designs without manual intervention. Additionally, the majority of CWEs suggested by LLMs displayed a strong bias towards software vulnerabilities, reducing their usability for enhancing hardware security. There is also a study [43] that explored the potential of LLM in formal property writing for functional verification, but lacks diversity in experimentation, focusing on a single design without addressing security concerns.

6) SIDE-CHANNEL ANALYSIS

Kulkarni et al. [162] introduced a language model-based approach for side-channel attacks. Their approach effectively addressed the challenge of finding an efficient leakage model by combining multitasking, multimodal, and deep metric learning. This work offered a new perspective on handling side-channel attacks, particularly in its approach to learning multiple targets and its ability to operate without a predefined leakage model. In another effort, Nair et al. [163] applied GPT-3 models, in a two-phase approach to perform pre-silicon side-channel leakage assessment and generate side-channel resilient hardware design. In the first phase, they fine-tuned Ada, a GPT-3 model, with electrical characteristics of nets to identify leaky and secure nets. Later, in the second phase, Curie, another GPT-3 model, was fine-tuned to generate the secured design. This approach significantly reduced the need for human expertise, accelerated the design process, and ensured functional equivalence in the secured hardware.

Prompt 1.1

$Input Design$ Does this module have any security issues? Describe where and why?

IV. EXPERIMENTS

We conducted an in-depth investigation of the capabilities of LLMs within the SoC security domain. To provide a basic understanding of how LLMs can be used in these tasks and to elucidate the optimal prompt structure, we selected seven exemplary case studies (design generation, vulnerability insertion, vulnerability detection in RISC-V SoCs, security metric calculation, countermeasure development, testbench generation, security assertion understanding) for a detailed analysis. The details of the case studies can be found in https://github.com/sahadipayan/LLM-for-SoC-Security-Case-Studies. These studies not only underline the applicability of LLMs in addressing SoC security challenges but also offer specific prompt guidelines that can significantly elevate the effectiveness of solutions in this domain. One example case study is explained in Section IV-A, and detailed investigations are discussed in Section IV-B.

A. CASE STUDY

To examine vulnerabilities at the SoC level and evaluate their detection and security implications using Gemini Advanced and GPT-4, we utilized two SoCs based on the RISC-V architecture: PULPissimo [164] and CVA6 [165]. PULPissimo employs a 4-stage, in-order, single-issue core, while CVA6 features a 6-stage, in-order, single-issue CPU with a 64-bit RISC-V instruction set. A commonality between them is the integrated debug module (JTAG). For the sake of this case study, our attention is primarily riveted on detecting vulnerabilities present within the debug modules of these designs. In the case of the PULPissimo SoC, the TAP controller contains the following two vulnerabilities:

- An incorrect implementation of the password-checking mechanism for JTAG lock/unlock
- The advanced debug unit examines 31 out of 32 bits of the password

In order to detect the vulnerabilities, we employed two distinct methods: context-free testing and contextual testing.

1) Context-free testing: As its name suggests, it places the LLMs in a position where it is devoid of any explicit context. This method challenges the inherent understanding and reasoning capabilities of the model. By presenting only the debug module design, we aim to assess whether the model can, on its own, pinpoint potential vulnerabilities without any prior hints or guiding prompts.

Prompt 1.2

$Input Design$ *Policy 1. Password-checking logic in debug unit should be correct. Policy 2. It should check all bits of the password. Policy 3. Debug register should not be accessed in halt mode.* Check if any of these policies are violated in the module. If violated, why and where? Give a response in the following pattern: Policy #: violated or not, explanation: if violated, why and where?, line no: violated line

2) Contextual testing: It presents the model with a more structured framework for evaluation. Here, both models are fed with prompts that provide context, aiding its understanding and analysis. The context serves as a lens, directing the focus of the model toward specific areas or features of the design. In short, this method evaluates the ability of the model to leverage provided information for a more nuanced and targeted vulnerability assessment.

For the context-free test, we utilized a basic and straightforward prompt, outlined in Prompt 1.1. As anticipated, the Gemini Advanced and GPT-4 model could not identify any weaknesses within the design, given that we did not include details regarding the context of the vulnerabilities of the design in the prompt. On the other hand, in the contextual testing using Prompt 1.2, the GPT-4 model's response, as showcased in Response to Prompt 1.2, was more discerning. As evident in Response to Prompt 1.2, the model exhibited an enhanced aptitude for vulnerability detection. managed to identify the second vulnerability related to the bit-checking of the password. However, this increased context did not guarantee full vulnerability detection, as the models overlooked an earlier and more intricate vulnerability. This suggests that while contextual clues certainly bolster the analytical capability of the model, they do not make it perfect. The balance between prompt specificity and model inference remains a significant consideration in vulnerability assessment.

Response to Prompt 1.2

Policy 1: Not violated, explanation: The code contains a password-checking logic in the debug unit. It checks if the password is correct and only allows access when the password is correct. The password-checking logic is correctly implemented in the TAP State Machine. Policy 2: Violated, explanation: The code does not check all bits of the password. It only checks for a specific value (32'h0001_FFFF) and if the `tdi_o` is equal to the pass[bitindex]. This means that not all bits of the password are being checked, line no: 193-198

Prompt 1.3

$Input Design$ *Policy 1. Password-checking logic in debug unit should be correct.* Check if this policy is violated in the module. If violated, why and where?

Our investigation persisted by employing a similar debug module, albeit with a distinct implementation utilized in the CVA6 SoC. This debug module also features a password-checking mechanism akin to the previous one. If valid, `pass_check` signal of the design signifies the successful completion of the password-checking process. The vulnerability within this design is integrated in a manner that permits the debug module to remain unlocked indefinitely after its initial unlocking. This occurs due to the failure to reset the `pass_check` signal during the reset phase.

```
PassChkValid: begin
    if(hashValid) begin
        if(hash ==pass_hash) begin
            pass_check =1'b1;
        end else begin
            pass_check =1'b0;
        end
        state_d =Idle;
    end else begin
       state_d =PassChkValid;
    end
end
```

Listing 1: Response from GPT-4 for prompt 1.3.

TABLE 4. Performance of GPT-3.5 in successfully insertion of different hardware vulnerabilities and weaknesses.

Vulnerability	CIA Violation	# Input Designs	# Successful Insertion	Success Rate (%)
V1	Availability	166	154	92.77
V2	Availability	152	143	94.08
V3	Integrity, Availability	251	220	87.64
V4	Integrity Availability	251	161	64.14
V5	Availability	273	241	88.27

We provided the design along with the little context to the models, as shown in Prompt 1.3.

Response to Prompt 1.3 (GPT)

The DMI JTAG module described in your code seems to implement a kind of password checking via the `PassChk` and `PassChkValid` states and the HMAC module, but there may be a potential violation of policy 1 - *"the password checking logic in the debug unit should be correct*. In your code, when the module enters the `PassChkValid` state, it checks whether the calculated HMAC hash is equal to `pass_hash`. If they are equal, it sets `pass_check` to 1 (true), else it sets it to 0 (false). This procedure is secure in itself. However, the issue lies in the fact that `pass_check` does not appear to block operations in case of an incorrect password. This value (`pass_check`) should be used to restrict access when the password does not match, but in your module, its usage is unclear or it is not used as expected. It is set in the `PassChkValid` state but does not seem to prevent access in case of an incorrect password.

The response depicted in Response to Prompt 1.3 initiates an intriguing discussion. While unable to pinpoint the precise location of the vulnerability (within the reset condition), both the GPT-4 and Gemini Advanced models brought to light another significant flaw in the design regarding the password-checking mechanism - a flaw previously unknown to us. The code snippet featured in Listing 1 showcases the GPT-4 model's response, revealing that the password-checking condition invariably leads to an unlocked debug session, regardless of whether the password matches successfully or not. However, with a more comprehensive prompt, complete with instructions on detecting the vulnerability, the models were able to identify the actual flaw in the design.

Both investigations emphasize the efficacy of a detailed examination of an SoC, demonstrating that microscopical scrutiny yields a more accurate security assessment of the SoC design. The main challenge lies in the token constraints inherent to the LLM interface, which require us to present smaller designs or segments of designs as prompts to encapsulate a complete hardware functionality. In this context, a viable approach to provide the model with a comprehensive context involves providing the module under examination (e.g., debug module). Furthermore, it should be noted that GPT-4 and Gemini Advanced can assist in uncovering previously unidentified design flaws that could be challenging to detect using conventional verification methods, significantly when constrained by tight time-to-market requirements.

