Empowering Hardware Security with LLM: The Development of a Vulnerable Hardware Database

Dipayan Saha, Katayoon Yahyaei, Sujan Kumar Saha, Mark Tehranipoor and Farimah Farahmandi
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL, USA
{dsaha, ka.yahyaei, sujansaha}@ufl.edu, {tehranipoor, farimah}@ece.ufl.edu

2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST) | 979-8-3503-7394-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/HOST55342.2024.10545393

***Abstract*—The scarcity of comprehensive databases and benchmarks in hardware design specifically tailored for security tasks is a significant challenge in the community. Such databases are crucial for developing machine learning-based methods and benchmarking, providing a foundation for evaluating and improving hardware security solutions. However, manually creating these extensive datasets is impractical due to the significant time and effort required. Given the proficiency of large language models (LLM) in natural language processing, coding, and advanced reasoning tasks, using LLM as an artificial intelligence (AI) agent presents a viable option to efficiently create such extensive datasets. In this light, this paper introduces Vul-FSM, a database of 10,000 vulnerable finite state machine (FSM) designs incorporating 16 distinct security weaknesses and vulnerabilities generated using the proposed SecRT-LLM framework. The framework combines the in-context learning capability of LLM, the guidance of developed prompting strategies, and the scrutiny of fidelity-check to not only insert but also detect hardware vulnerabilities and weaknesses. To demonstrate the efficacy of SecRT-LLM, we present an exhaustive analysis, highlighting the proficiency of GPT models in vulnerability insertion, detection, and mitigation. Our proposed SecRT-LLM framework, using *gpt-3.5-turbo*, demonstrates strong effectiveness, achieving macro-average pass rates of 81.98% and 80.30% on the first attempt and 97.37% and 99.07% within five attempts for vulnerability insertion and detection, respectively.**

***Index Terms*—Large Language Model, ChatGPT, RTL design, Hardware Security, Vulnerability Analysis, Common Weakness Enumeration (CWE)**

I. Introduction

Security in system-on-chip (SoC) designs has emerged as a crucial aspect of contemporary computing due to the pervasive presence of SoCs in many devices, from edge devices (e.g., smartphones, IoTs) to large-scale computers such as data centers. As SoCs become increasingly larger, complex, and heterogeneous, these also witness a concurrent increase in security vulnerabilities that make them vulnerable to a variety of sophisticated and crucial security threats, such as microarchitectural attacks [1], [2], information leakage [3], hardware Trojans [4], side-channel attacks [5], and fault injection attacks [6]. Such advanced hardware attacks not only pose a risk to data confidentiality and integrity but also threaten the overall reliability and trustworthiness of the computing infrastructure. These vulnerabilities may come from various sources: unintentional design errors, malicious insider tampering, optimizations by computer-aided design (CAD) tools that overlook security implications [7], or inherent weaknesses in test and debug frameworks. Furthermore, the complexity

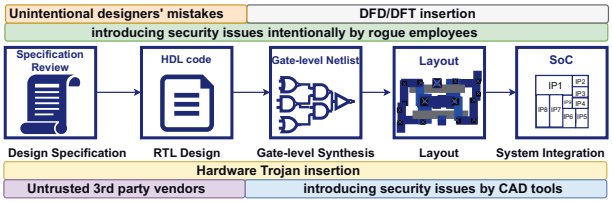


Fig. 1: Potential threats within the IP design flow.

of the SoC supply chain, which involves multiple third-party intellectual properties (3PIPs) from global vendors [8], introduces additional layers of risk. Figure 1 shows the security threats inherent in the IP design flow.

Vulnerability databases and studies such as common weakness enumeration (CWE) [9], common vulnerabilities and exposures (CVEs) [10], and initiatives such as Hack@Dac [11] and the HOST Microelectronics challenges [12] are instrumental in providing insights into hardware vulnerabilities. Currently, there are 117 hardware CWEs, three times the number reported in 2017. Such growth suggests a notable escalation in identified hardware vulnerabilities or weaknesses over these years. Unlike software bugs, which can often be patched post-deployment, these hardware vulnerabilities are not amendable to such fixes once the device is manufactured. This immutable nature of hardware requires detecting and mitigating vulnerabilities early in the design process, such as the register-transfer level (RTL), for enhanced security with cost efficiency. However, current hardware security verification methods, such as information flow tracking [13], assertion-based security property verification [14], fuzzing [15], runtime verification monitor [16], and Concolic testing [17], have their limitations. These techniques often lack adaptability, requiring considerable modifications to be applied to different designs. They also provide limited coverage and require substantial computational resources. Moreover, these methods typically demand extensive manual analysis of designs to identify threat models and vulnerabilities. These challenges emphasize the necessity for developing artificial intelligence (AI)-based security verification method.

Creating a comprehensive, security-aware database of hardware designs is imperative to effectively develop and enhance AI-based solutions for security-related tasks, especially for vulnerability detection and mitigation. This necessity is inspired by the widespread applications of the TrustHub vulnerability database [18], also proposed in previous stud-

TABLE I: Comparison of existing methods and proposed SecRT-LLM framework for vulnerability insertion

Methodology	Time Required	Effort Required	Accuracy
Manual Method	High	High	High
LLM Inference	Low	Low	Low
SecRT-LLM	Low	Moderate	High

ies as a potential future endeavor [19]. A rich and diverse database would support training deep learning architectures to autonomously detect and understand security vulnerability patterns in hardware designs and enhance the fine-tuning of pre-trained large language models (LLMs). Furthermore, such a database would serve as a critical benchmark in the field. It would provide a platform for evaluating and improving security verification methodologies, allowing qualitative and quantitative comparisons. Within the hardware security domain, there is a noticeable deficiency in comprehensive databases and benchmarks of hardware design. For instance, benchmarks such as Verigen [20] and RTLLM [21], though useful in non-security applications, cover a relatively small number of designs. This scarcity of an extensive security-aware database for vulnerable designs is primarily due to the challenges inherent in manually creating such extensive and detailed datasets for digital designs. The substantial effort and time required make manual compilation impractical.

To address the challenges in compiling an extensive security-aware dataset of hardware designs in a shorter time with high precision, employing AI agents presents a promising avenue to surmount these dataset creation hurdles. There is a pressing need for an AI agent that is capable of autonomous actions, possesses a nuanced understanding of complex hardware design, and can transfer knowledge of one design to another but requires the least manual intervention. In this context, LLMs stand out as promising candidates for this task with their natural language understanding, advanced reasoning, automated code generation, pattern recognition, and knowledge transfer capabilities. With such qualities, LLMs can significantly accelerate the database compilation process, making them practical and efficient tools. However, when traditional LLM inference methods are applied to hardware design security, they often lack precision, as highlighted in Table I. It is primarily due to the absence of strategic prompt engineering to adopt the hardware security knowledge and the lack of reliability. To address these issues, we present a comprehensive solution. Combining the text generation, linguistic understanding, and reasoning ability of LLM, the guidance of prompting strategies, and the scrutiny of fidelity-checking, our proposed SecRT-LLM framework enhances the accuracy of hardware security-oriented task completions. It is designed to introduce vulnerabilities into hardware designs and detect them with high proficiency.

The contributions of this work are multi-fold:

- We present a novel LLM-based framework adept at inserting and identifying hardware security vulnerabilities within a unified platform.
- We introduce an extensive database of 10,000 vulnerable RTL designs with various security weaknesses and breaches of security rules.
- We conduct an exhaustive analysis of the capabilities of different GPT models in terms of vulnerability insertion and detection within hardware designs.
- We formulate six specific prompting strategies to optimize the use of LLMs for hardware security-centric tasks.

