LASSO: LLM-Aided Security Property Generation for Assertion-based SoC Verification

Dinesh Reddy Ankireddy*, Sudipta Paria*, Aritra Dasgupta, Sandip Ray, and Swarup Bhunia
Department of Electrical and Computer Engineering, University of Florida, Gainesville, FL 32611, USA
{ankired.dineshre, sudiptaparia, aritradasgupta}@ufl.edu, {sandip, swarup}@ece.ufl.edu

2025 ACM/IEEE 7th Symposium on Machine Learning for CAD (MLCAD) | 979-8-3315-3762-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/MLCAD65511.2025.11189178

***Abstract*—Ensuring the security of modern System-on-Chip (SoC) designs poses significant challenges due to increasing complexity and distributed assets across the intellectual property (IP) blocks. Formal property verification (FPV) provides the capability to model and validate design behaviors through security properties with model checkers; however, current practices require significant manual efforts to create such properties, making them time-consuming, costly, and error-prone. The emergence of Large Language Models (LLMs) has showcased remarkable proficiency across diverse domains, including HDL code generation and verification tasks. Current LLM-based techniques often produce vacuous assertions and lack efficient prompt generation, comprehensive verification, and bug detection. This paper presents `LASSO`, a novel framework that leverages LLMs and retrieval-augmented generation (RAG) to produce non-vacuous security properties and SystemVerilog Assertions (SVA) from design specifications and related documentation for bus-based SoC designs. `LASSO` integrates commercial EDA tool for FPV to generate standard coverage metrics and iteratively refines prompts through a feedback loop to enhance coverage. The effectiveness of `LASSO` is validated through various open-source SoC designs, demonstrating high coverage values with an average of ∼88% denoting comprehensive verification through efficient generation of security properties and SVAs. `LASSO` also demonstrates bug detection capabilities, identifying five unique bugs in the buggy OpenTitan SoC from Hack@DAC'24 competition.**

***Index Terms*—SoC Security, Security Properties, SystemVerilog Assertion (SVA), Assertion Based Verification (ABV), Formal Property Verification (FPV), Large Language Models (LLMs).**

I. Introduction

Modern bus-based SoCs integrate multiple intellectual properties (IPs) on a single chip and utilize a common bus to facilitate communication between them. With the globalization of the IC supply chain and adoption of a zero-trust model, it becomes crucial to identify and fix vulnerabilities and protect secure assets in the regime of evolving security threats. Traditional verification methods, including simulation, random regression, directed-random testing [1], assertion-based formal verification [2], fuzzing [3], and hybrid techniques [4], etc., struggle to keep pace with the growing complexity of SoC designs, highlighting the need for greater automation in vulnerability detection and remediation. Formal Property Verification (FPV) provides a rigorous methodology for mathematically verifying hardware design correctness using model-checking. However, the effectiveness of FPV depends on meticulously crafted security properties that accurately capture the intended behaviors and uncover potential vulnerabilities. Assertion-Based Verification (ABV) uses assertions that are derived from its specification by the verification experts. These assertions are used to statically prove properties using formal verification tools or dynamically verified using simulation to identify potential vulnerabilities. Generating relevant security properties or assertions is a complex task involving substantial expertise and manual efforts by security experts, making it error-prone and not scalable for larger designs, highlighting the need for automation to streamline the verification process.

The rapid evolution of Large Language Models (LLMs) has extended their capabilities beyond natural language processing, making a significant impact in the domain of hardware security and verification. LLMs have shown remarkable proficiency in automating tasks like HDL code generation, verification, and bug fixing [5]. Recent studies on LLM-based generation of SystemVerilog Assertions (SVAs) [6]–[10], security properties [11], [12] have demonstrated the growing potential of LLMs in automating such verification tasks. LLM-based bug fixing [13], [14] aims to generate repairs for fixing security vulnerabilities involving static analysis and security-related feedback or policy-based enforcement [15]. However, these techniques are limited by the complexity of RTL designs and struggle to effectively incorporate diverse information sources, such as design specifications, threat models, and other security requirements, and lack in curating efficient prompts. Moreover, existing techniques lack vacuity checking, often resulting in the generation of vacuous or non-meaningful properties that hinder effective verification. These techniques also do not incorporate coverage analysis to assess the effectiveness of the generated properties, leaving ambiguity in terms of verification completeness. In this paper, we introduce a novel automated framework, `LASSO` (LLM-Aided Security Property Generation for Assertion-based SOC Verification), for efficiently generating security properties for comprehensive SoC verification and identifying bugs. This paper makes the following major contributions:

- We propose `LASSO`, a novel and efficient framework that leverages the knowledge base of LLMs to automatically generate security properties and SVAs, enabling comprehensive verification of generic bus-based SoC designs.
- `LASSO` integrates vacuity checking for discarding vacuous or non-meaningful properties to enhance verification

*Dinesh R. Ankireddy and Sudipta Paria contributed equally to this work.

979-8-3315-3762-3/25/$31.00 ©2025 IEEE

efficiency and reduce the computational overhead.
- LASSO employs standard FPV tools to perform coverage analysis and incorporates an iterative refinement step when coverage falls below a threshold, ensuring improved and comprehensive verification.
- Experimental results demonstrate the effectiveness of LASSO evaluated on open-source SoC benchmarks, achieving high coverage values that indicate substantial verification performance.
- LASSO is equipped with bug detection capabilities, as evidenced by the identification of five bugs in the buggy OpenTitan SoC from the Hack@DAC'24 competition.

The paper is organized as follows: Section II discusses the relevant background and related works. Section III presents the major stages of the proposed framework. Section IV demonstrates the experimental results and discussion. Finally, Section V concludes the paper.

II. Background

A. Security Property and Formal Property Verification

Security property is a formal specification or rule describing an observable behavior that the hardware design must satisfy to guarantee confidentiality, integrity, and availability (CIA) requirements [1]. These properties must adhere to three fundamental principles: Correctness, Consistency, and Completeness. Designers commonly use languages like PSL and SVAs, employing logic representations at the temporal level like LTL and CTL, to describe design behaviors. FPV is used to rigorously verify that security properties hold under all possible inputs. Commercial EDA tools employ model checkers to prove whether a property holds in all cases (safe) or find a counterexample (violation). By exhaustively proving or refuting properties, FPV enables the detection of security flaws that might evade traditional testing.