Response to Prompt 1.3 (Gemini)

The code has a potential bug where incorrect passwords might be incorrectly accepted due to a faulty state transition located within the `PassChkValid` state. This issue arises because the state incorrectly transitions to Idle even if the calculated `hash` (hash) does not match the stored hash (`pass_hash`). This behavior violates the policy requiring accurate password-checking logic. To fix this, you should introduce a new state representing password failure and transition to this state when `hashValid` is true, but the hashes do not match.

B. IN-DEPTH INVESTIGATIONS

The case studies alone are not sufficient to draw definitive conclusions. Consequently, in this section, we undertake a comprehensive investigation into the abilities of GPTs, focusing on four key areas: insertion of security vulnerability, detection of security rule violation, identification of hardware threat, detection of coding issues, and development of countermeasures.

Before discussing these security tasks, it is important to assess the proficiency of GPT-3.5 in generating syntactically correct hardware designs. In our analysis, we examined the 1806 Verilog designs produced by GPT-3.5. A meticulous syntax check revealed that 1580 of these designs were devoid of syntax errors. The remaining designs exhibited minor syntactical oversights, such as extraneous punctuations, omitted parentheses, usage of obsolete constructs, and missing 'begin' blocks. Additionally, there were instances where there was a mix-up between Verilog and SystemVerilog constructs. It should be noted that the syntax issues identified in the designs are relatively minor and can be rectified with ease.

Such errors are often common, even among experienced designers, especially during initial design drafts. Automated tools or linting software can quickly identify and correct these issues, ensuring the final design is both syntactically and semantically correct.

1) INSERTION OF VULNERABILITY

A database that contains vulnerable designs is crucial in the hardware security domain. Such a database can serve as a rich resource for training and testing security tools, enabling them to better recognize and address vulnerabilities in hardware design. However, there is a scarcity of such databases in the community because introducing vulnerability into hardware design is a complicated task. It requires advanced knowledge, effort, and time to create such a database manually. This motivates us to investigate the proficiency of GPT-3.5 in embedding various vulnerabilities and CWEs into hardware designs.

In this context, we selected five distinct security vulnerabilities, each with its own security implications. These vulnerabilities are as follows:

- V1: CWE 835 (Unreachable Exit Condition)
- V2: Unused states not handled through the 'default' statement
- V3: Presence of duplicate encoding State
- V4: Presence of unreachable state
- V5: Presence of static deadlock

For vulnerability insertion, in each case, a specific prompting technique has been applied. Table 4 shows the performance of GPT-3.5 in the successful insertion of these vulnerabilities. With the exception of the unreachable state scenario, GPT-3.5 boasts a success rate exceeding 85% in integrating these security flaws into the designs. The most notable success has been observed in scenarios involving the 'default' statement, while the insertion of unreachable states into finite state machine (FSM) designs posed the greatest challenge. An unreachable state in FSM is defined as a state without any input transitions that has one or more transitions to other states [33]. This definition bears a striking resemblance to the definitions of both dead state and static deadlock, leading to potential ambiguities in the vulnerability insertion process and results. Such ambiguities, in turn, inadvertently introduce other security vulnerabilities during this task, leading to a lower success rate. It is noteworthy that GPT-3.5 attains 88.27% success in creating static deadlock, which is a complicated situation. It indicates that, through proper guidance, LLMs can be used to insert complicated vulnerabilities and weaknesses into hardware designs.

2) DETECTION OF SECURITY RULE VIOLATION

In order to assess the performance of vulnerability detection, we selected the following three security rule violations in FSM designs defined by [33]

- R1: A state with a static deadlock scenario must not exist.

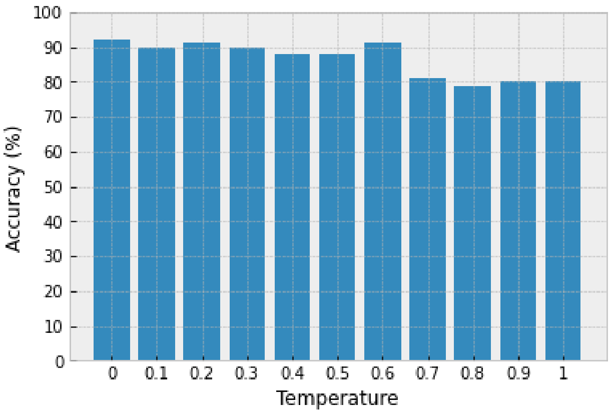


FIGURE 8. Impact of 'temperature' parameter of GPT-3.5 in the detection of "duplicate state encoding".

TABLE 5. Comparison of performance of GPT-3.5 and ARC-FSM in the detection of vulnerabilities embedded into designs.

Rule Violation	Framework	# Input Designs	# Accurate Detection	Accuracy (%)
R1	GPT-3.5	273	216	79.12
	ARC-FSM		242	88.64
R2	GPT-3.5	351	322	91.74
	ARC-FSM		321	99.69
R3	GPT-3.5	209	172	82.30
	ARC-FSM		209	100.00

- R2: Each state must be encoded uniquely.
- R3: When a state transition occurs between two consecutive unprotected states, the Hamming distance (HD) between those should be '1.'

In this task, the input design might either adhere to or violate certain security rules. The primary objective was to determine whether a given rule had been violated. We designed specific prompting strategies to detect any security rule violations. Analogously, distinct prompting methodologies were employed for each rule in this context. To estimate the accuracy of decisions made by GPT-3.5, we cross-referenced its decisions with the established ground truth.

Table 5 presents a comparative analysis of the performance of GPT-3.5 and ARC-FSM in detecting embedded vulnerabilities within design inputs. For each violation, the table lists the number of input designs tested, the count of accurately detected vulnerabilities, and the corresponding accuracy percentage for both frameworks. From the table, it is evident that while GPT-3.5 demonstrates commendable accuracy in detecting vulnerabilities, ARC-FSM consistently exhibits higher or near-perfect accuracy rates across the tested security rules. For instance, in the case of "Presence of static deadlock," GPT-3.5 achieved an accuracy of 79.12%, whereas ARC-FSM reached 88.64%. Similar trends are observed for other security rule violations, underscoring the effectiveness of specialized tools like ARC-FSM in vulnerability detection. However, it is noteworthy that GPT-3.5 still

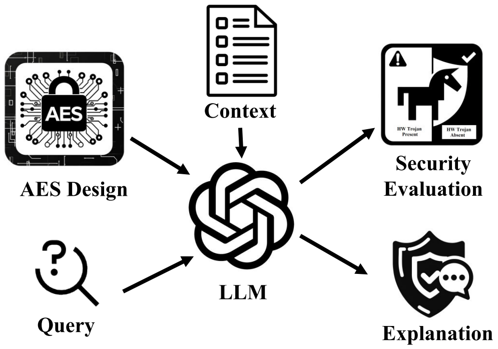


FIGURE 9. Contextual test for hardware Trojan detection.

offers competitive performance, especially considering its general-purpose nature compared to the specialized design of ARC-FSM.

In all of the above-mentioned experiments, we set the temperature of GPT-3.5 very low to keep the model deterministic. But intuitively temperature should have an impact on the performance of vulnerability detection. As we mentioned before the 'temperature' parameter in language models like GPT-3.5 essentially controls the randomness of the model's predictions. A lower temperature steers the model towards more deterministic outputs, while a higher value encourages diversity, potentially leading to more creative but less precise responses. In order to investigate this impact, we vary the temperature of GPT-3 from 0 to 1 with an increase of 0.1 for the task of detecting the violates of 'unique encoding state' in 100 input designs. The impact of the 'temperature' parameter is shown in Figure 8. This suggests that a higher temperature lowers the accuracy of the model in this specific context. However, it would be premature to conclude that a lower temperature is universally ideal for vulnerability detection tasks. We are not advocating for the exclusive use of a lower temperature setting. Instead, we emphasize that the impact of temperature is multifaceted and warrants meticulous investigation to harness the optimum performance of LLMs. The observation emphasizes the importance of carefully tuning the temperature parameter, especially in critical tasks such as vulnerability detection.

3) HARDWARE TROJAN DETECTION

Hardware Trojans present grave risks to electronic systems, with potential consequences ranging from unauthorized access and data breaches to complete system malfunctions. Detecting these Trojans is imperative, especially when considering the security of national critical infrastructures or defense systems. In our study, we investigated the abilities of GPT-3.5 and GPT-4 to identify hardware Trojans within AES designs, utilizing 28 distinct AES designs sourced from the Trust-Hub Trojan Benchmark [166].

To enhance the rigor of our investigation, we first sanitized the RTL code of any overt indications of Trojan presence. This involved renaming Trojan modules and eliminating explicit terms like "Trojan" and "trigger" from the RTL.

TABLE 6. Performance of proposed GPT-3.5 and GPT-4 in hardware trojan detection.

