SPELL: An End-to-End Tool Flow for LLM-Guided Secure SoC Design for Embedded Systems

Sudipta Paria, Aritra Dasgupta, *Student Member, IEEE*, and Swarup Bhunia, *Fellow, IEEE*

***Abstract*—Modern embedded systems and Internet of Things (IoT) devices contain System-on-Chips (SoCs) as their hardware backbone, which increasingly contain many critical assets (secure communication keys, configuration bits, firmware, sensitive data, etc.). These critical assets must be protected against wide array of potential vulnerabilities to uphold the system's confidentiality, integrity, and availability. Today's SoC designs contain diverse intellectual property (IP) blocks, often acquired from multiple 3rd-party IP vendors. Secure hardware design using them inevitably relies on the accrued domain knowledge of well-trained security experts. In this work, we introduce `SPELL`, a novel end-to-end framework for the automated development of secure SoC designs. It leverages conversational Large Language Models (LLMs) to automatically identify security vulnerabilities in a target SoC and map them to the evolving database of Common Weakness Enumerations (CWEs); `SPELL` then filters the relevant CWEs, subsequently converting them to SystemVerilog Assertions (SVAs) for verification; and finally, addresses the vulnerabilities via centralized security policy enforcement. We have implemented the `SPELL` framework using popular LLMs, such as ChatGPT and GEMINI, to analyze their efficacy in generating appropriate CWEs from user-defined SoC specifications and implement corresponding security policies for an open-source SoC benchmark. We have also explored the limitations of existing pre-trained conversational LLMs in this context.**

***Index Terms*—SoC Security, Security Policies, Assertion Based Verification, CWEs, LLMs.**

I. Introduction

The complexity of modern System-on-Chips (SoCs) in the Internet of Things (IoT) regime makes it highly challenging for designers to address the vast and diverse array of security vulnerabilities effectively. The increasing number of on-chip assets that an SoC designer needs to protect against unauthorized access accentuates these challenges. Consequently, automation of security design and verification is becoming imperative to mitigate manual efforts, strengthen the security of a target SoC design, and minimize the SoC manufacturing cost. The collaborative efforts of the semiconductor industry have introduced hardware-related concerns to MITRE's list of Common Weakness Enumerations (CWEs)[1]. Identifying potential vulnerabilities necessitates varying degrees of expertise about the design, assets, threat model, security requirements, etc. [1], [2]. Existing methods involve manual assessment of the Hardware Description Language (HDL) code, relying on human expertise to discover vulnerabilities. Conversational Large Language Models (LLMs) such as Open AI's ChatGPT and Google's GEMINI (formerly BARD) have displayed a remarkable ability to comprehend natural language prompts.

[1]https://cwe.mitre.org/data/definitions/1194.html

The works presented in [3]–[5] are limited to LLM-based Verilog code generation and evaluation without bug detection or fixes. Latest works on using LLMs for verification include generating security properties [6], SystemVerilog Assertions (SVAs) [7], [8] for formal verification, while bug fixing [9], [10] involves static analysis and security-related feedback [11]. Authors in [12] presented the effectiveness of LLMs in applications such as an engineering assistant chatbot, tool script generation, and bug summarization with domain-specific adaption. However, the current methodologies are limited to simpler hardware designs, mainly covering specific vulnerabilities but not explicitly addressing security requirements for generic bus-based SoC designs. This paper presents `SPELL`, an end-to-end framework for SoC security analysis and policy-based protection. `SPELL` leverages the ability of LLMs to identify the CWEs for a given SoC specification, employs a novel LLM-based filtering technique to determine the relevant CWEs and convert them into SVAs for verification. The proposed framework generates the equivalent security policies in a 3-tuple format for each SVA activated during verification, followed by security policy enforcement to address potential vulnerabilities. Table I provides a comparative analysis between the existing solutions and `SPELL`. The proposed `SPELL` framework is illustrated in Fig. 1. This paper makes the following major contributions:

- For the first time to our knowledge, a comprehensive end-to-end automated tool flow for generating secure SoC designs starting from security specifications and ending with security policy enforcement.
- Automated query generation from security requirements to identify CWEs by leveraging the LLM knowledge base for any generic bus-based SoC.
- Filtering to identify relevant CWEs for the given SoC context and vulnerabilities, as well as analysis & correction of SVAs generated by LLMs for a given SoC.
- Automatic mapping of SVAs into judicious security policies and centralized security policy enforcement.