B. Vacuity Checking Rules/Theorems for Security Properties

Definition: Vacuity in LTL refers to a situation where a specification (formula) is satisfied in a system (model), but not in a meaningful way, as part of the formula was irrelevant in the system's behavior. Vacuous properties usually indicate weak, trivial, or incorrect specifications for a given system.

Example:

Let us assume the following formula/property in Model M:

$$\varphi = G(p \rightarrow Fq) \tag{1}$$

This states "Always, if p holds, then eventually q holds."

In model M, if p never occurs, then φ is vacuously true because the implication $p \rightarrow Fq$ when p is false. Hence, we can conclude φ is vacuously satisfied in M with respect to p.

Formal Definition: (from [16], [17])

A system M satisfies a formula φ vacuously iff $M \models \varphi$ and there is some subformula ψ of φ such that ψ does not affect φ in M. For example, verifying a system with respect to the specification $\varphi = AG(req \rightarrow AFgrant)$ ("every request is eventually followed by a grant"), we say that φ is satisfied vacuously in systems in which requests are never sent.

Nine theorems for vacuity checking [16], [17] are considered in our proposed framework (refer to Appendix A).

C. Coverage Metrics in FPV

Commercial tools performing FPV offer a range of coverage metrics to assess the thoroughness and completeness of FPV. Cadence JasperGold employs three coverage metrics, namely, stimuli, checker, and formal coverage. **Stimuli coverage** indicates how well the input conditions and scenarios are applied to the DUT during formal verification. It includes both code coverage (such as branch, statement, expression, and toggle coverage) and functional coverage (defined by user-specified covergroups). **Checker Coverage** assesses the completeness of the formal assertions in verifying the design behavior. It includes *Cone of Influence (COI)* Coverage, which measures the extent to which the logic paths that influence a given assertion are exercised, and *Proof Core* Coverage identifies the minimal set of design elements necessary for the assertion's truth value and ensures that these elements are thoroughly checked. **Formal Coverage** is a composite metric that combines both stimuli and checker coverage to provide an overall assessment of the effectiveness of formal verification.

D. Existing Verification Techniques and Challenges

The current industry practices predominantly employ two major methodologies for SoC verification: (i) simulation-based and (ii) assertion-based. Simulation-based techniques rely on generating complex testbenches that drive inputs and monitor outputs under various scenarios. Tools such as ModelSim and Synopsys VCS are commonly employed to validate the functional correctness of designs. Simulation-based approach is widely used for its familiarity and ease of deployment, but it offers limited coverage and struggles with scalability as design complexity increases. ABV offers a more formal and rigorous framework that employs SVAs specifying intended design behavior through properties and constraints and verified using FPV. It provides exhaustive verification within bounded scopes and is particularly effective in uncovering subtle logic errors/bugs or security vulnerabilities. However, it can be limited in scalability due to the state-explosion problem.

E. Related work on Leveraging LLMs in Verification

The growing use of LLMs is greatly advancing state-of-the-art hardware security verification [5] by leveraging their natural language understanding and broad knowledge base to perform diverse automation tasks. Recent research explores integrating LLMs into IC design flow to generate complex testbenches for simulation-based verification and creating precise, context-specific SVAs for formal verification. LLM-aided verification enables early detection of vulnerabilities prior to fabrication. LLM-based SVA generation [6], [7], [21] at the RTL abstraction level has been explored using natural language specifications and prompt engineering. Frameworks like AssertLLM [9] employ specialized LLMs for different stages of assertion creation, while hybrid methods [19] combine LLM-driven formal verification with mutation testing to refine design invariants. ChipNeMo [18] showcases versatile LLM applications, including chat-based assistance, script generation, and bug summarization, emphasizing domain-specific adaptation. DIVAS [15] provides an end-to-end toolflow that

TABLE I: Comparative analysis between LASSO and existing LLM-based verification approaches.

Proposed Solutions	Baseline	SVA Generation?	Property/Policy Generation?	Vacuity Checking?	Prompt Engineering?	RAG?	Coverage Analysis?	Bug Detection?	Evaluation Benchmarks
Kande et al. [7]	Codex	✓	✗	✗	✓	✗	✗	✗	Hack@DAC21, OpenTitan
NSPG [11]	BERT	✗	✓	✗	✓	✗	✗	✗	OpenTitan, RISC-V, MIPS
ChipNeMo [18]	LLaMA2 7B/13B/70B	✗	✗	✗	✗	✓	✗	✗	Custom Benchmarks
AssertLLM [9]	GPT-3.5, GPT-4	✓	✗	✗	✓	✓	✗	✗	Custom Benchmarks
Hassan et al. [19]	GPT-4	✓	✗	✗	✗	✗	✗	✗	C432 (ISCAS-85)
Saha et al. [20]	GPT-3.5, GPT-4	✓	✗	✗	✗	✗	✗	✗	CWE, Trust-Hub
Orenes-Vera et al. [6]	GPT-4	✓	✗	✗	✓	✗	✗	✗	RISC-V CVA6 Ariane
SPELL [8]	GPT-3.5, GPT-4, Gemini	✓	✓	✗	✓	✗	✗	✗	MIT-CEP SoC, CWE
LAAG RV [21]	GPT-4	✓	✗	✗	✓	✗	✗	✗	OpenTitan
FVEVAL [22]	GPT-4o, Gemini-1.5, Llama-3.1	✓	✗	✗	✓	✗	✗	✗	Custom Benchmarks
LASP [12]	Gemini-1.5	✓	✓	✗	✓	✗	✗	✗	RSA, DES, SHA512, AES
LASSO* (This work)	GPT-4o, Llama-3.1, Gemini-1.5	✓	✓	✓	✓	✓	✓	✓	CEP, OpenTitan, Hack@DAC'24

automates CWE identification, SVA generation, and security policy enforcement using [23], highlighting broad applications of LLMs. Additional frameworks such as FVEval [22], LASP [12], and NSPG [11] assess and generate security properties directly from RTL or design documentation. Table I summarizes the characteristics of existing solutions and also highlights the features of the proposed LASSO framework.

F. Motivation

Current techniques face significant challenges due to the increasing complexity of SoC designs. They also lack the capability to curate efficient prompts by extracting relevant information from the corresponding documentation or design specifications. Current techniques follow more targeted verification and also do not guarantee the generation of non-vacuous properties that contribute to effective verification. Furthermore, existing approaches do not incorporate coverage analysis to evaluate the effectiveness of the generated properties or assertions, leaving gaps in verification completeness and increasing the risk of undetected errors. These limitations highlight the need for a more efficient automated framework that overcomes shortcomings of current approaches and ensures comprehensive verification of complex SoC designs.