LLM	Test Method	Total # Tests	# Detected	Accuracy (%)
GPT-3.5	Context-Free Test	275	0	0
	Contextual Test	275	45	16.36
GPT-4	Context-Free Test	216	175	81.01
	Contextual Test	108	99	91.67

TABLE 7. Performance of GPT-3.5 in the mitigation of hardware vulnerabilities.

Vulnerability	CIA Violation	# Input Designs	# Successful Mitigation	Success Rate (%)
V3	Integrity, Availability	161	147	91.30
V4	Integrity Availability	160	158	96.43
V5	Availability	168	162	96.43

Our assessment employed two distinct methodologies: the Context-Free Test and the Contextual Test. The former tests the raw capabilities of the models by providing no explicit context about the design. Conversely, the Contextual Test, shown in Figure 9, offers models supplementary information, outlining potential Trojan insertion techniques in AES designs. To further enhance the depth of our analysis, each test setup was executed under varying temperature settings, incrementally adjusted from 0 to 1 at intervals of 0.1. This process ensures the thoroughness of the investigations.

Table 6 provides a comparative analysis of the hardware Trojan detection capabilities of GPT-3.5 and GPT-4. From the result, it is very clear that GPT-3.5 does not have enough knowledge to detect such a security threat. When enhanced knowledge is provided, it attains 16.36% accuracy, which is still low. The length of the context also becomes a problem for GPT-3.5. It is challenging to provide the whole AES design with enough context for a high Trojan detection accuracy in GPT-3.5 due to its token limit which is comparatively shorter than GPT-4. On the other hand, GPT-4 significantly outperforms GPT-3.5 with an impressive detection accuracy of 81.01%, even when no context is provided. The contextual test, which provides the models with additional knowledge, further accentuates the superiority of GPT-4, achieving a 91.67% accuracy rate. These results clearly indicate the advances in GPT-4, highlighting its improved proficiency in hardware Trojan detection over its predecessor. Such superior performance of GPT-4 in both tests indicates a deeper understanding of hardware designs and threats, emphasizing its potential as a valuable tool in hardware security evaluations.

GPT-4 matches other machine learning algorithms in the accuracy of detecting hardware Trojans with fewer steps, utilizing natural language descriptions [167]. Its advanced natural language processing allows for a nuanced

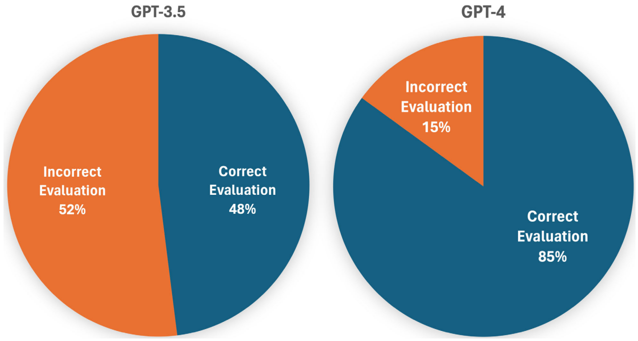


FIGURE 10. Experimental results of GPT-3.5 and GPT-4 in identifying simple coding issues in 75 different hardware designs.

understanding of hardware designs, overcoming design dependencies that challenge traditional ML methods. GPT-4's capability for contextual analysis leads to more thorough Trojan detection and can be further improved with targeted fine-tuning.

4) DETECTION OF CODING ISSUE

In the domain of hardware design, even seemingly minor coding issues can have profound implications. While these issues, such as linting discrepancies, structural anomalies, coding style inconsistencies, and synthesis problems, might not directly manifest as security vulnerabilities or weaknesses, they can act as precursors. Such coding issues can inadvertently introduce or propagate more severe security vulnerabilities in the design. Ensuring the absence of these coding issues is not just about maintaining a clean codebase; it is also about preempting potential security risks and ensuring the robustness and reliability of the hardware. For this experiment, we sourced 75 coding issues and the test designs from the Jasper Superlint reference manual [132]. It is important to note that we carefully sanitized the designs, which means we ensured that there is no explicit indications or markers pointing toward the coding issue.

Figure 10 offers a comparative analysis of GPT-3.5 and GPT-4 in pinpointing these coding issues within hardware designs. Of the 75 different hardware designs evaluated, GPT-3.5 accurately detected coding issues in nearly 48% of the cases. While this indicates a moderate capability of GPT-3.5 in identifying such issues, there is evident potential for enhancement. In contrast, GPT-4 showed proficiency in accurately detecting issues in 85% of the designs, a significant improvement 37% over GPT-3.5. This marked advancement suggests that GPT-4 has a more remarkable ability to discern and highlight coding problems in hardware designs.

5) VULNERABILITY MITIGATION

Vulnerabilities within hardware designs can be particularly detrimental, given their foundational role in many electronic systems. Recognizing this, we have examined the capabilities of GPT-3.5 to rectify these security vulnerabilities in FSM designs. In our experimental setup, we took a directed approach to harness the capabilities of GPT. Instead of leaving the model to identify vulnerabilities without context, we explicitly informed GPT about the specific security vulnerability present within the design. With this knowledge, GPT was then tasked with the challenge of mitigating the identified vulnerability. In this experiment, we have considered three distinct vulnerabilities: duplicate encoding state, presence of unreachable state, and presence of Static Deadlock. Each vulnerability is associated with potential violations of the CIA (Confidentiality, Integrity and Availability) triad, underscoring its criticality in the context of secure hardware design. Table 7 shows the performance of GPT-3.5 in mitigating these vulnerabilities.

For the problem of duplicate encoding state, remarkably, successful mitigation was achieved in 147 of 161 cases, translating to a success rate of 91.30%. This suggests that the mitigation techniques employed are highly effective against this particular vulnerability, ensuring that states in the design are uniquely encoded Thereby, ambiguous state transitions can be prevented. The mitigation technique showed even greater efficacy in the case of the unreachable state, with 158 designs successfully rectified, resulting in a success rate of 96.43%. Unreachable states can indicate design inefficiencies or hidden vulnerabilities, so their effective mitigation is paramount to the overall robustness of the system. Lastly, the presence of static deadlock, which primarily affects availability, was evaluated in 168 designs. Deadlocks can halt system operations, leading to potential system failures. The mitigation techniques employed demonstrated a consistent success rate of 96.43%, with 162 designs successfully addressed. The findings highlight the robustness and effectiveness of the employed mitigation techniques against common hardware vulnerabilities. The consistently high success rates across different vulnerabilities emphasize the scope of employing LLM in vulnerability mitigation to ensure secure and reliable hardware designs.

V. ACHIEVEMENTS, CHALLENGES AND PROSPECTS

A. ACHIEVEMENTS BY LLMS IN SOC SECURITY

The integration of LLMs has been a game-changer in the field of SoC security. These models, particularly GPT-4, have outperformed their predecessors due to their advanced understanding of language and code, as well as their reasoning abilities. This has led to significant breakthroughs in various domains. Some of the achievements of LLMs in this domain are listed below:

1) SUCCESS OF ICL

In-context learning (ICL) in LLMs has revolutionized adaptability, enabling the model to tackle new tasks without traditional fine-tuning, proving particularly beneficial in dynamic security analysis for SoC security.

2) SUCCESS IN AUTOMATION AND VERIFICATION

Initial studies indicate that LLMs can play a vital role in the hardware design landscape by automating processes, bridging the gap between natural language and design specifications, enhancing security policy formulation, exploring diverse applications like formal property writing, and improving design verification efficiency.

3) SUCCESS IN DATABASE CREATION

The use of LLMs, particularly GPT-3.5 and GPT-4, has demonstrated the capability to insert vulnerabilities successfully and effortlessly within hardware designs through detailed prompting with hands-on examples. It reduces the manual effort and time to compile large databases of vulnerable hardware designs.

4) SUCCESS IN VULNERABILITY IDENTIFICATION

GPT-3.5 has demonstrated commendable accuracy in detecting security rule violations in FSM designs, showcasing its applicability even when compared to specialized tools. Moreover, GPT-4 has demonstrated the capability to identify existing vulnerabilities within IP modules, albeit somewhat inconsistently, when given sufficient context.

5) SUCCESS IN SECURITY THREAT IDENTIFICATION

LLMs have begun contributing to the identification of security threats like hardware Trojans and side-channel attacks. GPT-4 has achieved hardware Trojan detection performance comparable to traditional ML algorithms but with a simplified process utilizing natural language descriptions.

6) SUCCESS AS LINTING TOOL

GPT-4 has shown a significant improvement in identifying and addressing coding issues, marking notable progress in the field of automated code analysis.