The remainder of this paper is organized as follows: Section II describes the background and motivation. Section III outlines the methodology, and Section IV contains the experimental results. We conclude the paper in Section V.

II. Background & Motivation

Bus-based SoCs typically include multiple IP cores sourced from vendors with varying trust levels, necessitating a process to identify common security issues in SoCs. CWEs act as a universal database for categorizing known hardware vulnerabilities and enforcing safeguards. Examples of

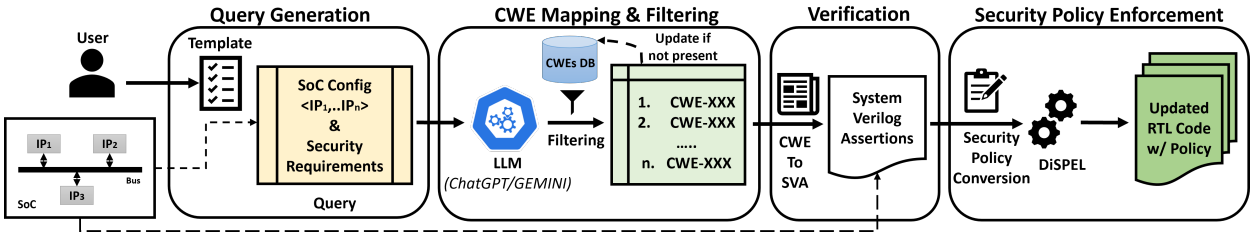

Fig. 1: SPELL: Overview of the proposed end-to-end secure SoC design and verification framework.

TABLE I: Comparison of SPELL with existing solutions.

Proposed Solutions	F_1	F_2	F_3	F_4	F_5	F_6	F_7
Don't CWEAT It [11]	✗	✗	✓	✗	✗	✗	✓
Chip-Chat [5]	✗	✓	✗	✗	✗	✗	✗
Ahmad *et al.* [9]	✓	✓	✓	✗	✗	✗	✓
Kande *et al.* [7]	✓	✓	✓	✗	✗	✓	✗
Pearce *et al.* [10]	✗	✓	✓	✗	✗	✗	✓
SPELL (This work)	✓	✓	✓	✓	✓	✓	✓

Features → F_1: Applicable to SoCs?, F_2: Uses LLMs?, F_3: Mapping to CWEs?, F_4: Filtering CWEs?, F_5: Support Generic SoCs?, F_6: Generate Assertions?, F_7: Perform Bug Fix?

CWEs corresponding to security vulnerabilities for bus-based SoCs include: *CWE-284* (Improper Access Control), *CWE-522* (Insufficiently Protected Credentials), *CWE-1231* (Improper Prevention of Lock Bit Modification), etc. Assertion-Based Validation (ABV) aims to uncover SoC security flaws that can potentially lead to attacks. Current ABV practices rely on manual efforts by security experts to test hardware designs against common CWEs. This method is slow, requires human creativity, and lacks broad applicability. Overcoming these issues requires automation that can identify vulnerabilities and apply necessary corrective measures for any given SoC specifications. Incorporating security policies is essential to systematically represent and enforce security requirements. Recent advancements in AI have led to the rise of LLMs which excel in generating context-specific answers from natural language descriptions. The main goal is to identify or map the user-given design specifications and security requirements to the existing list of CWEs available on the web, leveraging the knowledge base of LLMs. In this work, we analyze the performance of popular conversational LLMs like ChatGPT and GEMINI (formerly BARD) under various attack models prevalent in the SoC security literature.

III. Methodology

In this section, we describe the major stages of the SPELL framework, as illustrated in Fig. 2.