III. Methodology

In this section, we describe the main components of the LASSO framework, starting from prompt creation, followed by non-vacuous security properties generation and then conversion to SVAs leveraging LLMs, along with coverage analysis with iterative refinement and bug detection. The overall flow has been depicted in Fig. 1. The proposed framework can be categorized into five major stages as described below.

A. Prompt Generation

The prompt generation step includes the extraction of behavioral descriptions from the given RTL code and specification documents, including block diagrams and textual descriptions. Current LLMs are constrained by the number of input tokens they can process, making it difficult for them to effectively analyze large hardware designs that consist of thousands of lines of RTL code. LASSO breaks down the code into higher-level abstractions (e.g., modules, functions, or components) and uses the summarized descriptions in the creation of prompts. Additionally, LASSO employs Retrieval-Augmented Generation (RAG) techniques to extract and summarize the relevant information from related documentation or manuals at a high level. By combining the summarized design descriptions and retrieval-based information, LASSO generates more accurate, contextually aware prompts for generating the relevant security properties in the subsequent steps.

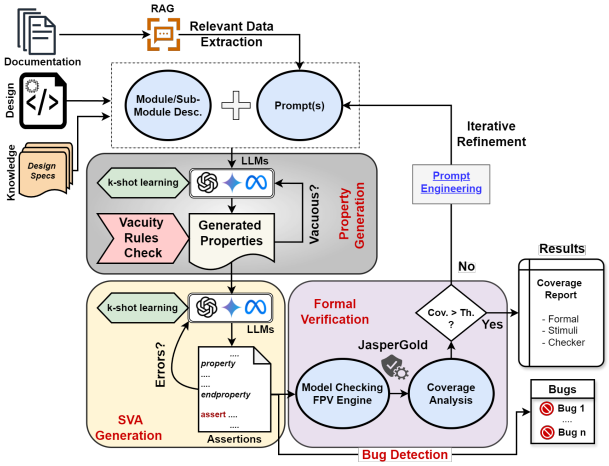

Fig. 1: Major stages in the proposed LASSO framework.

B. Security Property Generation

The proposed framework leverages pre-trained LLMs to generate relevant security properties for the SoC design under test. The security properties are typically expressed in temporal logic and explore specific or entire state-space of the design, exploring possible states and transitions. The framework also incorporates vacuous rules checking to improve property generation and discard trivial or non-interesting properties. We borrow the standard nine vacuous rules and theorems (described in Appendix A) from literature [16], [17] in evaluating the vacuous properties. Additionally, the framework integrates k-shot learning for generating valid security properties while reducing hallucinations and irrelevant outputs from LLMs.

C. SVA Generation

LASSO integrates pre-trained LLMs for translating non-vacuous security properties into equivalent SVAs. The framework employs re-prompting to correct syntax errors in generated assertions and produce revised versions. LASSO appends each SVA into the respective .sva file and automates the process of generating TCL scripts for evaluation using Cadence JasperGold. The evaluation employs FPV using model checkers to generate counterexamples (CEXs) to identify any failures that infer potential bugs or vulnerabilities. LASSO utilizes

k-shot learning by allowing the model to learn from a small set of example assertions to generate syntactically accurate and contextually relevant SVAs. This allows LASSO to adapt to diverse design specifications, producing high-quality assertions that align with the security goals and underlying threat model.

D. Coverage Analysis

Coverage analysis in FPV quantifies how effectively the verification process exercises the design under test and the generated SVAs derived from formal security properties. It provides quantitative metrics that help evaluate the completeness, effectiveness, and quality of the properties/assertions. LASSO integrates multiple coverage metrics: Formal, Stimuli, and Checker Coverage, leveraging the Cadence JasperGold Coverage tool to analyze branch, statement, expression, and toggle coverage. LASSO generates a comprehensive verification report summarizing assertion coverage and verification results. If any signals remain uncovered, further analysis can be performed by reviewing branch, statement, expression, toggle, and functional coverage reports.

E. Iterative Refinement

LASSO employs iterative refinement that enhances the existing prompts to improve the coverage by generating additional properties followed by SVAs until the pre-defined coverage threshold (%) is met. Failed SVAs indicate a potential issue, either due to incorrect formulation by LLM or uncovered design behavior that leads to potential bugs. The incorrect assertion generation by LLM necessitates prompt refinement through an iterative feedback path in LASSO to enhance the assertion generation process and ensure greater accuracy. Additionally, LASSO maintains additional prompts database and selectively applies relevant prompts aimed at generating more accurate properties, thereby enhancing coverage and improving overall verification completeness.

IV. Results and Discussion

A. Experimental set-up

We performed a comprehensive evaluation of our proposed LASSO framework on two popular open-source SoC benchmarks, namely, Common Evaluation Platform (CEP)[1] and OpenTitan[2]. The experiments were conducted on a Red Hat Enterprise Linux Server with AMD Epyc 7713 64-core processor and 1007.6 GiB of memory. FPV has been performed using Cadence JasperGold (ver. 2020.12). Our analysis utilizes diverse pre-trained LLMs, such as OpenAI's GPT-4o, Google's Gemini-1.5, and Meta's Llama-3.1, for generating properties and SVAs. We evaluated the performance of different pre-trained LLMs and found GPT-4o outperforms other pre-trained LLMs based on the percentage of generated security properties classified as #proved or #failed, averaged on multiple IPs (refer to Fig. 2). We selected GPT-4o as our primary LLM for integration with LASSO for automation tasks in different stages.

[1]https://github.com/mit-ll/CEP.git
[2]https://opentitan.org/book/hw/ip/index.html

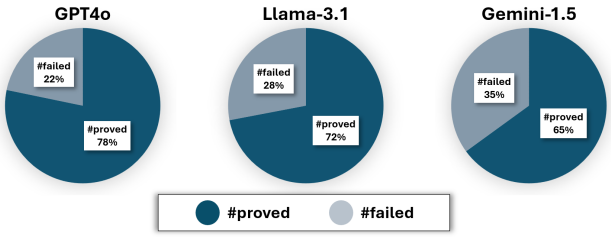

Fig. 2: Comparing LLMs in terms of generating properties.