B. CHALLENGES IN APPLYING LLMS TO SOC SECURITY

While LLMs have achieved many milestones, the journey is not without its challenges. These challenges stem from both technological limitations and the evolving nature of security threats:

1) LACK OF TRAINING DATA AND BENCHMARK

In training LLM for SoC security, a key challenge arises from the limited availability of domain-specific training data and benchmarks, both crucial for refining and validating model performance in specialized security scenarios. The scarcity of hardware design-related data significantly hampers the development of hardware domain-specific open-source LLMs, as highlighted in Section III-B. Moreover, databases specific to hardware security are even rarer in the public domain. Although recent initiatives, as referenced in [51], [112], and [48], provide some progress, they are insufficient. Creating a comprehensive hardware database is a laborious and time-consuming task. However, as discussed in this paper, one effective solution could be utilizing LLMs themselves. Through strategic prompting, LLMs can be guided to generate synthetic hardware designs, providing a rapid method to create extensive hardware security databases by minimizing manual effort. In this process, fidelity checking of the LLM-generated database should be a high priority, as discussed in Section III-D.

2) PROBLEM TO HANDLE LARGE DESIGN

Limited context length and token limitation are key challenges in employing LLMs for SoC security tasks. These limit the ability of LLM to maintain a long-term context and handle extensive information. For instance, a typical SoCs contain 100,000 lines of code. LLMs face difficulty in tackling these larger designs, as they can only process smaller designs or design segments at a time. This poses a challenge to the scalability of the LLM-assisted method. One viable way to use LLMs is to analyze large hardware designs in segments. LLMs can maintain contextual awareness across divided code snippets, ensuring that the integrity and coherence of the design are preserved even when processed in parts. This distinct capability of LLMs allows for comprehensive verification without the computational bottlenecks associated with processing large datasets in one go. Moreover, increasing the context length of LLM is one of the major focuses for both academia and industry. Recent advancements have seen significant increases in token limits. For instance, the recently released Gemini 1.5 Pro has a token limit of 10M [168]. Moreover, OpenAI's GPT-4 series models and Claude-2.1 now support up to 128k and 200k tokens, respectively. This trend suggests that token limitations may no longer pose a significant barrier to analyzing large hardware designs soon.

3) HIGH COST

The high cost of training or deploying LLMs is a significant barrier. LLMs require substantial computational resources for training and operation, leading to considerable expenses. Inferring closed models such as GPT-4 through API is also costly. The high cost associated with GPT models hinders their widespread adoption for large-scale hardware design. However, the introduction of parameter-efficient fine-tuning methods such as Qlora [169], which optimizes model architecture to decrease both hardware requirements and the number of training parameters, is helping to mitigate these challenges. Additionally, open-source LLMs provide a cost-effective alternative, being free to use and modify. As technology progresses and more competitors emerge in the market, the costs associated with LLMs continue to diminish, making them more accessible for a broader range of applications.

4) UNAVAILABILITY OF LLMS

The proprietary designs of GPT models are not open-sourced and restrict open access to their underlying designs and functionalities. This limitation hinders the ability of researchers and developers to thoroughly understand and innovate upon these models, particularly in specialized domains like SoC security. To counteract this, there is a growing call within the academic and industrial communities for more transparent, open-source LLM initiatives that foster broader innovation and allow more profound insights into model mechanisms.

5) UNPREDICTABLE RESPONSE

The non-deterministic nature of LLMs, highlighted by variability across testing iterations, poses a challenge in achieving consistent results. The outputs of LLMs vary significantly across different testing iterations. Developing methods to stabilize the model's output, such as fidelity checking, as discussed in Section III-D, might help mitigate this issue.

6) LACK OF HDL-SPECIALIZED LLM

The current landscape of LLMs is predominantly tailored for mainstream programming languages, leading to a gap in specialized models for HDLs. Encouraging specialized research programs and industry collaborations to develop and train LLMs specifically for HDL applications could address this gap, enhancing the tools available for hardware designers and security experts.

7) DYNAMIC NATURE OF SECURITY

As the domain of SoC security is continuously changing, LLMs might not be up-to-date with the latest developments, potentially impacting their effectiveness in addressing recent security concerns. Continuous learning solutions, where LLMs are regularly updated with the latest data, or adaptive learning frameworks that can dynamically adjust to new information, are potential strategies to keep these models effective and relevant.

8) NEED FOR HIGH-QUALITY PROMPT GENERATOR

Despite advancements in LLM prompting strategies for general tasks, an urgent need remains for automated, high-quality prompt generation tailored specifically to SoC security tasks, ensuring optimized performance and reliability. Such tools would ensure that LLMs operate at peak efficiency and effectiveness, generating outputs that are directly applicable to current security challenges.

9) UNFILTERED TRAINING DATA

The challenge lies in the risk of LLMs generating hardware designs with security vulnerabilities due to being trained on unfiltered data from open-source repositories. This may lead LLMs to generate hardware designs with embedded security vulnerabilities inadvertently. Implementing comprehensive data curation and validation protocols before training can help ensure that only high-quality, secure data is used, thus minimizing the risk of harmful outputs.

10) LACK OF CREATIVITY THROUGH DETAILED PROMPTING

When given explicit examples during the prompting phase, there is a tendency for LLMs to gravitate towards replicating the provided example rather than genuinely crafting a unique solution. This issue could be addressed by developing advanced prompting algorithms that encourage creative problem-solving capabilities within LLMs, possibly through techniques that foster divergent thinking or by integrating mechanisms that challenge the model to explore beyond the given examples.

C. PROSPECTS FOR ENHANCING SOC SECURITY WITH LLM

The future of LLM in ensuring SoC security appears particularly bright. We identify the following prospects of LLM in enhancing SoC security.

1) ADAPTING TO DOMAIN-SPECIFIC EXPERTISE

Fine-tuning can be strategically employed to adapt LLM to specific security tasks, seamlessly bridging the gap between generalized knowledge and domain-specific expertise. Existing LLMs proficient in mainstream programming languages can be fine-tuned for HDLs, enhancing security in SoC designs

2) INCORPORATING RA-ICL FOR CONTINUOUS LEARNING

The adoption of RA-ICL promises enhanced real-time adaptability in SoC security solutions, as it is capable of continuously updating its knowledge base with the most current information, ensuring that responses and solutions are always informed, relevant, and up-to-date.

3) SELECTING OPTIMAL MODEL ARCHITECTURE AND ADVANCED PROMPTING TECHNIQUES

By carefully choosing the right model architecture, greater precision can be achieved in executing specific security tasks. This optimization leads to more effective and efficient security solutions. Furthermore, employing advanced prompting methods like multi-step reasoning and few-shot prompts can significantly enhance the performance of LLMs in SoC security tasks. These techniques allow for more nuanced and accurate responses.

4) BUG DETECTION AND MITIGATION

LLM such as GPT-4 offers potential in uncovering previously unidentified design flaws, especially under tight time-to-market constraints, which might be elusive to traditional verification methods. The high success rates in mitigating security vulnerabilities also open the door to further exploration and utilization of LLMs like GPT-3.5 in automated vulnerability mitigation, potentially revolutionizing the field of hardware security.

5) DATABASE CREATION

LLM can simplify the creation of extensive hardware design datasets. Such a dataset requires time and lots of manual efforts which LLM can reduce. The superior performance of GPT-3.5 in the successful insertion of hardware vulnerabilities and weaknesses into hardware design suggests a promising avenue for utilizing LLMs in creating databases of vulnerable designs, essential for developing security tools

6) AUTOMATED TESTBENCH GENERATION

Through tailored prompts, targeting specific security weaknesses within test cases, the automated generation of testbenches using LLM technology displays the substantial potential to notably improve the speed, efficiency, and adaptability of testbench creation dedicated to security vulnerability detection.

7) SCOPE OF INTEGRATING WITH EDA

The minor syntactical oversights present in the GPT-generated designs highlight an opportunity for integrating automated tools or linting software, or even enhancing GPT models for self-scrutiny, to ensure the generation of error-free, optimized hardware designs

8) OPTIMIZATION OF CONTROL PARAMETER

There is potential to improve the consistency and accuracy of GPT models in detecting vulnerabilities by meticulously tuning parameters such as temperature and employing rigorous prompting strategies.

VI. CONCLUDING REMARKS

The rapid advancements in the SoC domain and their pervasive presence in modern electronics systems have accentuated the urgency for robust and innovative security solutions. As SoCs become integral to many devices, from smartphones to autonomous vehicles, their security challenges become increasingly multifaceted. In parallel, the emergence and evolution of LLMs have revolutionized the field of NLP and even coding and reasoning tasks. With their unparalleled linguistic and reasoning capabilities, these models offer a promising avenue for addressing the sophisticated challenges of SoC security. Recognizing this potential, our research is motivated to dive deep into the confluence of LLMs and SoC security, to harness the strengths of these models to perform SoC security tasks.