A. Design Specification & Query Generation

SPELL tokenizes the SoC design specifications detailing IPs, bus-level, and overall configuration to generate corresponding queries for LLMs like ChatGPT or GEMINI, including user-provided security assumptions. A user-friendly JSON template is provided for SoC specifications, accommodating those unfamiliar with specific security concerns.

B. CWE Mapping using LLMs & Filtering

SPELL leverages the context-sensitive factual answering capabilities of LLMs to identify relevant CWEs for the SoC. The performance of LLMs heavily relies on the query context. Table II displays CWEs generated by ChatGPT and GEMINI under various attack models for SoC configurations. Due to an LLM's unpredictable performance arising from its limited domain knowledge, a filtering step is essential to identify relevant CWEs. An extensive database (Λ), comprising approximately 180 CWEs and respective classifications, is constructed for filtering after a comprehensive analysis of various hardware vulnerabilities listed on the MITRE website[1]. Table III shows a partial snapshot of Extensive DB with different CWEs. The filtering method utilizes Cosine Similarity scores for semantic context matching for CWE bug description.

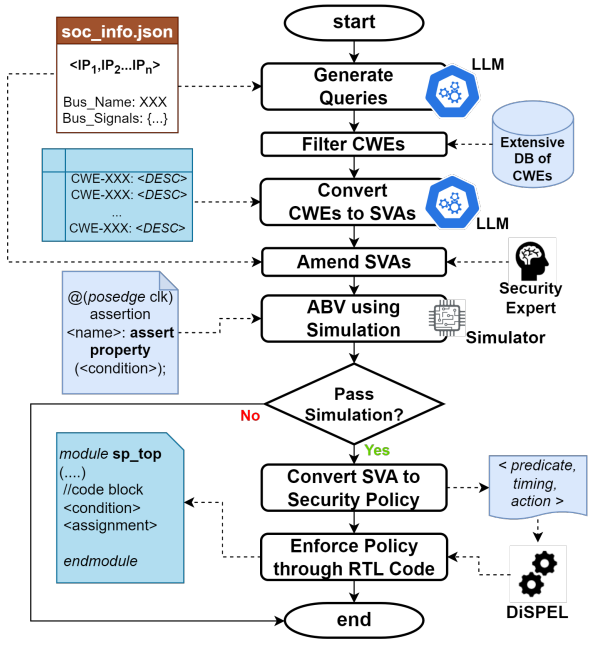

Fig. 2: SPELL: Flow diagram showing the major steps.

C. Security Assertion Creation & Verification

After filtering for relevant CWEs, SPELL employs ABV to identify the vulnerabilities of the current SoC. The automated flow utilizes LLMs to generate initial SVAs and then adapts them to match the design specifications for syntactical correctness. SPELL employs the template with a property block to define and assert verification conditions. SVAs are appended to the respective module and verified via simulation.

TABLE II: CWEs generated by ChatGPT-4 and GEMINI under various threat models for different SoC Configurations; Only the valid and relevant CWEs are marked in ***maroon***.

Config#	**Bus-Based Attacks**		**Side-Channel Attacks**		**DoS Attacks**		**Confidentiality**		**Access Control**	
	ChatGPT	*GEMINI*	*ChatGPT*	*GEMINI*	*ChatGPT*	*GEMINI*	*ChatGPT*	*GEMINI*	*ChatGPT*	*GEMINI*
Config 1	CWE-693	CWE-200	CWE-203	CWE-400	CWE-400	CWE-400	CWE-200	CWE-357	CWE-284	CWE-285
	CWE-203	CWE-900	CWE-200	CWE-201	CWE-770	CWE-399	CWE-312	CWE-201	CWE-285	CWE-642
	CWE-749	CWE-894	CWE-310	CWE-326	CWE-404	CWE-250	CWE-310	CWE-113	CWE-287	CWE-521
	CWE-200	CWE-201	CWE-385	CWE-116	CWE-406	CWE-690	CWE-522	CWE-380	CWE-724	CWE-306
	CWE-284	CWE-682	CWE-691	CWE-203	CWE-421	CWE-476	CWE-359	CWE-894	CWE-22	CWE-284
Config 2	CWE-352	CWE-264	CWE-203	CWE-273	CWE-400	CWE-399	CWE-200	CWE-264	CWE-284	CWE-272
	CWE-20	CWE-399	CWE-200	CWE-392	CWE-404	CWE-400	CWE-312	CWE-200	CWE-285	CWE-400
	CWE-200	CWE-400	CWE-310	CWE-201	CWE-770	CWE-422	CWE-319	CWE-392	CWE-287	CWE-284
	CWE-345	CWE-416	CWE-384	CWE-326	CWE-406	CWE-477	CWE-522	CWE-400	CWE-264	CWE-119
	CWE-284	CWE-200	CWE-385	CWE-611	CWE-768	CWE-779	CWE-310	CWE-476	CWE-22	CWE-611