B. Generation of Initial Prompts and RAG-based Extraction

LASSO adopts JSON-based template (<SPEC_FILE>) as shown partially in Listing 1 that formalize the specifications. For complex SoCs with multiple IPs involved, LASSO processes each IP individually and uses the corresponding IP-level documentation and <SPEC_FILE> to generate tailored prompts. For large hierarchical IPs, LASSO identifies the submodules based on design specifications and performs evaluation at the submodule level. The initial prompt includes high-level design specifications extracted from <SPEC_FILE>, either at the module or submodule level, augmented with contextual information retrieved via RAG-based techniques. These enriched prompts are fed sequentially to pre-trained LLMs to facilitate the generation of relevant security properties while mitigating the token limitations of LLMs. The example prompts are included in Appendix B.

```
"SoC_General":
{
    "NAME":"MIT-CEP",
    "TYPE":"Open-source",
    "BUS":"AXI4",
    "NO_OF_IP":"12",
    .... // more details
},
"IP_1":
{
    "NAME":"AES",
    "TYPE":"Slave",
    "OPERATION":"Crypto",
    .... // more details
}
"Assets":
{
    "NAME":"aes_key",
    "TYPE":"192-bit",
    .... // more details
}
```

Listing 1: Example SoC design specifications in JSON format.

C. Generation of Security Properties and Vacuity Check

LASSO employs vacuity check on the LLM-generated properties to discard all non-vacuous properties that violate at least one of the nine theorems (Section II-B). LASSO adopts a hierarchical analysis approach, treating each submodule independently to generate localized security properties, which are subsequently aggregated at the top module level to ensure comprehensive coverage and consistency. Table II presents the number of security properties generated for different IPs from CEP and OpenTitan benchmarks. The column '#correct' denotes the number of properties that are proven to be relevant and semantically correct, and '#non-vacuous' denotes the number of properties that pass vacuity check rules. Fig. 3 rep-

resents the number of proved and failed properties generated for each submodule of AES and FIR designs from CEP.

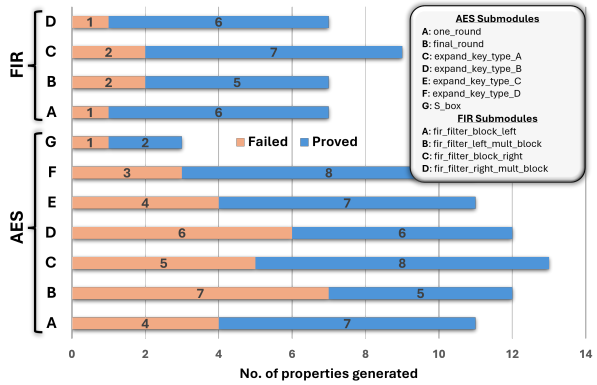

Fig. 3: Number of 'Proved' and 'Failed' properties generated by LASSO for AES and FIR submodules.

TABLE II: Number of generated non-vacuous properties and correct SVAs using LASSO.

Design	Properties			Assertions	
	#generated	#correct	#non-vacuous	#generated	#correct
AES-192▲	73	43	41	41	40
DES3▲	37	31	29	29	27
GPS▲	108	98	93	93	90
FIR▲	30	24	22	22	22
IIR▲	38	26	25	25	25
i2c★	78	75	71	71	70
adc_ctrl★	37	33	32	32	32
kmac★	49	47	44	44	42

▲denotes designs from CEP. ★denotes designs from OpenTitan.

D. Generation of Valid SVAs

LASSO leverages pre-trained LLMs to generate equivalent assertions for all non-vacuous properties that pass the vacuity check. We found most of the properties were directly translatable into SVAs (refer to Table II) while some posed challenges for direct conversion due to the dependence on the formulation of properties. There were some manual efforts involved for those properties requiring re-prompting for conversion to SVAs. LLMs often struggle with generating correct SVAs due to limited understanding of hardware behavior, signal timing, and implication semantics (e.g., overlapping vs. non-overlapping), etc. LLMs may generate SVAs that are syntactically valid but semantically incorrect, such as using non-boolean expressions in implication, incorrect signal scopes, or violating reset behavior constraints. LASSO integrates verification through Cadence JasperGold and discards the syntactically incorrect or semantically invalid SVAs.

Example of Vacuous Property/Assertion:

Listing 2 demonstrates an example property that violates one of the nine theorems of vacuity checking and will be treated as a vacuous property, and the corresponding assertion would also be discarded. This property is potentially vacuous since the antecedent `validCounter > 1` is never `true` if `validCounter` is stuck at 1 or 0.

Example of CEX:

Listing 3 illustrates a counterexample generated for an LLM-generated SVA for DES3 design. This condition violates the design behavior, and model checking in FPV can generate a counterexample for the same (refer to Fig. 4), making it semantically incorrect.

```
property p_validCounter_decrement;
  @(posedge clk) disable iff (rst)
  (validCounter > 1) |=> (validCounter == $past(
    validCounter) - 1);
endproperty
assert property (p_validCounter_decrement);
```

Listing 2: Example of Vacuous Property/Assertion.

```
property p_k_update;
  @(posedge clk) disable iff (reset)
  (!$stable(roundSel)) |=> (K !== $past(K));
endproperty
assert property (p_k_update);
```

Listing 3: Example of CEX for DES3 design.

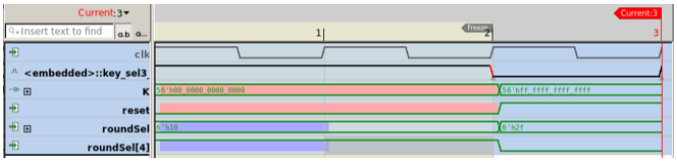

Fig. 4: Timing diagram depicting the counterexample (CEX).

E. Coverage Analysis

LASSO integrates Cadence JasperGold to perform FPV on the generated SVAs in the previous step and generate respective coverage metrics. LASSO follows a configurable coverage threshold (set to 80% for our experiments), representing the minimum coverage required for the verification to be considered *acceptable*. If coverage falls below this threshold, LASSO incorporates a feedback loop to generate additional relevant SVAs to improve the coverage. Higher coverage % indicates a more thorough and efficient verification, ensuring that all relevant states, behaviors, and properties of the design are fully explored and validated. LASSO can also be seamlessly integrated with other commercial EDA tool flows to perform FPV and generate respective coverage metrics. Table III presents the highest coverage values achieved for eight different IPs belonging to CEP and OpenTitan SoC. Coverage values at the submodule level are detailed in Appendix C.

TABLE III: Coverage values (%) for different IPs.