The structure of this paper is organized as follows: Section II provides background information on LLMs and their relevance in hardware design and security. Section III explains our proposed SecRT-LLM framework. Section IV introduces and discusses the Vul-FSM dataset presented in this research. The experimental procedures and their results, along with an in-depth analysis, are presented in Section V. Finally, Section VI concludes the paper.

II. Background and Related Works

A. Preliminaries

1) Hardware Vulnerabilities: Hardware security vulnerabilities are design flaws that attackers can exploit. The high complexity of today's SoCs, coupled with an increased number of IP blocks and the presence of sensitive information, has led to the emergence of unique bugs, expanded attack surfaces, and new security vulnerabilities. According to an estimate from the CVE-MITRE database, it is believed that the elimination of hardware-level vulnerabilities could potentially reduce the overall vulnerability of the system by around 43%. One such vulnerability is CWE-1245, which addresses the issue of improper FSMs in hardware logic. Faulty FSMs can lead to undefined system states, causing a denial of service (DoS) or unauthorized privilege escalation. Such vulnerability may arise from a missing default statement that can leave the FSM without a defined behavior for unspecified conditions or inputs. This gap creates an opportunity for the insertion of hardware Trojans, which are malicious modifications to the circuit. These Trojans can remain dormant until triggered by specific conditions, potentially causing unexpected behavior or compromising the system's integrity.

2) Large Language Models: Large language models (LLMs), a subset of generative artificial intelligence (GAI), are used to understand, generate, and manipulate human language. They are based on the transformer architecture [22], a neural network design that allows efficient processing of data sequences, such as text. LLMs such as GPT-3 [23], LLama [24], BERT [25], and GPT-4 [26] work by analyzing vast amounts of text data and learning patterns and structures in the language. Furthermore, coding-specialized models, such as LLama-Code [27] and Codex [28], are proficient in programming tasks such as code generation, translation, and documentation. LLMs have evolved into various architectural categories: encoder-decoder, decoder-only, encoder-only, and sparse models, each tailored for specific tasks and applications. In this work, we are particularly interested in decoder-only models such as the generative pre-trained transformer (GPT) series [23], [26] because of their proficiency in language generation and understanding. GPTs such as GPT-3 [23], GPT-3.5, GPT-4 [26], Codex [28] autoregressively, using previously generated tokens as context for subsequent predictions. These models have particularly distinguished themselves as proficient in tasks that require unconditional language generation. In the

context of hardware security, these architectures are expected to excel in tasks that are predominantly generative, such as vulnerability insertion, security policy & property generation, and testbench generation.

3) Challenges with LLM: There are limitations associated with LLMs. One major issue is "Hallucination," which refers to the phenomenon where these models generate plausible-sounding responses that are either factually incorrect or not grounded in reality. LLMs hallucinate because they generate responses based on patterns in their training data, which can lead to confidently asserting incorrect information or creating entirely fictional narratives. The LLM-generated responses are not always reliable because of this tendency for hallucination. LLMs can generate erroneous designs that lead to significant functional, financial, or safety implications in hardware designs where precision and accuracy are paramount.

4) In-Context Learning: The ability to learn in context is a prominent feature of GPT models, especially as highlighted in GPT-3 [23] and GPT-4 [26]. In-context learning (ICL) [23] utilizes the model's inherent ability to adapt output based on examples or instructions in the input prompt, eliminating the need for traditional fine-tuning, and advanced adaptations like retrieval-augmented in-context learning broaden their capabilities. This approach is efficient, saving substantial computational resources and time. ICL includes zero, one, and few-shot learning [23]. Zero-shot learning allows models to rely on their understanding of concepts, and one-shot learning lets models master new tasks with only one example. A few-shot learning further extends this adaptability, enabling models to excel in tasks with limited training data. Rather than relying on traditional fine-tuning, in ICL, the LLM generates improved responses based on the given instructions associated with one or multiple examples.

5) Prompt Engineering: Prompt engineering [29] refers to the process of crafting natural language prompts that guide an LLM during inference. It involves providing context or examples for the LLM to learn in context. The effectiveness of in-context learning is highly dependent on the quality of these prompts. Different prompting strategies can lead to different outcomes, even with the same underlying model. Techniques such as chain-of-thought (CoT) [30] have been developed to enhance the effectiveness of prompting, aiming to guide LLMs more accurately during inference. In complex fields like hardware security, carefully crafted prompts ensure that the LLM understands the task requirements and context correctly.

6) Temperature Parameter of LLM: LLMs offer output control through parameters such as temperature, which are used during inference, not training. 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 produces a more uniform probability distribution, producing diverse and creative outputs. In contrast, a low temperature narrows this distribution, yielding more predictable and less creative output. This parameter is crucial in code generation, influencing output randomness and diversity.

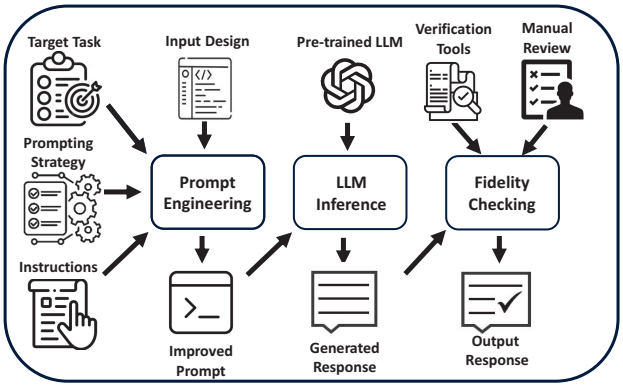


Fig. 2: Overview of proposed framewrok.

B. Related Works

Through literature review, practical case studies, and experiments, authors in [31] discussed the success, prospects, and challenges of using LLMs in SoC security verification. In another effort, M. Nair *et al.* [32] highlighted security issues in ChatGPT's prompt structure, proposing basic techniques for improvement, but need further research to address various hardware vulnerabilities. Another study [33] assessed code vulnerability but only conducted binary analysis without specifying the exact vulnerabilities. An LLM-based method in bug repair [19] outperformed non-AI solutions, but in this work, it is assumed that RTL designers detect vulnerabilities and LLMs solely assist in the repair. In another work, W. Fu *et al.* [34] proposed LLM4SECHW, a novel framework for hardware debugging using LLM. The framework utilized a dataset of open-source hardware design defects and remediation steps, enabling the identification and rectification of bugs in hardware designs. LLM4SECHW framework was designed to address the challenges of data scarcity and the need for domain-specific training in LLMs. While sharing some motivations with these studies, our work diverges significantly in approach and outcome. Our proposed SecRT-LLM framework efficiently generates a new dataset of vulnerable designs using LLM instead of compiling existing open-source vulnerable designs in [34].

III. Proposed Methodology

This section presents our proposed SecRT-LLM framework that seamlessly integrates LLM into two security tasks: vulnerability insertion and detection. It is important to note that these two tasks serve different yet complementary purposes. The vulnerability insertion is executed to generate a dataset of vulnerable hardware designs. On the other hand, the detection process aims to identify and address vulnerabilities in hardware designs, enhancing overall security and reliability.

A. Key Components

Figure 2 provides an overview of the SecRT-LLM framework. The figure shows that the proposed methodology includes three main components: prompt engineering, LLM inference, and fidelity-checking. The framework synergizes the precision of prompt engineering, the in-context learning capability of LLM, and the rigor of fidelity checks to redefine complicated security tasks.