Throughout this work, we embarked on a comprehensive exploration of the role LLMs can play in various SoC security tasks. Our extensive survey of existing LLMs and LLM-based hardware security works provided a detailed panorama of their development trajectories, capabilities, and potential applications. Furthermore, we have demonstrated practical case studies and showcased different LLM capabilities in scenarios such as vulnerability insertion, security assessment, security verification, and countermeasure development. We also presented large-scale investigations as a roadmap for future endeavors in this domain. Throughout this work, we identified six achievements, eleven challenges, and eight prospects of LLM in SoC security validation. By bridging the gap between LLM capabilities and SoC security needs, we have laid a robust foundation that researchers and industry professionals can build upon.

The future of LLMs in SoC security looks promising as we look ahead. These advanced systems keep improving and more flexible, opening up many possibilities in SoC security. Although our research has made progress, there is still a lot that we need to explore. We hope that our work guides and inspires more research, encourages new ideas, and helps collaborations in this growing field.

REFERENCES

[1] J. Li, T. Tang, W. Xin Zhao, J.-Y. Nie, and J.-R. Wen, "Pretrained language models for text generation: A survey," 2022, *arXiv:2201.05273*.
[2] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension," 2019, *arXiv:1910.13461*.
[3] W. Jiao, W. Wang, J.-T. Huang, X. Wang, S. Shi, and Z. Tu, "Is ChatGPT a good translator? Yes with GPT-4 as the engine," 2023, *arXiv:2301.08745*.
[4] J. Howard and S. Ruder, "Universal language model fine-tuning for text classification," 2018, *arXiv:1801.06146*.
[5] W. Zhang, Y. Deng, B. Liu, S. Jialin Pan, and L. Bing, "Sentiment analysis in the era of large language models: A reality check," 2023, *arXiv:2305.15005*.
[6] D. Su, Y. Xu, G. I. Winata, P. Xu, H. Kim, Z. Liu, and P. Fung, "Generalizing question answering system with pre-trained language model fine-tuning," in *Proc. 2nd Workshop Mach. Reading Question Answering*. Hong Kong: Association for Computational Linguistics, 2019, pp. 203–211.
[7] S. Imani, L. Du, and H. Shrivastava, "Mathprompter: Mathematical reasoning using large language models," in *Proc. 61st Annu. Meeting Assoc. Comput. Linguistics*, 2023, pp. 1–6.
[8] C. Qin, A. Zhang, Z. Zhang, J. Chen, M. Yasunaga, and D. Yang, "Is ChatGPT a general-purpose natural language processing task solver?" 2023, *arXiv:2302.06476*.
[9] T. Webb, K. J. Holyoak, and H. Lu, "Emergent analogical reasoning in large language models," *Nature Human Behaviour*, vol. 7, no. 9, pp. 1526–1541, Jul. 2023.
[10] D. Driess, F. Xia, M. S. Sajjadi, C. Lynch, A. Chowdhery, B. Ichter, A. Wahid, J. Tompson, Q. Vuong, and T. Yu, "PaLM-E: An embodied multimodal language model," 2023, *arXiv:2303.03378*.
[11] J. Wei, Y. Tay, R. Bommasani, C. Raffel, B. Zoph, S. Borgeaud, D. Yogatama, M. Bosma, D. Zhou, D. Metzler, E. H. Chi, T. Hashimoto, O. Vinyals, P. Liang, J. Dean, and W. Fedus, "Emergent abilities of large language models," 2022, *arXiv:2206.07682*.
[12] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell, "Language models are few-shot learners," in *Proc. NeurIPS*, 2020, pp. 1877–1901.
[13] OpenAI et al., "GPT-4 technical report," 2023, *arXiv:2303.08774*.
[14] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, and S. Gehrmann, "PaLM: Scaling language modeling with pathways," 2022, *arXiv:2204.02311*.
[15] H. Nori, N. King, S. M. McKinney, D. Carignan, and E. Horvitz, "Capabilities of GPT-4 on medical challenge problems," 2023, *arXiv:2303.13375*.
[16] M. Bommarito II and D. M. Katz, "GPT takes the bar exam," 2022, *arXiv:2212.14402*.
[17] P. Mirowski, K. W. Mathewson, J. Pittman, and R. Evans, "Co-writing screenplays and theatre scripts with language models: Evaluation by industry professionals," in *Proc. CHI Conf. Human Factors Comput. Syst.*, 2023, pp. 1–34.
[18] K. Singhal, S. Azizi, T. Tu, S. S. Mahdavi, J. Wei, H. W. Chung, N. Scales, A. Tanwani, H. Cole-Lewis, and S. Pfohl, "Large language models encode clinical knowledge," *Nature*, vol. 620, pp. 1–9, Jul. 2023.
[19] S. Wu, O. Irsoy, S. Lu, V. Dabravolski, M. Dredze, S. Gehrmann, P. Kambadur, D. Rosenberg, and G. Mann, "BloombergGPT: A large language model for finance," 2023, *arXiv:2303.17564*.

[20] H. Dai, Z. Liu, W. Liao, X. Huang, Y. Cao, Z. Wu, L. Zhao, S. Xu, W. Liu, N. Liu, S. Li, D. Zhu, H. Cai, L. Sun, Q. Li, D. Shen, T. Liu, and X. Li, "AugGPT: Leveraging ChatGPT for text data augmentation," 2023, *arXiv:2302.13007*.

[21] J. Lee, W. Yoon, S. Kim, D. Kim, S. Kim, C. H. So, and J. Kang, "BioBERT: A pre-trained biomedical language representation model for biomedical text mining," *Bioinformatics*, vol. 36, no. 4, pp. 1234–1240, Feb. 2020.

[22] I. Beltagy, K. Lo, and A. Cohan, "SciBERT: A pretrained language model for scientific text," 2019, *arXiv:1903.10676*.

[23] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, and A. Dal Lago, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022.

[24] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, and G. Brockman, "Evaluating large language models trained on code," 2021, *arXiv:2107.03374*.

[25] A. Nahiyan, K. Xiao, K. Yang, Y. Jin, D. Forte, and M. Tehranipoor, "AVFSM: A framework for identifying and mitigating vulnerabilities in FSMs," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[26] 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 *Proc. 22nd Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2017, pp. 617–622.

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

[28] J. Lee, M. Tehranipoor, and J. Plusquellic, "A low-cost solution for protecting IPs against scan-based side-channel attacks," in *Proc. 24th IEEE VLSI Test Symp.*, 2006, p. 6.

[29] N. Pundir, J. Park, F. Farahmandi, and M. Tehranipoor, "Power side-channel leakage assessment framework at register-transfer level," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 30, no. 9, pp. 1207–1218, Sep. 2022.

[30] N. Farzana, F. Rahman, M. Tehranipoor, and F. Farahmandi, "SoC security verification using property checking," in *Proc. IEEE Int. Test Conf. (ITC)*, Nov. 2019, pp. 1–10.

[31] M. Tehranipoor and F. Koushanfar, "A survey of hardware trojan taxonomy and detection," *IEEE Design Test Comput.*, vol. 27, no. 1, pp. 10–25, Aug. 2010.

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

[33] R. Kibria, F. Farahmandi, and M. Tehranipoor, "Arc-FSM-G: Automatic security rule checking for finite state machine at the netlist abstraction," in *Proc. IEEE Int. Test Conf. (ITC)*, Oct. 2023, pp. 320–329.

[34] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 1691–1696.

[35] 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 *Proc. 28th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2023, pp. 579–584.

[36] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "RTL-ConTest: Concolic testing on RTL for detecting security vulnerabilities," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 466–477, Mar. 2022.

[37] 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," *Cryptol. ePrint Arch.*, Mar. 2022. [Online]. Available: https://eprint.iacr.org/2022/394.pdf

[38] M. M. M. Rahman, S. Tarek, K. Z. Azar, M. Tehranipoor, and F. Farahmandi, "Efficient SoC security monitoring: Quality attributes and potential solutions," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 41, no. 4, pp. 26–34, Jul. 2023.

[39] K. Chang, Y. Wang, H. Ren, M. Wang, S. Liang, Y. Han, H. Li, and X. Li, "ChipGPT: How far are we from natural language hardware design," 2023, *arXiv:2305.14019*.

[40] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, "How hardened is your hardware? Guiding ChatGPT to generate secure hardware resistant to CWEs," in *Cyber Security, Cryptology, and Machine Learning*. Cham, Switzerland: Springer Nature, 2023, pp. 320–336.

[41] X. Meng, A. Srivastava, A. Arunachalam, A. Ray, P. H. Silva, R. Psiakis, Y. Makris, and K. Basu, "Unlocking hardware security assurance: The potential of LLMs," 2023, *arXiv:2308.11042*.