Design	Checker Coverage		Stimuli Coverage	Formal Coverage	
	COI	Proof Core		COI	Proof Core
AES-192▲	99.02%	83.59%	99.93%	99.37%	85.05%
DES3▲	93.97%	91.70%	100%	93.98%	91.71%
GPS▲	98.48%	83.39%	99.40%	98.63%	84.70%
FIR▲	92.64%	80.35%	92.82%	91.59%	80.50%
IIR▲	90.68%	81.07%	99.01%	90.97%	81.42%
i2c★	91.44%	84.40%	99.25%	91.44%	84.40%
adc_ctrl★	86.37%	80.53%	95.18%	86.12%	80.78%
kmac★	89.35%	81.53%	96.79%	90.22%	82.19%

▲denotes designs from CEP. ★denotes designs from OpenTitan.

F. Iterative Refinement

LASSO includes a feedback path involving prompt engineering to regenerate more non-vacuous properties and context-specific SVAs for further improvement of coverage metrics

TABLE IV: Detected bugs from Hack@DAC'24 buggy OpenTitan SoC using LASSO framework.

Bug#	Bug Description	Code Reference	Security impact	SVA for Detection
1	Incorrect parity checks for UART receiver. The rx_parity_err in uart_rx does not depend on parity_enable.	uart_rx.sv: Line: 102-103	It can lead to incorrect FIFO sync (prim_fifo_sync) and false hardware interrupt (intr_hw_rx_parity_err).	property p_parity_err_without_enable; @(posedge clk_i) disable iff (!rst_ni) (!parity_enable && rx_valid_q) |->!rx_parity_err; endproperty assert property(p_parity_err_without_enable);
2	Incorrect wr_data is assigned. Assigned q (reg value) instead of d (hw value).	prim_subreg_arb.sv: Line: 51	Incorrect assignment leads to non-clearance of hw value.	property p_w1s_hw_clear_should_use_d; @(posedge clk_i) disable iff (!rst_ni) (SwAccess == SwAccessW1S && de && !we) |->(wr_data === d); endproperty assert property(p_w1s_hw_clear_should_use_d);

if it falls below the threshold. Listing 4 shows some example prompts providing additional semantic context that are included with the existing prompts, refining the requirements, and generating relevant properties and SVAs. In this stage, some manual expertise might be required for correcting partially valid properties, resolving ambiguities, and incorporating design-specific knowledge that may not be captured through k-shot learning examples. Fig. 5 illustrates the iterative improvement on coverage metrics across five different designs from the CEP SoC benchmark. This iterative approach ensures that LASSO enhances formal coverage, leading to more robust and comprehensive verification outcomes.

```
prompt_database:
  "Enhance coverage by adding reset conditions."
  "Add corner case assertions for boundary values."
  "Include sequential behavior checks."
  "Cover unreachable states if any."
  "Extend assertions to multi-cycle paths."
```

Listing 4: Example prompts for iterative refinement.

G. Bug Detection

LASSO facilitates bug detection by generating context-aware SVAs that capture the security and functional properties of the design. For generating tailored SVAs for detecting bugs, LASSO includes additional high-level information into the prompts, including micro-architectural events, threat models, secure assets, etc. These improved prompts guide the language model to produce tailored SVAs that more accurately reflect the design's security objectives. When verified, these SVAs can expose violations that indicate design flaws. This capability is demonstrated through the detection of five bugs in the buggy OpenTitan SoC from the Hack@DAC'24 competition, with two examples tabulated in Table IV. The remaining bugs identified using LASSO are tabulated in Appendix D.

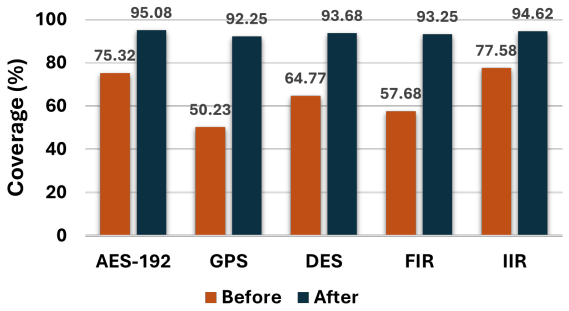

Fig. 5: Improvement on coverage values for different IPs through iterative refinement in LASSO.

H. Discussion

While LASSO demonstrates promising results, further enhancements are still possible. Incorporating more robust vacuity checking rules or refining existing ones could help filter out vacuous properties. LLMs struggle with complex, specialized tasks and are constrained by a finite set of use cases in generating SVAs, often leading to incomplete or erroneous responses. Fine-tuning LLMs on domain-specific datasets can significantly enhance their performance. Integrating syntax-checking tools helps minimize errors in generated responses and significantly reduces manual efforts. Enhancing LLMs with advanced reasoning capabilities and more sophisticated prompt engineering could improve automation tasks. Additionally, adopting custom coverage metrics would offer a more accurate representation of verification completeness, guiding iterative improvements effectively. Bug detection is limited by fine-grained analysis and specific design knowledge, making it less generalizable across various IPs, but can be improved using LLMs with enhanced reasoning for design analysis or integrating agentic framework with flexible verification flow.

V. Conclusion

In this paper, we presented LASSO, a novel efficient framework for automating security verification for generic bus-based SoCs. LASSO incorporates RAG-based implementation to extract relevant data from hardware documentation and formalize SoC specifications for generating effective LLM prompts. By leveraging pre-trained LLMs and combining vacuity checking and k-shot learning, LASSO efficiently generates contextually relevant non-vacuous security properties and then SVAs at both module and submodule levels. The experimental results demonstrate higher coverage values (avg. 88%) with iterative refinement, highlighting LASSO's effectiveness in comprehensive verification. LASSO also showcases bug detection capabilities, enabling the identification of potential vulnerabilities at the early stages of the design flow. Future work includes incorporating domain-adapted LLMs with fine-tuning to improve the performance and also extending to other SoC interconnect fabrics, e.g., Network-on-Chip (NoC).

Acknowledgment

This material is based upon work supported by the National Science Foundation (NSF) grant #2350365. The opinions, findings, conclusions, or recommendations expressed are those of the author(s) and do not necessarily reflect views of NSF.

REFERENCES

[1] S. Bhunia and M. Tehranipoor, *Hardware Security: A Hands-on Learning Approach*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2018.

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

[3] R. Saravanan, S. Paria, A. Dasgupta, V. N. Patnala, S. Bhunia, and S. M. P. D, "Synfuzz: Leveraging fuzzing of netlist to detect synthesis bugs," 2025.