1) Prompt engineering: As discussed in Section II-A5, the effectiveness of an LLM is heavily dependent on how the input prompt is structured because the model generates responses based on the cues and context provided in the prompt. In the SecRT-LLM framework, we have designed six distinct prompt strategies and applied them to ensure the finesse of the target task execution. These strategies are crucial for inserting or detecting vulnerabilities, as they guide the creation of improved and effective prompts for precise LLM inference. The choice of which strategy to employ depends on the nature and complexity of the task.

a) Reflexive Verification Prompting (RVP): It is a critical strategy emphasizing continuous self-scrutiny and correction by LLM throughout the inference process. This method, shown in Figure 3a, is generally used at the end of the prompt in the vulnerable design generation tasks and security assessment. We utilize this approach in two distinct scenarios. Firstly, the strategy involves the LLM verifying its adherence to all instructions specified in the prompt. RVP is crucial in tasks where precise steps must be followed, such as inserting weaknesses like deadlock, unreachability, or dead states. Occasionally, the LLM might not implement these steps accurately. The reflexive approach ensures that the LLM meticulously follows each instruction and corrects its response if any step is missed. This aspect of the strategy aids in rectifying flawed decision-making, thereby reducing the likelihood of hallucination by LLM, as discussed in Section II-A3. Secondly, RVP detects and rectifies potential coding issues, like syntax errors, that might be overlooked during the design generation. This prompting strategy integrates a robust initial filter into the design generation and security assessment phases through the LLM's self-evaluative capabilities. This integration enhances the quality and reliability of the outputs, ensuring more accurate and reliable results.

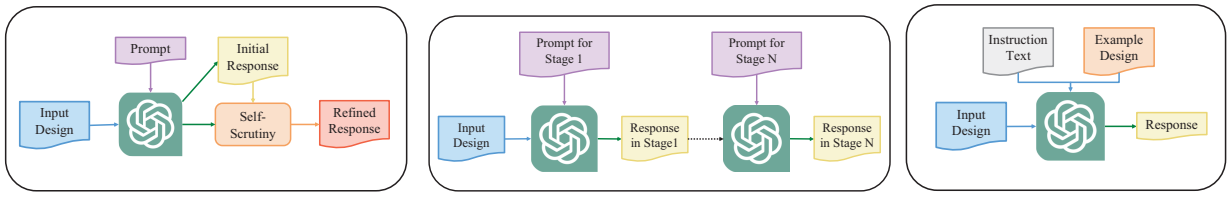
b) Sequential Integration Prompting (SIP): This prompting strategy, inspired by chain-of-thought (CoT) [30], involves a multi-stage methodology where the complete task is divided into multiple sub-tasks, and the output of one sub-task is used as input into the next. In this way, SIP, shown in Figure 3b, creates a coherent sequence of task execution and makes the decision-making process more logical. This technique is applied in complicated tasks of security vulnerability detection, which demands a comprehensive decision-making process. For instance, consider the task of detecting a violation of a specific security rule, such as ensuring that the Hamming distance (HD) between two consecutive unprotected states is exactly 1. Addressing this using a single-step approach can be challenging for an LLM, as it necessitates simultaneous arithmetic and logical reasoning. However, employing SIP, this task can be efficiently managed through a sequence of targeted steps: initially extracting the state transition graph (STG), followed by calculating the HDs, and finally, verifying adherence to the rule. Following such a SIP-based strategy for an FSM representing an RSA controller, the LLM responses are illustrated in Figure 4. This methodical breakdown significantly enhances the accuracy of security verification. Such a structured approach not only simplifies complex tasks but also ensures thoroughness in analysis and resolution.

c) Exemplary Demonstration Prompting (EDP): This strategic approach, shown in Figure 3c, employs practical, hands-on examples to guide and optimize task execution. The core motivation of EDP lies in providing clear and tangible reference points, significantly aiding in comprehending and implementing complex tasks. We use this method for both vulnerability insertion and detection. For instance, mere instructions may prove insufficient when integrating dynamic deadlocks into FSMs due to the task's complexity. To address this, EDP incorporates two demonstrative examples – one illustrating the design before and another after the insertion of a dynamic deadlock. This method mirrors the one-shot or few-shot learning concepts in in-context learning paradigms, enriching the pre-trained models' understanding of security vulnerabilities as highlighted in Section II-A4. By providing relatable examples, EDP not only eases the learning curve but also significantly enhances the precision and efficiency of task implementation, making it an invaluable strategy in the realm of complex hardware security tasks.

d) Contextual Security Prompting (CSP): This prompting method involves incorporating relevant security policy context into inserting and identifying security vulnerabilities and weaknesses. Typically, existing pre-trained LLMs lack detailed awareness of hardware vulnerabilities. To effectively perform intricate security tasks, these models require supplementary context. Hence, the core idea of CSP is to provide LLM with an enhanced understanding of potential security threats, bridging the gap between general language processing capabilities and specialized hardware security knowledge. For example, CSP defines and explains the concept within the prompt in detecting a specific vulnerability, such as CWE-367: Time-of-check time-of-use race condition. Such contextual clues certainly enhance the analytical capabilities of LLM.

e) Focused Assessment Prompting (FAP): It is a targeted approach emphasizing a detailed examination of a specific design element rather than the entire hardware design. The core motivation for FAP is its capacity to deliver thorough and focused analysis on particular components or aspects, particularly useful in pinpointing vulnerabilities tied to certain parts of a design. For instance, when identifying structural vulnerabilities such as deadlocks, the LLM's attention is best directed solely towards the STG. Similarly, in cases like detecting duplicate encoding states, a focused examination of the parameter declarations of encoding states is more efficient than analyzing the entire FSM. Broadly considering the entire FSM design for security flaws can lead to slower and potentially inaccurate results. By concentrating on specific areas, FAP facilitates uncovering detailed insights, thereby enabling a more effective and quicker security evaluation process.

f) Structured Data Prompting (SDP): This strategy focuses on the systematic arrangement of extensive data in a structured manner, such as a tabular format, thereby simplifying the process of vulnerability detection. The primary motivation for SDP stems from the necessity of efficiently handling large volumes of data. Organizing data in a structured manner makes it more comprehensible and navigable, particularly in scenarios involving extensive data computation. For instance, consider the task of detecting security vulnerability $V16$ men-



(a) Reflexive Verification Prompting (b) Sequential Integration Prompting (c) Exemplary Demonstration Prompting

Fig. 3: Different prompting strategies used in SecRT-LLM

Step 1: State transitions are listed, and the transition associated with the protected state is omitted.

State transitions occurred between unprotected states:
IDLE (0011) → INIT (0010)
MULT (1100) → SQR (1000)
SQR (1000) → MULT (1100)
SQR (1000) → RESULT (0001)
RESULT (0001) → IDLE (0011)
Since state LOAD1 and LOAD2 are protected, the state transition from INIT to LOAD1 and LOAD2 to MULT are removed.

Step 2: : For each transition, the XOR operation is performed, and the HD value is calculated.

HD: for modified state transitions (no protected state)
<state transition 1: IDLE (0011) → INIT (0010), 0001, calculated HD=1>
<state transition 2: MULT (1100) → SQR (1000), 0100, calculated HD=1>
<state transition 3: SQR (1000) → MULT (1100), 0100, calculated HD=1>
<state transition 4: SQR (1000) → RESULT (0001), 1001, calculated HD=2>
<state transition 5: RESULT (0001) → IDLE (0011), 0010, calculated HD=1>

Step 3: From the calculated HD values, rule violation is checked.