[42] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "LLM-assisted generation of hardware assertions," 2023, *arXiv:2306.14027*.

[43] P. Srikumar, "Fast and wrong: The case for formally specifying hardware with LLMS," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, Mar. 2023, pp. 1–13.

[44] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "Using LLMs to facilitate formal verification of RTL," 2023, *arXiv:2309.09437*.

[45] S. Paria, A. Dasgupta, and S. Bhunia, "DIVAS: An LLM-based end-to-end framework for SoC security analysis and policy-based protection," 2023, *arXiv:2308.06932*.

[46] L. Bening, *Principles of Verifiable RTL Design*. New York, NY, USA: Springer, 2000.

[47] *Common Weakness Enumeration*. Accessed: Oct. 20, 2024. [Online]. Available: https://trust-hub.org/#/data/Security-Properties-Rules

[48] *Hub.org*. Accessed: Oct. 20, 2024. [Online]. Available: https://trust-hub.org/#/data/Security-Properties-Rules

[49] *HACK@DAC'23*. Accessed: Oct. 20, 2024. [Online]. Available: https://hackatevent.org/hackdac23/

[50] (2023). *Host 2023 Microelectronics Security Competition: Soc Security Track*. [Online]. Available: http://www.hostsymposium.org/host2023/doc/SoC-Security-Track2023.pdf

[51] S. Tarek, H. A. Shaikh, S. R. Rajendran, and F. Farahmandi, "Benchmarking of SoC-level hardware vulnerabilities: A complete walkthrough," in *Proc. IEEE Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, Jun. 2023, pp. 1–6.

[52] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2015, pp. 1–6.

[53] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu, "RTL-ConTest: Concolic testing on RTL for detecting security vulnerabilities," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 41, no. 3, pp. 466–477, Mar. 2022.

[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 *Proc. 28th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2023, pp. 579–584.

[55] F. Hutter, D. Babic, H. H. Hoos, and A. J. Hu, "Boosting verification by automatic tuning of decision procedures," in *Proc. Formal Methods Comput. Aided Design*, 2007, pp. 27–34.

[56] H. Witharana, Y. Lyu, S. Charles, and P. Mishra, "A survey on assertion-based hardware verification," *ACM Comput. Surveys*, vol. 54, no. 11s, pp. 1–33, Sep. 2022.

[57] 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 *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Apr. 2023, pp. 1–6.

[58] *IEEE Standard for Systemverilog–Unified Hardware Design, Specification, and Verification Language*, IEEE Standard 1800-2012, 2013, pp. 1–1315.

[59] *IEEE Standard for Property Specification Language (PSL)*, Standard IEEE Std 1850-2010, 2010, pp. 1–182.

[60] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 6, pp. 952–965, Jun. 2013.

[61] L. A. Breslow and D. W. Aha, "Simplifying decision trees: A survey," *Knowl. Eng. Rev.*, vol. 12, no. 1, pp. 1–40, Jan. 1997.

[62] B. Ahmed, F. Rahman, N. Hooten, F. Farahmandi, and M. Tehranipoor, "AutoMap: Automated mapping of security properties between different levels of abstraction in design flow," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2021, pp. 1–9.

[63] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 503–516, Mar. 2015.

[64] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "Sapper: A language for hardware-level security policy enforcement," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.* New York, NY, USA: Association for Computing Machinery, 2014, pp. 97–112.

[65] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: A hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.

[66] M.-M. Bidmeshki and Y. Makris, "Toward automatic proof generation for information flow policies in third-party hardware IP," in *Proc. IEEE Int. Symp. Hardw. Oriented Secur. Trust (HOST)*, May 2015, pp. 163–168.

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

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

[69] J. De Ruiter and E. Poll, "Protocol state fuzzing of TLS implementations," in *Proc. 24th USENIX Conf. Secur. Symp.* Berkeley, CA, USA: USENIX Association, 2015, pp. 193–206.

[70] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 474–484.

[71] M. M. Hossain, F. Farahmandi, M. Tehranipoor, and F. Rahman, "BOFT: Exploitable buffer overflow detection by information flow tracking," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Feb. 2021, pp. 1126–1129.

[72] S. Shah and B. M. Mehtre, "An overview of vulnerability assessment and penetration testing techniques," *J. Comput. Virol. Hacking Techn.*, vol. 11, no. 1, pp. 27–49, Feb. 2015.

[73] M. Fischer, F. Langer, J. Mono, C. Nasenberg, and N. Albartus, "Hardware penetration testing knocks your SoCs off," *IEEE Des. Test. IEEE Des. Test. Comput.*, vol. 38, no. 1, pp. 14–21, Feb. 2021.

[74] K. Sen, "Concolic testing," in *Proc. twenty-second IEEE/ACM Int. Conf. Automated Softw. Eng.* New York, NY, USA: Association for Computing Machinery, Nov. 2007, pp. 571–572.

[75] R. Zhang, C. Deutschbein, P. Huang, and C. Sturton, "End-to-end automated exploit generation for validating the security of processor designs," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2018, pp. 815–827.

[76] L. Shen, D. Mu, G. Cao, M. Qin, J. Blackstone, and R. Kastner, "Symbolic execution based test-patterns generation algorithm for hardware trojan detection," *Comput. Secur.*, vol. 78, pp. 267–280, Sep. 2018.

[77] N. Corteggiani, G. Camurati, and A. Francillon, "Inception: System-wide security testing of real-world embedded systems software," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 309–326.

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

[79] Z. Huang, Q. Wang, Y. Chen, and X. Jiang, "A survey on machine learning against hardware trojan attacks: Recent advances and challenges," *IEEE Access*, vol. 8, pp. 10796–10826, 2020.

[80] P. Gaur, S. S. Rout, and S. Deb, "Efficient hardware verification using machine learning approach," in *Proc. IEEE Int. Symp. Smart Electron. Syst. (iSES) (Formerly iNiS)*, Dec. 2019, pp. 168–171.

[81] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–13.

[82] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, and J. Schulman, "Training language models to follow instructions with human feedback," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 27730–27744.

[83] OpenAI. (2022). *Introducing ChatGPT*. [Online]. Available: https://openai.com/blog/chatgpt.

[84] O. Ram, Y. Levine, I. Dalmedigos, D. Muhlgay, A. Shashua, K. Leyton-Brown, and Y. Shoham, "In-context retrieval-augmented language models," 2023, *arXiv:2302.00083*.

[85] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-T. Yih, and T. Rocktäschel, "Retrieval-augmented generation for knowledge-intensive NLP tasks," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 9459–9474.

[86] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[87] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A robustly optimized BERT pretraining approach," 2019, *arXiv:1907.11692*.

[88] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a distilled version of BERT: Smaller, faster, cheaper and lighter," 2019, *arXiv:1910.01108*.

[89] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–13.

[90] C. Raffel N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, Wei Li, and Peter J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *J. Mach. Learn. Res.*, vol. 21, no. 1, pp. 5485–5551, 2020.

[91] Y. Tay, M. Dehghani, V. Q. Tran, X. Garcia, J. Wei, X. Wang, H. W. Chung, D. Bahri, T. Schuster, and S. Zheng, "UL2: Unifying language learning paradigms," in *Proc. 11th Int. Conf. Learn. Represent.*, 2022, pp. 1–13.

[92] *Create Chat Completion*. Accessed: Oct. 20, 2024. [Online]. Available: https://platform.openai.com/docs/api-reference/chat/create

[93] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou, "CodeBERT: A pre-trained model for programming and natural languages," 2020, *arXiv:2002.08155*.

[94] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, Z. Wang, L. Shen, A. Wang, Y. Li, T. Su, Z. Yang, and J. Tang, "CodeGeeX: A pre-trained model for code generation with multilingual benchmarking on HumanEval-X," 2023, *arXiv:2303.17568*.

[95] E. Nijkamp, B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, S. Savarese, and C. Xiong, "CodeGen: An open large language model for code with multi-turn program synthesis," 2022, *arXiv:2203.13474*.

[96] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, "CodeGen2: Lessons for training LLMs on programming and natural languages," in *Proc. ICLR*, 2023, pp. 1–11.

[97] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, and J. Rapin, "Code llama: Open foundation models for code," 2023, *arXiv:2308.12950*.

[98] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware unified pre-trained encoder–decoder models for code understanding and generation," 2021, *arXiv:2109.00859*.

[99] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "CodeT5+: Open code large language models for code understanding and generation," 2023, *arXiv:2305.07922*.

[100] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, W.-T. Yih, L. Zettlemoyer, and M. Lewis, "InCoder: A generative model for code infilling and synthesis," 2022, *arXiv:2204.05999*.