[4] C. Chen, R. Kande, N. Nguyen, F. Andersen, A. Tyagi, A.-R. Sadeghi, and J. Rajendran, "Hypfuzz: formal-assisted processor fuzzing," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[5] S. Paria, A. Dasgupta, and S. Bhunia, "Navigating soc security landscape on llm-guided paths," in *Proceedings of the Great Lakes Symposium on VLSI 2024*, ser. GLSVLSI '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 252–257.

[6] M. Orenes-Vera, M. Martonosi, and D. Wentzlaff, "Using LLMs to Facilitate Formal Verification of RTL," 2023.

[7] R. Kande, H. Pearce, B. Tan, B. Dolan-Gavitt, S. Thakur, R. Karri, and J. Rajendran, "(Security) Assertions by Large Language Models," *IEEE Transactions on Information Forensics and Security*, pp. 1–1, 2024.

[8] S. Paria, A. Dasgupta, and S. Bhunia, "SPELL: An End-to-End Tool Flow for LLM-Guided Secure SoC Design for Embedded Systems," *IEEE Embedded Systems Letters*, vol. 16, no. 4, pp. 365–368, 2024.

[9] W. Fang, M. Li, M. Li, Z. Yan, S. Liu, H. Zhang, and Z. Xie, "AssertLLM: Generating and Evaluating Hardware Verification Assertions from Design Specifications via Multi-LLMs," 2024.

[10] S. Paria, A. Dasgupta, and S. Bhunia, "Towards automated verification of ip and cots: Leveraging llms in pre- and post-silicon stages," in *2025 IEEE 43rd VLSI Test Symposium (VTS)*, 2025, pp. 1–5.

[11] 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.

[12] A. Ayalasomayajula, R. Guo, J. Zhou, S. K. Saha, and F. Farahmandi, "LASP: LLM Assisted Security Property Generation for SoC Verification," in *2024 ACM/IEEE 6th Symposium on Machine Learning for CAD (MLCAD)*, 2024, pp. 1–7.

[13] 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.

[14] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, "Fixing Hardware Security Bugs with Large Language Models," 2023.

[15] S. Paria, A. Dasgupta, and S. Bhunia, "DIVAS: An LLM-based End-to-End Framework for SoC Security Analysis and Policy-based Protection," 2023.

[16] O. Kupferman and M. Y. Vardi, "Vacuity detection in temporal model checking," in *Correct Hardware Design and Verification Methods*, L. Pierre and T. Kropf, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 82–98.

[17] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh, "Efficient detection of vacuity in actl formulas," in *Computer Aided Verification*, O. Grumberg, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 279–290.

[18] M. Liu *et al.*, "ChipNeMo: Domain-Adapted LLMs for Chip Design," 2024.

[19] M. Hassan, S. Ahmadi-Pour, K. Qayyum, C. K. Jha, and R. Drechsler, "LLM-guided Formal Verification Coupled with Mutation Testing," 2024.

[20] D. Saha, S. Tarek, K. Yahyaei, S. K. Saha, J. Zhou, M. Tehranipoor, and F. Farahmandi, "LLM for SoC Security: A Paradigm Shift," 2023.

[21] K. Maddala, B. Mali, and C. Karfa, "Laag-rv: Llm assisted assertion generation for rtl design verification," 2024.

[22] M. Kang, M. Liu, G. B. Hamad, S. Suhaib, and H. Ren, "FVEval: Understanding Language Model Capabilities in Formal Verification of Digital Hardware," 2024.

[23] 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.

APPENDIX

A. Theorems for Vacuity Checking

We have utilized the following nine theorems from the literature [16], [17] to perform vacuity checking in LASSO and ensure that the generated properties are meaningful and non-trivially satisfied.

Theorem 1. (Efficient vacuity checking) For every formula φ, a subformula ψ of φ, and a system M, the following are equivalent:
(1) ψ does not affect φ in M.
(2) M satisfies $\varphi[\psi \leftarrow \textbf{\textit{true}}]$ iff M satisfies $\varphi[\psi \leftarrow \textbf{\textit{false}}]$.

Theorem 2. (Polynomial time complexity) The problem of checking whether a system M satisfies a formula φ vacuously can be solved in time $\mathcal{O}(C_M(|\varphi|) \cdot |\varphi|)$.

Theorem 3. (Complexity of checking vacuity in CTL) For φ in CTL,a subformula ψ of φ with multiple occurrences, and a system M, the problem of deciding whether ψ does not affect φ in M is co-NP-complete.

Theorem 4. (Linearly witnessable CTL formula) Given a CTL formula φ, deciding whether φ is linearly witnessable is in 2EXPTIME (Double Exponential Time or $\mathcal{O}(2^{2^{p(n)}})$, where $p(n)$ denotes polynomial function of n) and is EXPTIME(Exponential Time)-hard.

Theorem 5. (Linearly counterable formula) For a branching temporal logic formula φ, we have that φ is linearly counterable iff $\neg\varphi$ is linearly witnessable.

Theorem 6. (Branching temporal logic) For a branching temporal logic formula φ and a system M, we have that $M \not\models A\varphi^d$ iff M has a path π such that $\pi \not\models \varphi$.

Theorem 7. (Linearly Witnessable CTL* formula) For a CTL* formula φ and a system M, deciding whether φ is linearly witnessable in M is PSPACE-complete (polynomial amount of memory).

Theorem 8. (Counterexamples to find interesting witnesses) For a formula φ and a system M, a counterexample for $\neg witness(\varphi)$ in M is an interesting witness for φ in M.

Theorem 9. (Complexity of finding an interesting witness) For an LTL or a CTL* formula φ and a system M, an interesting witness for φ in M can be generated in polynomial space. Deciding whether such a witness exists is PSPACE-complete.

B. Example Prompts

Prompt 1 : Initial Prompt

You are an expert in Formal Verification and in generating SystemVerilog Assertions, specializing in writing comprehensive properties, including liveness, safety, and fairness constraints. You ensure thorough coverage by checking for deadlock, livelock, and starvation scenarios. To manage the state space explosion, you effectively apply constraints, symbolic tokens, and assumptions. You can also utilize auxiliary code and strive to write efficient, modular properties in place of overly complex assertions for the designs.

Prompt 2: Generating properties using design specs.