Evaluation: "rule is violated"
Explanation: The calculated HD for state transition 4 is 2, which is greater than 1. According to the rule, the Hamming distance (HD) between any two states should be 1 or less. As this rule is violated in this case, we can conclude that the rule is violated.

Fig. 4: Responses to prompts used in detecting the violation of the Hamming distance rule using SIP strategy.

tioned in Table II, where the LLM is required to compute the fault injection feasibility (FIF) metric [35], involving extensive calculations like multiplications and logical operations (XOR, OR, AND). In such cases, employing a tabular format is essential, as it helps to organize these calculations, which the LLM might not be inherently designed to handle efficiently. Additionally, SDP streamlines the detection of patterns and irregularities.

2) LLM Inference: LLM inference refers to the process of using a pre-trained model to generate a response based on input prompts. LLM, prompt, and control parameters - are the three key components of this inference process. Choosing

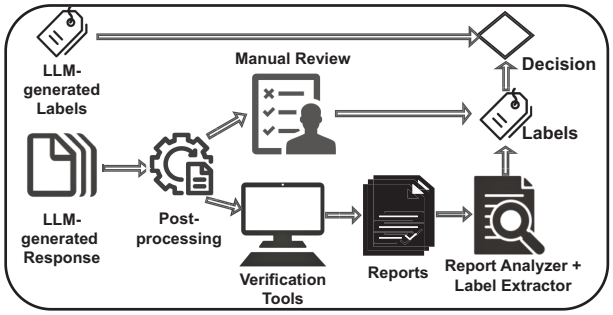


Fig. 5: Fidelity-checking framework in the context of vulnerability insertion.

LLM is crucial in this framework, with considerations like the task's complexity, the model's capabilities, and cost playing key roles. For more straightforward tasks like detecting duplicate encoding states in FSM design, more cost-effective models such as *text-davinci-003* or *gpt-3.5-turbo* are preferred, as they can efficiently handle such tasks at a lower cost. However, for more complex tasks such as inserting dynamic deadlock into FSM design, a more advanced model like *gpt-4* is recommended due to its superior capabilities. In this framework, we intentionally set this temperature parameter to a lower value to ensure a deterministic response behavior, discussed in Section II-A6. Additionally, we configure the context length settings to be sufficiently large, ensuring that we do not exceed the limits of the model.

3) Fidelity-Checking: To ensure the soundness of the generated design or detection report, the proposed SecRT-LLM includes a fidelity-checking process at the end of the flow. Static code analysis tools, such as linting tools, focus on coding standards and best practices but often lack a security-centric approach. On the other hand, tools such as ARC-FSM [35] are tailored for security vulnerability detection but have limited coverage due to their recent development. EDA tools, such as JG Superlint, offer automated formal checks, with the 'AutoFormal' mode further enhancing vulnerability detection by combining RTL analysis and property checking. Despite these advancements, specific vulnerabilities, such as CWE-364 and CWE-367, remain unverifiable by existing tools, necessitating time-consuming manual reviews. To bridge this gap, SecRT-LLM implements a hybrid verification approach that combines static analysis, formal verification, and manual code review. This strategy allocates vulnerabilities to either ARC-FSM or JasperGold (JG) Superlint based on the strengths of each tool. Vulnerabilities outside the scope of these tools are subjected to manual checks. Such a hybrid approach significantly trims

down verification time for expansive datasets. The assignment of various vulnerabilities to the respective checking methods within the SecRT-LLM framework is cataloged in Table II, ensuring a swift and reliable validation process.

In the SecRT-LLM framework, the fidelity-checking stage varies between vulnerability insertion and detection tasks. Figure 5 shows the process in the context of vulnerability insertion. As depicted in the figure, it begins with the LLM-generated response, which includes the LLM-modified design file and other metadata, such as an LLM-generated explanation of the presence of vulnerability. The LLM-generated response undergoes two post-processing phases: initially, a Python script separates the Verilog code from its accompanying metadata, and subsequently, an information extraction phase parses critical design information such as the top module, clock, reset, and if pertinent, the protected state's name. This information extraction step is mainly required for verification tools: JG Superlint and ARC-FSM. JG Superlint is also employed across all designs to verify the absence of syntax errors. These verification tools generate reports for each design which are subjected to analysis by report analyzers. These analyzers are Python-based scripts that identify predefined keywords in the reports associated with security weaknesses and then determine the appropriate labels. For vulnerabilities not covered by the automated tools, a manual inspection by human experts is conducted. This review is streamlined by the additional data provided by the SecRT-LLM framework, such as the STG generated by ARC-FSM and the LLM's explanations, facilitating a quicker manual review process. The vulnerability labels from the automated tools and manual review are then aggregated and compared with the LLM's assessment. If there is alignment between the labels and the LLM's assessment, the LLM-generated design is validated and accepted. However, if there is a discrepancy, it suggests that the LLM was ineffective in introducing the vulnerability into the design, and as a result, the design is flagged for rejection.

In the context of vulnerability detection, the fidelity checking is manually executed. Here, experts review the LLM's decisions and justifications. If there is a misalignment between the LLM's explanation, the design, or the decision, the identified decision is deemed inaccurate and is subsequently discarded.

B. Algorithms

1) Vulnerability Insertion: The algorithm for inserting vulnerability through the proposed SecRT-FSM is detailed in Algorithm 1. At first, the framework takes an HDL design $\mathcal{D}$ and a set of vulnerabilities $\mathcal{V}$ to be inserted as input. For each vulnerability v in the set $\mathcal{V}$, it selects an appropriate prompting strategy $\mathcal{PS}_v$ from a predefined set of strategies $\mathcal{PS}$ (**Line 2**). This selection process is manual, as described in Section III-A1. An improved and combined prompt is constructed using selected prompting strategy $\mathcal{PS}_v$, set of specific instructions I_v for inserting each vulnerability (**Line 3**). This prompt also includes the input design $\mathcal{D}$. For each vulnerability, an appropriate LLM is chosen $\mathcal{L}_v$ based on the nature of the vulnerability (**Line 4**). Later, selected LLM $\mathcal{L}_v$ first attempts to insert the vulnerability into the design and generated $\mathcal{D}'_v$ (**Line 5**). The generated design goes through a compliance check by LLM itself. In this step, the LLM checks through a set of queries $\mathcal{Q}_v$ whether the generated design has followed the instructions $\mathcal{I}_v$ (**Line 6-8**). If this initial attempt does not comply, the methodology uses the LLM to modify the design further. This step is part of the RVP strategy, described in Section III-A1a. After modification, the framework performs fidelity-checking through verification tools or manual intervention, explained in Section III-A3 (**Line 9-11**). If the design ($\mathcal{D}'_v$) passes this check, it is considered a valid modified design with the vulnerability inserted. This way, the framework outputs the final modified design ($\mathcal{D}_v$), incorporating the targeted vulnerabilities.