Config 1: #Masters: 1, #Slaves: 11 (AES, DES, SHA, GPS, etc.), Bus: AXI4. **Config 2:** #Masters: 2, #Slaves: 9 (DES, SHA, FFT, UART, etc.), Bus: Wishbone.

TABLE III: A snapshot of the Extensive DB of CWEs used for filtering in SPELL.

CWE-ID	**Bug Description**	**Classification**	**Timing Requirements**	**Type of Violation**
CWE-284	Improper Access Control	Bus + IP Level	Synchronous	Access Control
CWE-120	Buffer Copy without Checking Size of Input	N/A	N/A	Inadequate Error Handling
CWE-1391	Use of Weak Credentials	IP Level	Synchronous	Access Control
CWE-330	Use of Insufficiently Random Values	IP Level	Asynchronous	Information Flow
CWE-362	Concurrent Execution using Shared Resource with Improper Synchronization	Bus Level	Synchronous	Liveness
CWE-367	TOCTOU Race Condition	Bus + IP Level	Synchronous	TOCTOU

D. Assertion to Security Policy & Policy Enforcement

SPELL employs an automated translation for each assertion activated during simulation to an equivalent security policy represented in a 3-tuple *<predicate, timing, action>* format. SPELL is equipped to handle both sequential events or static conditions in SVAs when generating the *'predicate'*. SPELL includes specific *clock* or *reset* value requirements in the *'timing'* tuple, including the operating mode, denoted by an integer value (e.g., user mode: 0, debug: 1, etc.). The *'action'* can be customized based on security requirements and defined by the user. SPELL interprets and appends the corresponding action by assigning the required values to observable signals. SPELL integrates DiSPEL framework [13] for the enforcement of security policies. A centralized security module is used to enforce bus-level policies, and IP-level policies are appended to their respective bus-level wrappers.

IV. Experimental Results

Our experimental setup uses the open-source framework from MIT-LL[2]. Synopsys DC and VCS was used for synthesis and simulation, respectively. We use a JSON-based template to represent SoC design specifications, as shown in Listing 1.

Using context-specific questions from SoC design specifications, we evaluated LLM's capability to generate relevant CWEs. We noted variations in LLM responses for the same SoC configurations under different assumptions.

Limitations of LLMs while identifying CWEs

i. Inconsistent ranking of CWEs.
ii. Incorrect mapping between CWE-ID & bug description.
iii. Generating out-of-context/irrelevant CWEs.
iv. Ambiguity on descriptive text for the same CWE-ID.
v. Mapping multiple issues to the same CWE-ID.
vi. Incomplete list of CWEs under different assumptions.

[2]https://github.com/mit-ll/CEP.git

```
"SoC_General":
{
    "NAME":"MIT-CEP", "BUS":"AXI4",
    "NO_OF_MASTERS":"1", "NO_OF_SLAVES":"11"
},
"BUS_INTERFACE":
{
    "INTERFACE_NAME":"Master/Slave", "#PORTS":"17",
    "SIGNAL_NAMES":"AWVALID,AWADDR,WDATA,,..."
},
"IP_1":
{
    "NAME":"AES", "TYPE":"Slave", "ADDRESS_RANGE":"0
    x9300:0x93FF", "PROTECTED_RANGE":"0x9314:0x932C"
}
```

Listing 1: SoC Design Specifications in JSON format

These limitations necessitate filtering to identify the relevant context-specific CWEs. Fig. 3 shows an example of how CWE generation varies under different assumptions along with relevant CWEs (marked in ***green***) and non-relevant CWEs (marked in ***red***).