[101] N. Shazeer, A. Mirhoseini, K. Maziarz, A. Davis, Q. Le, G. Hinton, and J. Dean, "Outrageously large neural networks: The sparsely-gated mixture-of-experts layer," 2017, *arXiv:1701.06538*.

[102] S. Chandel, C. B. Clement, G. Serrato, and N. Sundaresan, "Training and evaluating a Jupyter notebook data science assistant," 2022, *arXiv:2201.12901*.

[103] W. Fu, S. Li, Y. Zhao, H. Ma, R. Dutta, X. Zhang, K. Yang, Y. Jin, and X. Guo, "Hardware phi-1.5B: A large language model encodes hardware domain specific knowledge," in *Proc. 29th Asia South Pacific Design Autom. Conf. (ASP-DAC)*, Jan. 2024, pp. 349–354.

[104] Y. Li, S. Bubeck, R. Eldan, A. Del Giorno, S. Gunasekar, and Y. Tat Lee, "Textbooks are all you need II: Phi-1.5 technical report," 2023, *arXiv:2309.05463*.

[105] F. Christopoulou, G. Lampouras, M. Gritta, G. Zhang, Y. Guo, Z. Li, Q. Zhang, M. Xiao, B. Shen, and L. Li, "PanGu-coder: Program synthesis with function-level language modeling," 2022, *arXiv:2207.11280*.

[106] B. Shen, J. Zhang, T. Chen, D. Zan, B. Geng, A. Fu, M. Zeng, A. Yu, J. Ji, J. Zhao, Y. Guo, and Q. Wang, "PanGu-coder2: Boosting large language models for code with ranking feedback," 2023, *arXiv:2307.14936*.

[107] W. Uddin Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation," 2021, *arXiv:2103.06333*.

[108] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program.*, 2022, pp. 1–10.

[109] S. Liu, W. Fang, Y. Lu, Q. Zhang, H. Zhang, and Z. Xie, "RTLCoder: Outperforming GPT-3.5 in design RTL generation with our open-source dataset and lightweight solution," 2023, *arXiv:2312.08617*.
[110] L. B. Allal, R. Li, D. Kocetkov, C. Mou, C. Akiki, C. M. Ferrandis, N. Muennighoff, M. Mishra, A. Gu, and M. Dey, "SantaCoder: Don't reach for the stars!" 2023, *arXiv:2301.03988*.
[111] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, and J. Chim, "StarCoder: May the source be with you!" 2023, *arXiv:2305.06161*.
[112] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, "VeriGen: A large language model for verilog code generation," *ACM Trans. Design Autom. Electron. Syst.*, vol. 29, no. 3, pp. 1–31, May 2024.
[113] Z. Luo, C. Xu, P. Zhao, Q. Sun, X. Geng, W. Hu, C. Tao, J. Ma, Q. Lin, and D. Jiang, "WizardCoder: Empowering code large language models with evol-instruct," 2023, *arXiv:2306.08568*.
[114] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Proc. Extended Abstr. CHI Conf. Human Factors Comput. Syst.*, May 2021, pp. 1–7.
[115] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi, "Least-to-Most prompting enables complex reasoning in large language models," 2022, *arXiv:2205.10625*.
[116] D. Dua, S. Gupta, S. Singh, and M. Gardner, "Successive prompting for decomposing complex questions," 2022, *arXiv:2212.04092*.
[117] T. Wu, M. Terry, and C. J. Cai, "AI chains: Transparent and controllable human-AI interaction by chaining large language model prompts," in *Proc. CHI Conf. Human factors Comput. Syst.*, 2022, pp. 1–22.
[118] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 24824–24837.
[119] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa, "Large language models are zero-shot reasoners," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 22199–22213.
[120] S. Yao, D. Yu, J. Zhao, I. Shafran, T. L. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," 2023, *arXiv:2305.10601*.
[121] Y. Fu, H. Peng, A. Sabharwal, P. Clark, and T. Khot, "Complexity-based prompting for multi-step reasoning," 2022, *arXiv:2210.00720*.
[122] Z. Gou, Z. Shao, Y. Gong, Y. Shen, Y. Yang, N. Duan, and W. Chen, "CRITIC: Large language models can self-correct with tool-interactive critiquing," 2023, *arXiv:2305.11738*.
[123] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, S. Gupta, B. Prasad Majumder, K. Hermann, S. Welleck, A. Yazdanbakhsh, and P. Clark, "Self-refine: Iterative refinement with self-feedback," 2023, *arXiv:2303.17651*.
[124] X. Chen, M. Lin, N. Schärli, and D. Zhou, "Teaching large language models to self-debug," 2023, *arXiv:2304.05128*.
[125] G. Kim, P. Baldi, and S. McAleer, "Language models can solve computer tasks," 2023, *arXiv:2303.17491*.
[126] M. Nye, A. Johan Andreassen, G. Gur-Ari, H. Michalewski, J. Austin, D. Bieber, D. Dohan, A. Lewkowycz, M. Bosma, D. Luan, C. Sutton, and A. Odena, "Show your work: Scratchpads for intermediate computation with language models," 2021, *arXiv:2112.00114*.
[127] A. Creswell and M. Shanahan, "Faithful reasoning using large language models," 2022, *arXiv:2208.14271*.
[128] O. Rubin, J. Herzig, and J. Berant, "Learning to retrieve prompts for in-context learning," 2021, *arXiv:2112.08633*.
[129] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing reasoning and acting in language models," 2022, *arXiv:2210.03629*.
[130] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou, "Self-consistency improves chain of thought reasoning in language models," 2022, *arXiv:2203.11171*.
[131] Synopsys. *Synopsys Spyglass Lint Tool*. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-lint.html
[132] Cadence. *Jasper Superlint App*. Accessed: Oct. 20, 2024. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-verification-platform/jaspergold-superlint-app.html
[133] Rasheed Kibria. (2023). *ARC-FSM: Static Security Rule Checker*. Accessed: Oct. 20, 2024. [Online]. Available: https://gitlab.com/RasheedKibria/STATICSECURITYRULECHECKER
[134] J. Shi, Z. Yang, B. Xu, H. J. Kang, and D. Lo, "Compressing pre-trained models of code into 3 MB," in *Proc. 37th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2022, pp. 1–12.
[135] Y. Chen, Z. Ding, L. Alowain, X. Chen, and D. Wagner, "DiverseVul: A new vulnerable source code dataset for deep learning based vulnerability detection," 2023, *arXiv:2304.00409*.
[136] T. Chen, L. Li, L. Zhu, Z. Li, X. Liu, G. Liang, Q. Wang, and T. Xie, "VulLibGen: Generating names of vulnerability-affected packages via a large language model," 2023, *arXiv:2308.04662*.
[137] M. Amine Ferrag, A. Battah, N. Tihanyi, R. Jain, D. Maimut, F. Alwahedi, T. Lestable, N. Singh Thandi, A. Mechri, M. Debbah, and L. C. Cordeiro, "SecureFalcon: Are we there yet in automated software vulnerability detection with LLMs?" 2023, *arXiv:2307.06616*.
[138] C. Tony, M. Mutas, N. E. D. Ferreyra, and R. Scandariato, "LLMSecEval: A dataset of natural language prompts for security evaluations," 2023, *arXiv:2303.09384*.
[139] H. Tu, Z. Zhou, H. Jiang, I. N. B. Yusuf, Y. Li, and L. Jiang, "Isolating compiler bugs by generating effective witness programs with large language models," 2023, *arXiv:2307.00593*.
[140] A. Happe and J. Cito, "Getting pwn'd by AI: Penetration testing with large language models," 2023, *arXiv:2308.00121*.
[141] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2023, pp. 2339–2356.
[142] Y. Wu, N. Jiang, H. Viet Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," 2023, *arXiv:2305.18607*.
[143] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," 2023, *arXiv:2304.03816*.
[144] B. Yetiştiren, I. Özsoy, and M. Ayerdem, "Evaluating the code quality of AI-assisted code generation tools: An empirical study on GitHub copilot, Amazon CodeWhisperer, and ChatGPT," 2023, *arXiv:2304.10778*.
[145] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara, "How secure is code generated by ChatGPT?" 2023, *arXiv:2304.09655*.
[146] H. Hajipour, K. Hassler, T. Holz, L. Schönherr, and M. Fritz, "CodeLMSec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models," 2023, *arXiv:2302.04012*.
[147] L. Niu, S. Mirza, Z. Maradni, and C. Pöpper, "$*CodexLeaks*$: Privacy leaks from code generation language models in $*GitHub*$ copilot," in *Proc. 32nd USENIX Secur. Symp. (USENIX Security)*, 2023, pp. 2133–2150.
[148] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-chat: Challenges and opportunities in conversational hardware design," 2023, *arXiv:2305.13243*.
[149] Y. Fu, Y. Zhang, Z. Yu, S. Li, Z. Ye, C. Li, C. Wan, and Y. C. Lin, "GPT4AIGChip: Towards next-generation AI accelerator design automation via large language models," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Oct. 2023, pp. 1–9.
[150] M. Liu, T.-D. Ene, R. Kirby, C. Cheng, N. Pinckney, R. Liang, J. Alben, H. Anand, S. Banerjee, and I. Bayraktaroglu, "ChipNeMo: Domain-adapted LLMs for chip design," 2023, *arXiv:2311.00176*.
[151] Y.-D. Tsai, M. Liu, and H. Ren, "RTLFixer: Automatically fixing RTL syntax errors with large language models," 2023, *arXiv:2311.16543*.
[152] S. Thakur, J. Blocklove, H. Pearce, B. Tan, S. Garg, and R. Karri, "AutoChip: Automating HDL generation using LLM feedback," 2023, *arXiv:2311.04887*.
[153] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "RTLLM: An open-source benchmark for design RTL generation with large language model," 2023, *arXiv:2308.05345*.
[154] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: Evaluating large language models for verilog code generation," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Aug. 2023, pp. 1–8.
[155] S. Thakur, B. Ahmad, Z. Fan, H. Pearce, B. Tan, R. Karri, B. Dolan-Gavitt, and S. Garg, "Benchmarking large language models for automated verilog RTL code generation," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Apr. 2023, pp. 1–6.
[156] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing hardware security bugs with large language models," 2023, *arXiv:2302.01215*.
[157] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "LLM4SecHW: Leveraging domain-specific large language model for hardware debugging," in *Proc. Asian Hardw. Oriented Secur. Trust (AsianHOST)*, 2023, pp. 1–6.
[158] R. Hansen and D. Rizzo. (2019). *Opentitan*. [Online]. Available: https://opentitan.org/
[159] (2019). *Cva6*. [Online]. Available: https://github.com/openhwgroup/cva6