Algorithm 1 Security Vulnerability Insertion

Require: $\mathcal{D}$: HDL design, $\mathcal{V}$: Set of vulnerabilities to insert, $\mathcal{PS}$: Prompting Strategies, $\mathcal{I}_v$: Instructions for inserting $v \in \mathcal{V}$, $\mathcal{Q}_v$: Set of Queries, $\mathcal{L}$: Set of Pre-trained Decoder-Only LLMs.
Ensure: $\mathcal{D}_v$: Modified HDL design with vulnerability.
1: **for** $v \in \mathcal{V}$ **do**
2: $\mathcal{PS}_v \leftarrow$ SelectStrategy$(\mathcal{PS}, v)$
3: $P_{\text{improved}} \leftarrow$ Construct$(\mathcal{D}, \mathcal{PS}_v, \mathcal{I}_v)$
4: $\mathcal{L}_v \leftarrow$ SelectLLM$(\mathcal{L}, v)$
5: $\mathcal{D}'_v \leftarrow \mathcal{L}_v(P_{\text{improved}})$ /* LLM generates modified design with vulnerability v */
6: **if** !CheckCompliance$(\mathcal{L}_v, \mathcal{D}'_v, \mathcal{I}_v)$ **then**
7: $\mathcal{D}'_v \leftarrow$ Modify$(\mathcal{D}'_v)$ /* LLM modifies the design*/
8: **end if**
9: **if** FidelityCheck$(\mathcal{D}'_v)$ **then**
10: $\mathcal{D}_v \leftarrow \mathcal{D}'_v$ /* Accept design if it passes check */
11: **end if**
12: **end for**
13: **return** $\mathcal{D}_v$

Algorithm 2 Security Vulnerability Detection

Require: $\mathcal{D}$: HDL design, $\mathcal{V}$: Set of vulnerabilities, SP_v: Security Policy for $v \in \mathcal{V}$, $\mathcal{PS}$: Prompting Strategies, $\mathcal{I}_v$: Instructions for $v \in \mathcal{V}$, $\mathcal{L}$: Set of Pre-trained Decoder-Only LLMs.
Ensure: $\{R_v \mid v \in V\}$: Set of reports for each vulnerability.
1: **for** $v \in \mathcal{V}$ **do**
2: $\mathcal{PS}_v \leftarrow$ SelectStrategy$(\mathcal{PS}, v)$
3: $\mathcal{L}_v \leftarrow$ SelectLLM$(\mathcal{L}, v)$
4: $P_{\text{improved}} \leftarrow$ Construct$(\mathcal{D}, \mathcal{PS}_v, SP_v, \mathcal{I}_v)$
5: $(Dec_v, Expl_v) \leftarrow \mathcal{L}_v(P_{\text{improved}})$ /* LLM returns decision and explanation */
6: **if** FidelityCheck$(Dec_v, Expl_v)$ **then**
7: $R_v \leftarrow (Dec_v, Expl_v)$ /* Accept decision if explanation is reasonable */
8: **end if**
9: **end for**
10: **return** $\{R_v \mid v \in V\}$

2) Vulnerability Detection: Algorithm 2 describes the detailed methodology adopted for detecting vulnerability in the input design in the proposed SecRT-FSM. The algorithm

requires an HDL design $\mathcal{D}$, a set of defined vulnerabilities $\mathcal{V}$, a set of prompting strategies $\mathcal{PS}$, and also security policies SP_v, and set of instructions $\mathcal{I}_v$ for detecting each vulnerability v. Similar to Algorithm 1, here also suitable prompting strategies $\mathcal{PS}_v$ and LLM $\mathcal{L}_v$ are chosen according to the nature and complexity of vulnerability needs to be detected (**Line 2-3**). Later, an effective prompt is formulated using design $\mathcal{D}$, $\mathcal{PS}_v$, SP_v, $\mathcal{C}_v$ and $\mathcal{I}_v$ (**Line 4**) for the specific vulnerability v. The selected LLM $\mathcal{L}_v$ processes the design $\mathcal{D}$ and a decision Dec_v and explanation $Expl_v$ (**Line 5**). These generated responses of the LLM undergo a manual fidelity-checking that evaluates the reasonableness of the LLM-generated explanation. If the explanation for a particular vulnerability is deemed reasonable, the decision, along with its explanation, is accepted and added to the report (R_v) (**Line 6-8**). In this way, the proposed methodology outputs a set of reports R_v, each corresponding to a specific vulnerability v.

IV. Vul-FSM Dataset

We present a comprehensive database comprising 10,000 security-aware design files, each accompanied by security vulnerability labels, reports from JG Superlint or ARC-FSM fidelity checks, and other metadata. FSM designs featured in the dataset contain varying state counts ranging from 3 to 10. The dataset incorporates a set of security weaknesses, including eight hardware CWEs and eight additional hardware-related security guidelines gathered from previous studies [35]. Each vulnerability is briefly described in Table II, along with the verification method employed (JG Superlint, ARC-FSM, or manual review) and specific prompting strategies applied. Furthermore, we emphasize the significance of these 16 unique security weaknesses by delineating their impact on confidentiality, integrity, and availability (CIA) violations, both directly and indirectly. The potential consequences and security threats to the system containing such designs are also detailed. The Vul-FSM dataset is generated through our proposed SecRT-LLM framework using three GPTs: *text-davinci-003*, *gpt-3.5-turbo*, and *gpt-4*. Out of the 16 identified security vulnerabilities, 14 are introduced through the SecRT-LLM framework, with the exceptions being $V1$ and $V16$. These two are created by modifying the encoding states of the FSM via LLM. Initially, the framework embeds a single vulnerability into each of the 400 base FSM designs. It then adds another vulnerability on top, using an iterative process that builds on the model's responses at each step. The dataset is available at https://github.com/HWSecurityTeam/LLM-for-Vulnerability-Insertion-and-Detection.

V. Experiment and Results

A. Experimental Setup

In our experiments, we incorporate three GPT models within the proposed framework: *text-davinci-003*, *gpt-3.5-turbo*, and *gpt-4*. Though *text-davinci-003* and *gpt-3.5-turbo* come from the same GPT-3.5 series, they have distinct focuses. While *text-davinci-003* is larger and excels in complex and nuanced text generation, making it suitable for tasks needing in-depth understanding such as detailed problem-solving, *gpt-3.5-turbo* is optimized for speed and efficiency, ideal for real-time interactions in applications such as chatbots or quick information retrieval. Models like *gpt-4* offer increased context length and enhanced performance. Instead of local deployment, we specifically leverage the *gpt-4*, *gpt-3.5-turbo*, and *text-davinci-003* models available via APIs offered by OpenAI.

B. Evaluation Metrics

To characterize the performance of the framework, we use the $pass@k$ metric as an evaluation metric. It has been widely conceived to evaluate LLMs that translate natural language prompts into code [28]. Its significance lies in its ability to offer a nuanced understanding of the likelihood of selecting a sample that meets predefined benchmarks from a larger set. k signifies the number of samples used to estimate $pass@k$, while c represents the count of correct samples within the n samples. Given n, the $pass@k$ metric offers insight into the probability of selecting a sample that successfully meets predefined criteria when attempting to sample k from the set of n, as shown in the equation

$$pass@k := \underset{\text{problems}}{E}[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}] \tag{1}$$

A lower k value offers a more detailed, fine-grained assessment, while a higher k tends to yield statistically more stable outcomes. In prior studies [36] related to similar applications, k values of 1, 5, and 100 have been employed. Given the practical implications and drawing from these precedents, we have selected k=1 and k=5. Employing a k value greater than 5 complicates performance comparisons in our experiments.