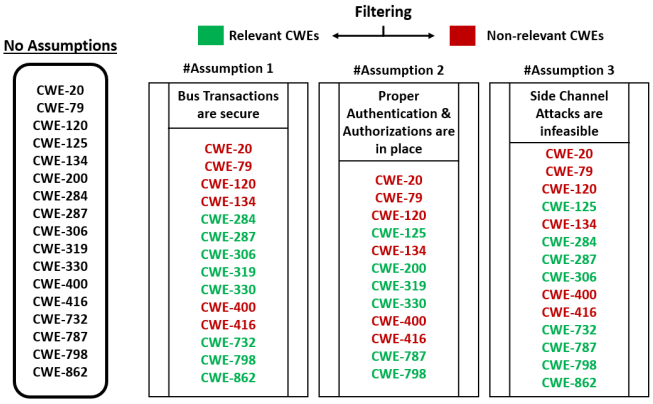

Fig. 3: Generating CWEs under different assumptions and identifying relevant/non-relevant CWEs using filtering.

We quantify the performance of an LLM for a given SoC

configuration by determining the number of valid CWEs ($\#CWE_{valid}$) and relevant CWEs ($\#CWE_{relevant}$) out of all the CWEs ($\#CWE_{total}$) in the generated response. A CWE is marked as *valid* if the CWE-ID and description generated by LLMs align with the Extensive DB, while a *relevant* CWE denotes that it is valid and related to the SoC context. The performance analysis of LLMs is presented in Fig. 4.

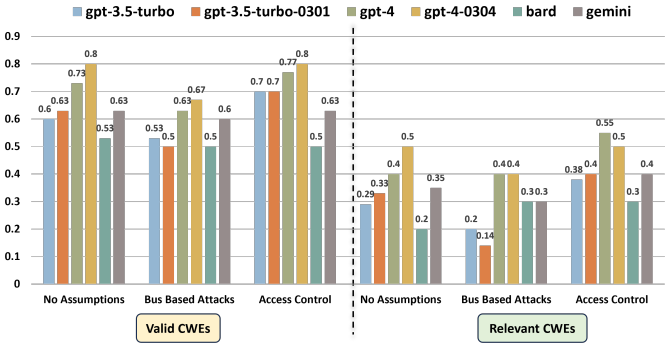

Fig. 4: Performance analysis of Conversational LLMs in generating valid and relevant CWEs for a given SoC configuration.

We have thoroughly tested LLM performance in generating SVAs for relevant CWEs. We note that the LLMs are proficient in generating context-relevant Verilog and SystemVerilog code but lack syntactic correctness and periodically deviate from actual requirements. SPELL automates the correction of SVAs based on design specifications, ensuring they are syntactically correct and pass verification. SPELL converts each activated SVA to the flexible 3-tuple security policy representation for addressing security requirements. Bus-level security policies are enforced through a centralized policy module, while IP-level policies are implemented via the bus-level wrappers at the respective IPs. Fig. 5 shows the conversion of an SVA to the 3-tuple policy followed by its enforcement.

The IPs were synthesized using 130nm SkyWater PDK after incorporating necessary code fixes using [13], as shown in Table IV with *CWE_ID#* column representing the identified CWE-IDs by SPELL. The results show that overheads are generally minimal under default synthesis settings without any constraints. In some cases, overheads decrease due to heuristic-based optimizations, making them negligible.