You need to generate relevant security properties for the hardware verification for module/submodule <NAME>. Here is the hardware design specifications <SPEC_FILE>. For each <CRITICAL_ASSET> in <SPEC_FILE>, extract the following information: 1. Signal name 2. Port Declearation:-Output, Input 3. Description: Definition, bit width, Signal type 4. Functionality 5. Any interconnects with other signals 6. Additional information required for assertions 7. Microarchitectural design. Additionally, consider the security assumptions or requirements, block diagrams, theory of operation, and all relevant information for module/submodule <NAME> from <SPEC_FILE> while generating the properties.

Prompt 3 : Vacuity checking rules/theorem

Given nine Rules <RULES.txt> Here are the 9 vacuity checking rules you need to memorize. Utilize these 9 theorems to generate valid and non-vacuous properties for the <module/submodule>.

Prompt 4: Vacuity checking

Can you evaluate and check if each property generated is non-vacuous? A property must fail at least one vacuity condition among the 9 theorems to be non-vacuous. Property is vacuous or non-vacuous using the 9 vacuity rules provided, and respond as if Non-Vacuous True else False.

Prompt 5: Generating CEXs

Can you help analyze the Counter Examples (CEXs) report from formal verification runs. Please analyze and explain the failure in the given CEXs and generate or modify the corresponding Assertions.

Prompt 6: Fixing SVA errors

There is an error in the properties generated by you. Check for Syntax or Semantics errors and correct them accordingly. Here is the error [ERROR (error code: VERI-1137)] <FILE.sva> syntax error near <LINE_NO>.

Prompt 7: Generating SVAs through iterative refinement

Based on this coverage report <statement, toggle, expression, branch> analysis, so the following cases <FAILED_CASES> are not covered, can you generate more properties to cover all the cases.

Prompt 8: k-shot learning example.

Generate the SVAs using the following template:
property <property_name>;
@(<clocking_event>) <antecedent_expression> |-> <consequent_expression>;
endproperty
assert property (<property_name>);
Example: Reset behavior - preventing data leakage by forcing secure_data to zero whenever reset is asserted.
property secure_data_cleared_on_reset;
@(posedge clk) (rst_n) |-> (secure_data == 0);
endproperty
assert property (secure_data_cleared_on_reset);

C. Coverage analysis at the submodule level

Coverage analysis at the submodule level involves assessing the checker, stimuli, and formal coverage generated using Cadence JasperGold integrated with LASSO framework. This approach enhances the overall verification efforts, allowing for a more granular exploration of design behavior at the submodule level and ensuring greater accuracy in identifying potential issues. Additionally, this approach also enhances scalability by enabling targeted testing of submodules, which can be independently verified and then aggregated for comprehensive verification of the overall design. Table V presents the coverage metrics at the submodule level for various IPs from the OpenTitan and CEP SoC benchmarks. The results highlight high coverage values for the generated properties using the LASSO framework, demonstrating the effectiveness of the proposed approach.

D. Additional bugs detected using LASSO

As described in Section IV-G, LASSO is equipped with bug detection capabilities by generating properties aimed at discovering bugs in different IPs in buggy OpenTitan SoC sourced from Hack@DAC'24 competition. We have identified five unique bugs across different IPs, two of which are tabulated in Table IV and the remaining three are presented in Table VI.

TABLE V: Coverage values for different IPs at the submodule level.

Benchmarks	Sub module	LOC♦	Checker Coverage Settings		Stimuli Coverage	Formal Coverage	
			COI	Proof Core		COI	Proof Core
AES-192▲	AES_192_top	145	97.94%	81.25%	99.56%	97.52%	85.50%
	Expand_key_type_A	47	95.26%	81.23%	100%	98.16%	83.16%
	Expand_key_type_B	35	100%	81.35%	100%	100%	81.44%
	Expand_key_type_C	44	100%	81.20%	100%	100%	82.35%
	Expand_key_type_D	41	100%	96.25%	100%	100%	96.25%
	final_round	35	100%	80.23%	100%	100%	81.60%
DES3▲	des3_top	172	99.82%	93.03%	100%	99.82%	93.03%
	CRP	36	99.34%	99.34%	100%	99.34%	99.34%
	key_sel3	860	82.74%	82.72%	100%	82.78%	82.75%
GPS▲	gps_top	146	100%	85.34%	100%	100%	83.67%
	AES_192_top	145	97.94%	81.25%	99.56%	97.52%	85.50%
	Expand_key_type_A	47	95.26%	81.23%	100%	98.16%	83.16%
	Expand_key_type_B	35	100%	81.35%	100%	100%	81.44%
	Expand_key_type_C	44	100%	81.20%	100%	100%	82.35%
	Expand_key_type_D	41	100%	96.25%	100%	100%	96.25%
	final_round	35	100%	80.23%	100%	100%	81.60%
	pcode	254	94.62%	80.86%	97.01%	94.78%	82.54%
	cacode	115	100%	84.75%	99%	98.61%	84.75%
FIR▲	fir_top	202	90.58%	87.56%	94%	90.60%	87.89%
	fir_filter_block_left	73	96.89%	83.56%	89.23%	95.54%	83.73%
	fir_filter_block_left_multiply block	88	92.85%	71.65%	88.94%	90.23%	71.78%
	fir_filter_block_right	38	94.30%	86.70%	96.24%	94.35%	86.82%
	fir_filter_block_right_multiply block	24	88.56%	72.29%	95.71%	87.25%	72.29%
IIR▲	iir_top	202	91.78%	88.54%	98.65%	92.62%	89.34%
	iir_filter_block_left	73	94.65%	80.03%	100%	94.65%	80.03%
	iir_filter_block_left_multiply block	88	89.02%	75.18%	99%	89.27%	75.58%
	iir_filter_block_right	64	87.89%	72.76%	98%	88.24%	73.32%
	iir_filter_block_right_multiply block	82	90.05%	88.82%	100%	90.05%	88.82%
i2c★	bus_monitor	296	91.20%	81.40%	98.20%	91.20%	81.40%
	controller_fsm	968	92.65%	84.40%	99.54%	92.65%	84.40%
	target_fsm	1021	90.47%	87.40%	100%	90.47%	87.40%
adc_ctrl★	adc_ctrl_fsm	384	85.58%	79.63%	93.83%	85.02%	79.18%
	adc_ctrl_intr	131	90.02%	82.08%	94.86%	89.58%	82.72%
	adc_ctrl_core	212	83.52%	79.89%	96.84%	83.76%	80.45%
kmac★	keccak_round	563	90.52%	85.33%	99.56%	90.52%	85.50%
	sha3_top	521	86.57%	78.23%	95.42%	87.16%	79.45%
	kmac core	463	87.54%	81.35%	97.45%	89.43%	81.44%
	kmac_entropy	744	92.76%	81.20%	94.72%	93.76%	82.35%
Average			93.96%	83.08%	98.04%	94.02%	83.70%

▲denotes designs from CEP SoC. ★denotes designs from OpenTitan SoC. ♦LOC: Lines of Code in RTL.