C. Case Study

To explain how the proposed SecRT-LLM framework functions, we present a case study about the insertion of a security vulnerability. The presence of unreachable states in an FSM design that have transitions to non-reset states can introduce significant security vulnerabilities. Such vulnerabilities could be exploited for fault injection attacks, causing a denial of service, privilege escalation, or unintended data exposure. Inserting such a security bug is not a straightforward task. It requires a deep understanding of the logic and structure of the FSM design by LLM to introduce such vulnerabilities effectively. In this case, For LLM to infer effectively, we utilize the prompt structure depicted in Figure 6. The prompt starts with importing the FSM design and outlining the intended vulnerability insertion. Leveraging the CSP approach, detailed in Section III-A1d, we provide a clear context by explicating the definition of an "unreachable state" to avoid ambiguity. Subsequently, we furnish a precise, three-step procedure for the LLM to inject an unreachable state into the design. Following the EDP strategy described in Section III-A1c, we present two example designs that illustrate the FSM with and without the inserted vulnerability, noting that these examples differ from the input design. Further, we append optional instructions to standardize module and signal names across designs, simplifying the fidelity-checking process. These steps also guide the LLM in correcting common syntactical errors. Next, we apply the RVP strategy, outlined in Section III-A1a, to enable self-review by the LLM. At the end of the prompt, LLM is requested to provide the modified, vulnerable design alongside an explanation of the introduced vulnerability, which

TABLE II: Description of security weakness and vulnerability included in the dataset. CIA stands for Confidentiality(Con.), Integrity(Int.), Availability(Avl). Also I: indirect, P: primary. PS stands for the protected state.

ID	Weakness	Security Weakness/ Vulnerability Description	CIA violation	Consequence	Possible Attack	Fidelity-Checking	PS for Insertion	PS for Detection
V1	Hamming Distance	HD between two consecutive unprotected states is not 1	Int., Avl.(I), Con.(I)	Denial of Service, Data Corruption, Access Control	Timing Attacks, Fault Injection	ARC-FSM, JG Superlint	-	SIP, SDP
V2	Static Deadlock	Absence of any outgoing paths from current state with a condition	Avl.	Denial of Service, Data Loss	Timing Attacks, Injection Attacks, Information Leakage	ARC-FSM, JG Superlint	EDP, RVP, CSP	SIP, CSP
V3	Dynamic Deadlock	Existence of a group of states with a dynamic deadlock loop in the gate-level STG with a PS	Avl.	Denial of Service, Data Loss, Access Control	Timing Attacks, Injection Attacks, Information Leakage	ARC-FSM, JG Superlint	EDP, RVP, CSP	SIP, EDP, CSP
V4	Dead State	Including dead (inactive) states from the extracted STG of an FSM	Avl., Con.(I)	Deadlocks, Denial of Service, Data Integrity	Injection Attack, Information Leakage	ARC-FSM, JG Superlint	CSP, EDP, RVP	SIP, CSP
V5	Unreachable State	Presence of unreachable states that have a transition to a non-reset state in the STG	Con.(I), Int.(I), Avl.(I)	Data Corruption, Access Control, Denial of Service	Information Leakage, Privilege Escalation, Injection Attacks	ARC-FSM, JG Superlint	CSP, EDP, RVP	SIP, CSP
V6	Livelock	Absence of any paths back to the initial state	Avl.	Denial of Service, Data Loss or Corruption	Timing Attack, Injection Attack, Information Leakage	ARC-FSM, JG Superlint	EDP, RVP, CSP	SIP, CSP
V7	Duplicate Encoding	Repetitive encoding in the states of a control FSM	Avl., Int.(I)	Data Corruption, Denial of Service	Information Leakage, Injection Attacks, Privilege Escalation	ARC-FSM, JG Superlint	EDP, RVP, FAP	CSP, EDP, FAP
V8	CWE-190	Integer Overflow or wraparound	Int.	Data Corruption, Inaccurate Results, Denial of Service	Buffer Overflow Attack, Arbitrary Code Execution	JG Superlint	EDP, RVP	EDP, CSP
V9	CWE-364	Signal Handler Race Condition	Con., Int., Avl.	Access Control	Information Leakage, Denial of Service, Fault Injection	Manual, JG Superlint	EDP, CSP, RVP	EDP, CSP
V10	CWE-367	Time-of-check Time-of-use Race Condition	Int.(P), Con., Avl.(I)	Unauthorized Data Modification, Unauthorized Access	Privilege Escalation, Denial of Service	Manual, JG Superlint	EDP, CSP, RVP	EDP, CSP
V11	CWE-561	Presence of dead code	Int.	Unexpected Behavior, Inconsistent Computations	Fault Injection, Denial of Service, Information Leakage	JG Superlint	EDP, CSP, RVP	CSP
V12	CWE-570	Expression Always False	Con., Int., Avl.(I)	Data Corruption	Unauthorized Access, Man-in-the-Middle Attack	Manual, JG Superlint	CSP, EDP, RVP	CSP
V13	CWE-571	Expression Always True	Int.(P), Con., Avl.(I)	Memory Corruption, Information Disclosure, System Crash	Fault Injection, Information Leakage	Manual, JG Superlint	CSP, EDP, RVP	CSP
V14	CWE-835	Loop with Unreachable Exit Condition	Avl.	Denial of Service, Resource Exhaustion, Operational Disruptions	Botnet Amplification, Exploitation of Delayed Responses	JG Superlint	CSP, EDP, RVP	CSP
V15	CWE-1245	Presence of unused states without the 'default' statement	Avl.	Access Control, Denial of Service, Instability	Manipulating State Attack	ARC-FSM, JG Superlint	CSP, FAP, RVP	CSP, FAP
V16	FIF metric	FIF metric between two consecutive unprotected states is not 0	Int., Avl.(I), Con.(I)	Denial of Service, Data Corruption, Access Control	Timing Attacks, Fault Injection	ARC-FSM, JG Superlint	-	SIP, EDP, CSP

aids in subsequent fidelity verification if needed. The LLM's response is outputted in a text file, from which a Python-based code extractor separates the LLM-generated code from the accompanying metadata. The final design is then subjected to fidelity-checking using verification tools, as elucidated in Section III-A3.

D. Results

1) Insertion of Vulnerability: In our study, we have evaluated the capability of SecRT-LLM to insert security vulnerabilities using *gpt-3.5-turbo* as the LLM, presented in Table III. As noted in Section IV, vulnerabilities $V1$ and $V16$ were inserted into the design indirectly. Therefore, the table does not include the performance metrics of *gpt-3.5-turbo* in injecting these specific vulnerabilities. Additionally, to insert the dynamic deadlock vulnerability ($V3$), we exclusively used *gpt-4* due to the greater complexity involved in this task. From the table, we can observe that no vulnerability scored below 95% in the $pass@5$ metric, suggesting the generalizability of the framework. Even for complex structural issues such as static deadlock ($V2$), the framework achieved an 86.23% score in $pass@1$. The lowest $pass@1$ scores are observed for $V6$ and $V5$, representing livelock and unreachable state creation in FSM design. Such lower performance can be linked to our specific definition of an unreachable state in an FSM, which closely mirrors the definitions of both dead state and static deadlock, causing potential ambiguities. These ambiguities inadvertently introduce other vulnerabilities, resulting in a reduced $pass@1$.