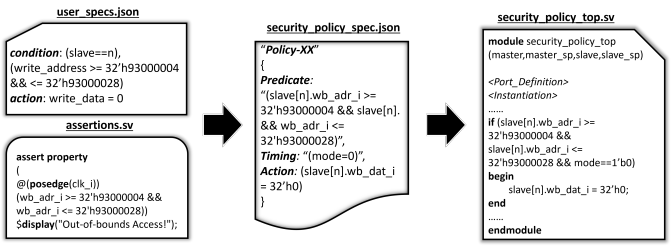

Fig. 5: Conversion of SystemVerilog Assertion to Security Policy followed by Policy Enforcement.

Discussion: Identifying relevant CWEs for a given SoC is challenging due to inaccuracies and lack of contextual listing in conversational LLMs. The updated ChatGPT models (v4.x) showed considerable improvement in generating and listing CWEs, although accuracy remained a primary concern. Google GEMINI's real-time web search also performs inadequately in generating relevant CWEs. Hence, a robust filtering step is essential. Further, incorporating domain-specific LLMs by fine-tuning general-purpose LLMs with a comprehensive dataset is expected to improve the performance significantly.

TABLE IV: Overhead analysis of different IPs after incorporating code fix for identified CWEs using SPELL.

Involved IP(s)	CWE_ID#	Synthesis Overheads (%)		
		Area	Power	Delay
μP, AES	319,327,330,787,798	0.16 ↑	−1.21 ↓	10.63 ↑
μP, DES3	125,319,327,522	1.16 ↑	0.29 ↑	0.21 ↑
μP, SHA256	200,319,522	1.33 ↑	1.49 ↑	0.45 ↑
μP, MD5	203,522	1.32 ↑	−2.17 ↓	−2.58 ↓
μP, RSA	200,787	0.05 ↑	0.11 ↑	0.8 ↑

V. Conclusion

We have presented a novel LLM-guided end-to-end secure SoC design framework, SPELL, which can automatically incorporate security measures in a target SoC design. SPELL incorporates filtering to identify relevant CWEs effectively for bus-based SoC configurations. It integrates SVA-based verification and automated translation from assertions to respective security policies followed by policy enforcement. Experiments show robust security against diverse CWEs is achieved with minimal hardware overhead. The fully automated process, from vulnerability identification to mitigation, can be integrated into commercial SoC design flow, reducing the manual work of security experts and increasing flexibility. While we focus on bus-based SoCs in this study, the approach can be extended to other fabrics, e.g., Network-on-Chip (NoC).

References

[1] G. Dessouky *et al.*, "HardFails: Insights into Software-Exploitable Hardware Bugs," in *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019, pp. 213–230.
[2] M. M. Bidmeshki, Y. Zhang, M. Zaman, L. Zhou, and Y. Makris, "Hunting Security Bugs in SoC Designs: Lessons Learned," *IEEE Design & Test*, vol. 38, no. 1, pp. 22–29, 2021.
[3] S. Thakur *et al.*, "VeriGen: A Large Language Model for Verilog Code Generation," 2023.
[4] M. Liu, N. Pinckney, B. Khailany, and H. Ren, "VerilogEval: Evaluating Large Language Models for Verilog Code Generation," 2023.
[5] J. Blocklove, S. Garg, R. Karri, and H. Pearce, "Chip-Chat: Challenges and Opportunities in Conversational Hardware Design," in *MLCAD*, 2023, pp. 1–6.
[6] X. Meng *et al.*, "Unlocking Hardware Security Assurance: The Potential of LLMs," 2023.
[7] R. Kande *et al.*, "LLM-assisted Generation of Hardware Assertions," 2023.
[8] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "Using LLMs to Facilitate Formal Verification of RTL," 2023.
[9] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing Hardware Security Bugs with Large Language Models," 2023.
[10] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt, "Examining zero-shot vulnerability repair with large language models," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2339–2356.
[11] B. Ahmad *et al.*, "Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design," in *ICCAD*, ser. ICCAD, 2022.
[12] M. Liu *et al.*, "ChipNeMo: Domain-Adapted LLMs for Chip Design," 2024.
[13] S. Paria, A. Dasgupta, and S. Bhunia, "DiSPEL: A Framework for SoC Security Policy Synthesis and Distributed Enforcement," in *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, 2024, pp. 271–281.