[160] (2019). *Lowrisc(ibex).* [Online]. Available: https://github.com/lowRISC/ibex

[161] M. Orenes-Vera, A. Manocha, D. Wentzlaff, and M. Martonosi, "AutoSVA: Democratizing formal verification of RTL module interactions," in *Proc. 58th ACM/IEEE Design Autom. Conf. (DAC)*, Dec. 2021, pp. 535–540.

[162] P. Kulkarni, V. Verneuil, S. Picek, and L. Batina, "Order vs. chaos: A language model approach for side-channel attacks," *Cryptol. ePrint Arch.*, 2023.

[163] M. Nair, R. Sadhukhan, H. Pearce, D. Mukhopadhyay, and R. Karri, "Netlist whisperer: Ai and NLP fight circuit leakage!" in *Proc. Workshop Attacks Solutions Hardw. Secur.*, 2023, pp. 83–92.

[164] P. D. Schiavone, D. Rossi, A. Pullini, A. Di Mauro, F. Conti, and L. Benini, "Quentin: An ultra-low-power PULPissimo SoC in 22 nm FDX," in *Proc. IEEE SOI-3D-Subthreshold Microelectron. Technol. Unified Conf. (S3S)*, Oct. 2018, pp. 1–3.

[165] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-GHz 64-bit RISC-V core in 22-nm FDSOI technology," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 27, no. 11, pp. 2629–2640, Nov. 2019.

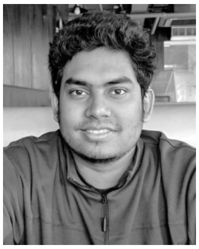
[166] H. Salmani, M. Tehranipoor, S. Sutikno, and F. Wijitrisnanto, "Trust-hub trojan benchmark for hardware trojan detection model creation using machine learning," 2022.

[167] F. Wijitrisnanto, S. Sutikno, and S. D. Putra, "Efficient machine learning model for hardware trojan detection on register transfer level," in *Proc. 4th Int. Conf. Signal Process. Inf. Secur. (ICSPIS)*, Nov. 2021, pp. 37–40.

[168] M. Reid, N. Savinov, D. Teplyashin, D. Lepikhin, T. Lillicrap, J.-b. Alayrac, R. Soricut, A. Lazaridou, O. Firat, and J. Schrittwieser, "Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context," 2024, *arXiv:2403.05530*.

[169] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLORA: Efficient finetuning of quantized LLMs," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 36, 2024, pp. 1–28.

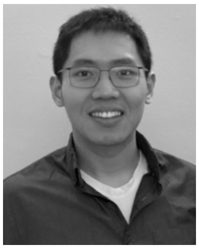
DIPAYAN SAHA (Member, IEEE) received the B.Sc. degree in electrical and electronics engineering from Bangladesh University of Engineering and Technology (BUET), in 2018, and the M.Sc. degree in electrical and computer engineering from the University of Florida, in 2024, where he is currently pursuing the Ph.D. degree with the Department of ECE. His research works have been published in multiple journals and conference proceedings. His research interests include artificial intelligence (AI)-based hardware security solutions. Specifically, he is working on the development of deep learning approaches for RTL code analysis and pre-silicon side-channel leakage assessment.



SHAMS TAREK (Member, IEEE) received the B.Sc. degree in electrical and electronic engineering from Bangladesh University of Engineering and Technology (BUET), in 2019, and the M.Sc. degree in electrical and computer engineering from the University of Florida, in 2024, where he is currently pursuing the Ph.D. degree with the Electrical and Computer Engineering Department, under the supervision of Dr. Farimah Farahmandi. His research focuses on hardware security and trust.

KATAYOON YAHYAEI (Student Member, IEEE) received the B.Sc. and M.Sc. degrees from the K. N. Toosi University of Technology, Iran, in 2018 and 2020, respectively. She is currently pursuing the Ph.D. degree in electrical and electronics engineering with the University of Florida. She is a Research Assistant with Florida Institute of Cyber Security (FICS), under Dr. Tehranipoor. Her research interest includes leveraging artificial intelligence for the security assessment of digital hardware designs.

SUJAN KUMAR SAHA (Member, IEEE) received the B.Sc. degree in electrical and electronic engineering from Bangladesh University of Engineering and Technology, in 2011, and the M.Sc. and Ph.D. degrees in computer engineering from the University of California, Riverside, in Spring 2018 and in Summer 2023, respectively. He is a Postdoctoral Associate with the Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA. His research interests include multi-processor system-on-chip (MPSoC) design, SoC security, cloud FPGA security, and real-time system design.



JINGBO ZHOU (Member, IEEE) received the B.Eng. degree in telecommunication engineering from Beijing University of Post and Telecommunication, Beijing, China, in 2018, and the Ph.D. degree from The Ohio State University, in 2023. He is a Postdoctoral Research Associate with the Department of Electrical and Computer Engineering, University of Florida. His research interests include hardware security, logic locking, side-channel attack, and VLSI design.

MARK TEHRANIPOOR (Fellow, IEEE) is currently the Intel Charles E. Young Preeminence Endowed Chair Professor in cybersecurity and the Chair of the Department of Electrical and Computer Engineering (ECE), University of Florida. His current research projects include hardware security and trust; supply chain security; IoT security; and VLSI design, test, and reliability. He has published over 400 journal articles and refereed conference papers and has delivered more than 220 invited talks and keynote addresses. He has 21 patents issued, 25 pending invention disclosures, and has published 16 books. He is a fellow of ACM and the National Academy of Inventors (NAI), a Golden Core Member of IEEE Computer Society, and a member of ACM SIGDA. He was a recipient of a dozen best paper awards and nominations. He served as the Founding Director for Florida Institute for Cybersecurity (FICS) Research, from 2015 to 2022. He has co-founded the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) and IEEE International Conference on Physical Assurance and Inspection of Electronics (PAINE).

FARIMAH FARAHMANDI (Member, IEEE) received the B.S. and M.S. degrees from the Department of ECE, University of Tehran, Iran, in 2010 and 2013, respectively, and the Ph.D. degree from the Department of CISE, University of Florida, in 2018. She is an Assistant Professor with the Department of ECE, University of Florida. Her research interests include design automation of system-on-chips and energy-efficient systems, formal verification, hardware security validation, and post-silicon validation and debug. Her research has resulted in five books, nine book chapters, and several publications in premier ACM/IEEE journals and conferences, including DAC and DATE, with awards including, the IEEE System Validation and Debug Technology Committee Student Research Award, the Gartner Group Info-Tech Scholarship, a nomination for the Best Paper Award in ASPDAC 2017, and the DAC Richard Newton Young Student Fellowship. She is a member of ACM.

• • •