Your task is to perform the following actions:
————————Importing input design————————
Now, read the following Verilog code delimited by <>
Code: <Input Design>
————————Target Task————————
Modify the code by inserting an unreachable state.
————Contextual Security Prompting————
Unreachable state is a state that has a transition to one or multiple non-reset stats but has no incoming state condition.
————————Instructions————————
To do this, execute the following steps
Step 1: Add an additional state to the parameter list and case statement.
Step 2: There should be no transition to this additional state from any other existing state. Therefore, do not mention this state in the other case statement for the transition.
Step 3: Choose any non-reset state (suppose state A) from the parameter list. Make an outgoing transition from the additional state to state A.
———Exemplary Demonstration Prompting———
For example,
Before adding unreachable state: <design without unreachable state>
After adding unreachable state: <design with unreachable state>
Now implement the unreachable state in the code.
————Reflexive Verification Prompting————
Make sure that all three steps are followed.
Explanation: Where and how have Step 1, Step 2, and Step 3 been followed in the code? Mention line no. also where steps 1, 2, and 3 have been implemented.
Review 1: In the modified code, which is the additional state? Does it have an incoming connection? if yes, rewrite the code.
Review 2: Does the additional state have any outgoing connection? To which state is it connected?
Review 3: Is there any issue regarding syntax, coding style, and synthesis? If so, correct the problem.
————————-Output————————-
While giving a response, write the modified code in the following format delimited by [ ].
[code: < modified code >
explanation: < explain how it has unreachable state >]

Fig. 6: Prompt used in LLM inference for insertion of unreachable state.

2) Vulnerability Detection: To assess the ability of the proposed SecRT-LLM framework to detect vulnerability, we supplied both vulnerable and secure FSM designs to *gpt-3.5-turbo*. We removed all comments and altered variable names to ensure the designs gave no hints about the vulnerabilities. Importantly, we have not solely relied on the decisions of LLMs. We thoroughly reviewed their explanations, considering a detection correct only if their reasoning was precise and valid. Table III shows the performance of *gpt-3.5-turbo* in detecting security vulnerabilities in terms of $pass@1$ and $pass@1$. The macro-average scores across all vulnerabilities for $pass@1$ and $pass@5$ are 80.30% and 99.07%, respectively. These averages suggest the model's overall strong capability in vulnerability detection, with exceptionally high reliability in identifying vulnerabilities within the first five attempts. For $pass@1$, which measures the model's ability to identify a vulnerability on its first attempt correctly, the performance varies across different vulnerabilities. For instance, the detection rates are particularly high for comparatively easy-to-detect vulnerabilities such as duplicate encoding ($V7$), CWE-190 ($V8$), CWE-571 ($V13$), CWE-835 ($V14$), and CWE-1245 ($V15$), with scores above 90%. However, for complex vulnerabilities such as unreachable state ($V5$), the detection rate is considerably lower, at around 32.98%. Overall, the results suggest that while *gpt-3.5-turbo* excels in detecting various vulnerabilities, especially when given multiple attempts, there are certain types of vulnerabilities where its initial detection capability could be further improved.

TABLE III: Performance of *gpt-3.5-turbo* using proposed SecRT-LLM in inserting and detecting vulnerability in terms of $pass@1$ and $pass@5$ metrics.

ID	Vulnerability Insertion		Vulnerability Detection	
	$pass@1$	$pass@5$	$pass@1$	$pass@5$
$V1$	-	-	82.29	99.98
$V2$	86.23	99.99	82.83	99.99
$V4$	75.11	99.91	84.23	99.99
$V5$	64.14	99.12	32.98	86.83
$V6$	52.51	97.77	66.01	99.60
$V7$	87.65	99.99	91.36	99.99
$V8$	75.00	99.92	97.33	99.99
$V9$	89.82	99.99	73.99	99.90
$V10$	83.84	99.99	78.57	99.96
$V11$	100.00	100.00	72.66	99.88
$V12$	91.83	99.99	89.33	99.99
$V13$	90.40	99.99	90.71	99.99
$V14$	92.77	99.99	92.20	99.99
$V15$	76.44	99.93	91.10	99.99
$V16$	-	-	78.94	99.97
Macro-Average	81.98	97.37	80.30	99.07

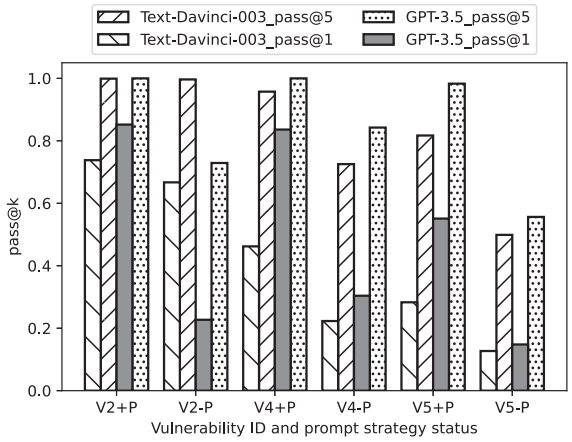


Fig. 7: Comparison of *gpt-3.5-turbo* and *text-davinci-003* in injecting vulnerability with or without prompting strategies in terms of $pass@1$ and $pass@5$ metrics. 'V2+P' and 'V2-P' indicate the inclusion and the absence of prompting strategies, respectively.

3) Impact of Prompt Engineering: In SecRT-LLM, we employed various prompting strategies, described in Section III-A1, to adeptly guide the LLM in executing complex security tasks. To investigate the influence of these strategies across different LLM architectures in inserting vulnerabilities, we compared the performances of *gpt-3.5-turbo* and *text-davinci-003* in injecting vulnerabilities across three distinct

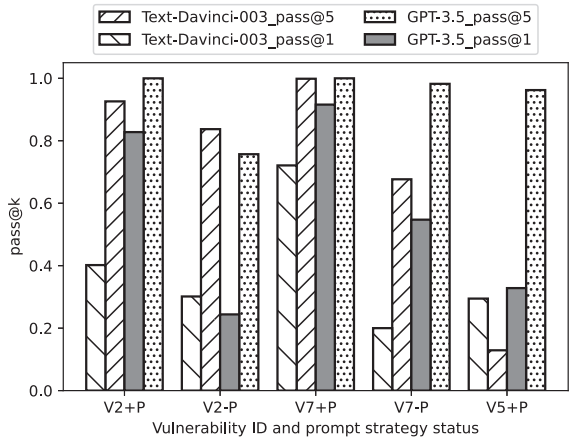


Fig. 8: Comparison of *gpt-3.5-turbo* and *text-davinci-003* in the detection of vulnerability with or without prompting strategies in terms of $pass@1$ and $pass@5$ metrics.

scenarios, as depicted in Figure 7. This evaluation was carried out on 1,661 designs, with both LLMs tasked to insert vulnerabilities $V2$, $V4$, and $V5$ into the designs, both with and without prompting strategies. For static deadlock insertion ($V2$), including prompting strategies enhances the $pass@1$ score for *gpt-3.5-turbo* by 27%. Intriguingly, in the absence of these strategies, *text-davinci-003* has outperformed *gpt-3.5-turbo*, possibly due to its training corpus being richer in static deadlock data, granting it a superior understanding. Although both models have benefitted from the prompting guidelines, *gpt-3.5-turbo* seems to benefit more from the prompting strategies than *text-davinci-003*, especially in vulnerabilities $V2$ and $V4$. It suggests that while *text-davinci-003* might have inherent strengths in certain areas, the performance of *gpt-3.5-turbo* can be significantly enhanced with the proper guidance.