TABLE VI: Addition three bugs detected using LASSO in OpenTitan SoC from Hack@DAC'24 competition.

Bug#	Bug Description	Location or code reference	Security Impact	SVA for Detection
3	Potential OTP word Overflow	otp_ctrl_lci.sv: Line: 67	Overflow will cause system interruption by generating wrong values.	property p_otp_word_check; @(posedge clk_i) disable iff (!rst_ni) (LastLcOtpWord != LastLcOtpWordInt[CntWidth-1:0]); endproperty assert property(p_otp_word_check);
4	HMAC hashing key leaked through reg_rdata_next	hmac_reg_top.sv: Line: 1267-1273; 1343-1345	Malicious attempt to leak the HMAC hashing key.	property p_hmac_key_read_blocked; @(posedge clk_i) disable iff (!rst_ni) (addr_hit[8] || addr_hit[9]) |-> reg_rdata_next == '0; endproperty assert property(p_hmac_key_read_blocked);
5	Incorrect error detection logic	prim_subreg_shadow.sv: Line: 76-66; 184-185	Since error_s is not used hence the error detection logic will not work correctly leading to incorrect operation.	property p_error_s_known; @(posedge clk_i) disable iff (!rst_ni) $isunknown(error_s) == 0; endproperty assert property(p_error_s_known);

ARTIFACT APPENDIX

A. Abstract

This appendix describes the artifacts associated with the proposed novel framework LASSO for generating non-vacuous security properties, followed by SystemVerilog Assertions (SVAs) for comprehensive SoC verification. The code, data, and related instructions presented in this artifact are available on GitHub (Link: https://github.com/sudipta-23/lasso).

B. Artifact check-list (meta-information)

- **Algorithm:** The algorithm involves the creation of prompts followed by the generation of non-vacuous security properties and SVAs leveraging LLMs and coverage analysis using Cadence JasperGold with iterative refinement step if the coverage falls below the specified threshold.
- **Compilation:** Python compiler.
- **Model:** OpenAI GPT-4o (pre-trained).
- **Data set:** MIT-CEP, OpenTitan.
- **Run-time environment:** RedHat Enterprise Linux Server 7.9, Python 3.8 or above, and required packages specified in the requirements. (See README.md for more details)
- **Hardware:** CPU (AMD Epyc 7713 64-core), GPU (Nvidia RTX A6000), Memory (1007.6 GB).
- **Execution:** The toolflow parses the design specifications from the given JSON file and also related documentation (if available), and also identifies the submodules for the specified top module of a design. It appends the LLM-generated SVAs into the respective modules and automatically generates Tcl scripts followed by performing coverage analysis using JasperGold.
- **Metrics:** Three coverage metrics: stimuli, checker, and formal coverage generated using Cadence JasperGold.
- **Output:** LLM-generated SVAs and coverage report generated by Cadence JasperGold.
- **Experiments:** The tooflow can be invoked using relevant commands along with required and/or optional arguments (refer to README.md file). The GitHub repository includes an example script for running toolflow and generating the results.
- **Proprietary EDA tools:** Cadence JasperGold, Synopsys VCS.
- **How much disk space required (approximately)?:** Approx. 32 GB or above.
- **How much time is needed to prepare workflow (approximately)?:** Approx. 30-45 minutes.
- **How much time is needed to complete experiments (approximately)?:** Varies with design size, typically ranging from minutes to one or more hours.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache-2.0
- **Data licenses (if publicly available)?:** Apache-2.0
- **Archived (provide DOI)?:** No.

C. Description

1) How to access: The repository, including the code, related data, and example scripts, can be accessed on GitHub (Link: https://github.com/sudipta-23/lasso). The detailed instructions for setup and usage are provided in the README.md file.

2) Hardware dependencies: CPU / Cores: Multi-core (16 or above) x86_64 processor; Memory: 32 GB or above; OS: RHEL 7.9 or newer; GPU: Nvidia RTX A6000 (optional).

3) Software dependencies: Python 3.8 or above; python packages specified in the requirements file.

4) Commercial software dependencies: Cadence JasperGold (ver. 2020.12 FCS 64 bits for Linux) and Synopsys VCS (ver. W-2024.09-SP1) with respective valid licenses.

5) Data sets: Open-source SoC benchmarks:
- MIT-CEP (https://github.com/mit-ll/CEP)
- OpenTitan (https://opentitan.org/book/hw/ip/index.html).

The toolflow can support any open-source hardware designs written in Verilog or SystemVerilog (some SV constructs might not be supported).

6) Models: Pre-trained GPT-4o with access to the OpenAI API Key for interacting via API calls. No additional fine-tuning or customization of the model was performed. Tested with other pre-trained LLMs such as Llama-3.1, Gemini-1.5, and is flexible to support them through respective modifications in the configuration file.

D. Installation

The README.md file available in the GitHub repository provides detailed installation instructions. These include setting up the Python environment, installing dependencies, and updating the configuration JSON file.

E. Experiment workflow

It is recommended to follow the instructions provided in the README.md. The following steps should be followed:

- Setting up the Python environment
- Installing dependencies (requirements.txt)
- Updating the configuration file (config.json)
- Providing the design specifications in design_specs.json file and/or document file (optional)
- Executing the sample run script (run.sh) with required arguments

F. Evaluation and expected results

Evaluation results are provided in the paper, demonstrating the performance of the proposed toolflow in performing comprehensive formal verification with high coverage values across diverse designs.

G. Experiment customization

The experiment can be run via the command line with the appropriate arguments or through shell scripts (example, run.sh). Users can customize the experiment by adjusting optional parameters in the run script and ensuring correct directory and file paths are specified in the arguments.

H. Notes

For detailed and complete instructions, please refer to README.md in the artifact evaluation code repository on GitHub. See 'Disclaimer' and 'Common issues' sections for details on assumptions, limitations, and known issues.

I. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- http://cTuning.org/ae/submission-20201122.html
- https://github.com/ml-eda/artifact-evaluation/