Similarly, to investigate the impact of prompting in the context of vulnerability detection, we provide both vulnerable and secure FSM designs to *gpt-3.5-turbo* and *text-davinci-003*, instructing them to determine the presence of specific vulnerabilities. Figure 8 illustrates the impact of prompting in different scenarios. For all types of vulnerabilities, namely static deadlock, duplicate encoding, and unreachable states, introducing a prompting strategy consistently has increased the accuracy for both LLMs, suggesting the significance of apt guidance for improved LLM performance. For vulnerabilities such as duplicate encoding and unreachable states, *gpt-3.5-turbo* have consistently surpassed *text-davinci-003*. However, for the "static deadlock" vulnerability, *text-davinci-003* has a slight edge over *gpt-3.5-turbo* without prompting, outperforming it by 5.74%. This result aligns with our prior observations from the vulnerability insertion experiment, suggesting that *text-davinci-003* inherently possesses a better grasp of the static deadlock scenario when unguided. Interestingly, without prompting, both models have exhibited diminished performance in vulnerability detection compared to insertion. This can be attributed to the inherent design of GPT models, which are decoder-only and are naturally more adept at generation tasks than analysis. Notably, for the detection of unreachable state ($V5$), the absence of prompting led to an omission in the figure, as the LLMs incorrectly analyzed all provided designs without prompting guidance.

4) Time and Cost: Adopting the SecRT-LLM framework for creating a large database of 10,000 hardware designs, as detailed in Section IV, significantly reduces the time and effort compared to manual processes. Consider a hardware engineer needs about 10 minutes per design to write and verify a Verilog code embedded with hardware vulnerability, which equates to roughly 1667 working hours for a database similar to Vul-FSM. In contrast, SecRT-LLM drastically cuts down this time, taking only 8 seconds to 1 minute to inject or detect a vulnerability during the LLM inference phase, depending on the connection to the OpenAI server and task complexity. Fidelity-checking through verification tools is also quick, needing just a few seconds, and manual review required only occasionally, takes about 5-10 minutes. In this way, the SecRT-LLM approach not only accelerates the database creation process but also minimizes the manual labor involved.

The cost of utilizing LLMs depends on the number of input tokens and output tokens involved in the task. We worked with designs ranging from 500 to 2200 tokens in our experiments. On average, we provided 2000 input tokens (1000 tokens for input design and 1000 tokens for prompt) and expected output of around 1500 tokens. For such a design, utilizing *gpt-3.5-turbo* incurs minimal costs. To put it in perspective, a single operation of inserting a vulnerability in a design of this token range costs only about 0.005 USD with *gpt-3.5-turbo* using OpenAI's API. The affordability becomes even more evident when considering the creation of a comprehensive database. Specifically, all of our experimental works with the SecRT-LLM framework, described in Section V-D1, Section V-D2 and Section V-D3, cost 191.90 USD only. This amount included 86.90 USD for *text-davinci-003*, 65.09 USD for *gpt-4*, and 40 USD for *gpt-3.5-turbo*. Higher costs for *gpt-4* and *text-davinci-003* were due to their token rates being 33.3 and 13.33 times more than *gpt-3.5-turbo*. Moreover, it is essential to note that the cost of utilizing LLMs is decreasing. For example, the rate for *gpt-3.5-turbo* at the time of our experiments was 50% higher than its current rate. This decreasing cost trend makes LLMs an even more attractive option for complex security tasks, offering a combination of high efficiency and affordability.

VI. Conclusion

SecRT-LLM offers an effective solution to compile large datasets of vulnerable hardware designs in the least time with high precision and minimal effort. The framework is also capable of detecting hardware bugs. SecRT achieves this advanced level of security task execution by integrating the advanced capabilities of LLM in text generation, linguistic understanding, and reasoning, along with effective prompting strategies and rigorous fidelity checks. In this work, we also introduce a database of 10,000 vulnerable FSM designs developed through SecRT-LLM. This dataset can be instrumental in developing AI-based security solutions. We plan to use this database to develop a GNN-based approach for hardware vulnerability detection and fine-tune LLM for vulnerability mitigation.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," *Communications of the ACM*, vol. 63, no. 7, pp. 93–101, 2020.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom *et al.*, "Meltdown: Reading kernel memory from user space," *Communications of the ACM*, vol. 63, no. 6, pp. 46–56, 2020.

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

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

[5] Z. He and R. B. Lee, "How secure is your cache against side-channel attacks?" in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 341–353.

[6] A. Nahiyan, F. Farahmandi, P. Mishra, D. Forte, and M. Tehranipoor, "Security-aware fsm design flow for identifying and mitigating vulnerabilities to fault attacks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 6, pp. 1003–1016, 2019.

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

[8] S. Tarek, H. Al Shaikh, S. R. Rajendran, and F. Farahmandi, "Benchmarking of soc-level hardware vulnerabilities: A complete walkthrough," in *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2023.

[9] "Common weakness enumeration." [Online]. Available: https://cwe.mitre.org/data/index.html

[10] "Common vulnerabilities and exposures," 1999. [Online]. Available: http://cve.mitre.org

[11] "Hack@dac'23." [Online]. Available: HACK@DAC23,https://hackatevent.org/hackdac23/

[12] "Host 2023 microelectronics security competition: Soc security track." [Online]. Available: http://www.hostsymposium.org/host2023/doc/SoC-Security-Track_2023.pdf

[13] A. Ardeshiricham *et al.*, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, 2017, pp. 1691–1696.

[14] H. Witharana *et al.*, "Automated generation of security assertions for rtl models," *J. Emerg. Technol. Comput. Syst.*, vol. 19, 2023.

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

[16] Ali. Kassem *et al.*, "Detecting fault injection attacks with runtime verification," in *Proc. of the 3rd ACM Workshop on Software Protection*, 2019, p. 65–76.

[17] X. Meng *et al.*, "Rtl-contest: Concolic testing on rtl for detecting security vulnerabilities," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 41, no. 3, pp. 466–477, 2021.

[18] "trust-hub.org." [Online]. Available: https://trust-hub.org/#/data/Security-Properties-Rules.

[19] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "On hardware security bug code fixes by prompting large language models," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2024.

[20] 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 Transactions on Design Automation of Electronic Systems*, 2023.

[21] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "Rtllm: An open-source benchmark for design rtl generation with large language model," *arXiv preprint arXiv:2308.05345*, 2023.

[22] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2023.

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

[24] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar *et al.*, "Llama: Open and efficient foundation language models," *arXiv preprint arXiv:2302.13971*, 2023.

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

[26] "Gpt-4 technical report." [Online]. Available: https://arxiv.org/pdf/2303.08774.pdf

[27] "Introducing code llama," Software, Meta AI, 2023. [Online]. Available: https://github.com/facebookresearch/codellama

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

[29] L. Reynolds and K. McDonell, "Prompt programming for large language models: Beyond the few-shot paradigm," in *Extended Abstracts of the 2021 CHI Conference on Human Factors in Computing Systems*, 2021, pp. 1–7.

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

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

[32] M. Nair, R. Sadhukhan, and D. Mukhopadhyay, "How hardened is your hardware? guiding chatgpt to generate secure hardware resistant to cwes," in *International Symposium on Cyber Security, Cryptology, and Machine Learning*. Springer, 2023, pp. 320–336.

[33] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," in *2022 IEEE Symposium on Security and Privacy (SP)*, 2022, pp. 754–768.

[34] W. Fu, K. Yang, R. G. Dutta, X. Guo, and G. Qu, "Llm4sechw: Leveraging domain-specific large language model for hardware debugging," in *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 2023, pp. 1–6.

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

[36] S. Fakhoury, S. Chakraborty, M. Musuvathi, and S. K. Lahiri, "Towards generating functionally correct code edits from natural language issue descriptions," *arXiv preprint arXiv:2304.03816*, 2023.