

LASHED: LLMs And Static Hardware Analysis for Early Detection of RTL Bugs

Baleegh Ahmad
NYU Tandon
Brooklyn, USA
ba1283@nyu.edu

Hammond Pearce
University of New South Wales
Sydney, Australia
hammond.pearce@unsw.edu.au

Ramesh Karri
NYU Tandon
Brooklyn, USA
rkarri@nyu.edu

Benjamin Tan
University of Calgary
Calgary, Canada
benjamin.tan1@ucalgary.ca

Abstract—While static analysis is useful in detecting early-stage hardware security bugs, its efficacy is limited because it requires information to form checks and is often unable to explain the security impact of a detected vulnerability. Large Language Models can be useful in filling these gaps by identifying relevant assets, removing false violations flagged by static analysis tools, and explaining the reported violations. LASHED combines the two approaches (LLMs and Static Analysis) to overcome each other’s limitations for hardware security bug detection. We investigate our approach on four open-source SoCs for five Common Weakness Enumerations (CWEs) and present strategies for improvement with better prompt engineering. We find that 87.5% of instances flagged by our recommended scheme are plausible CWEs. In-context learning and asking the model to ‘think again’ improves LASHED’s precision.

Index Terms—LLMs, Static Analysis, Security, Bug Detection, CWE

I. INTRODUCTION

Security vulnerabilities in hardware are difficult to detect [1]. It is critical to identify them early on in the system-on-chip (SoC) design life-cycle because of the higher costs of fixing issues downstream (pre-silicon) or even recalls (post-silicon) [2]. An exhaustive search for these defects is not possible because of the high complexity of modern processors and SoCs. Therefore, there is a need for innovative solutions that provide **early-stage** information on potential security issues at the Register-Transfer Level (RTL).

Existing strategies for security verification include simulation with test benches, formal assertions [3], [4], hardware fuzzing [5], [6] and information flow tracking [7], [8]. Security checks “as-you-go,” while implementation is ongoing, are more challenging. Recent works have proposed static analysis [9], [10] and large language models (LLMs) [11]–[13] for this purpose. Static analysis checks source code without “executing” it, e.g., without simulating the design to perform directed tests. The code is instead checked against a set of coding patterns that can indicate undesirable behavior. Linters [14], [15] and formal verification tools [16], [17] are the two most commonly used static analysis [9], [18] methods for RTL. Linting is the automated checking of source code for stylistic, structural, design and programmatic checks [19], and can include data-flow analysis and control-flow analysis, as well as more abstract techniques such as pattern matching for bug-

specific heuristic patterns. Formal methods use mathematical models to analyze and verify a design [20].

Development and use of LLMs [21] has provided a possible means to detect bugs in code without the explicit need for a fully mature testing framework. LLMs have been used for RTL generation [22] and repair [23] with reasonable degrees of success, but their ability to **detect** security bugs has yet to be proven. In part, this is because LLMs do not verify their outputs. Static analysis can address this shortcoming. Prior work in the software space has explored the combination of LLMs and static analysis; for example, IRIS [24] uses CodeQL as the static analysis tool coupled with LLMs to detect the code injection vulnerabilities in Java code. Chapman et al. present an approach to interleave LLMs with the EESI static analysis tool to detect the issue of error-specific inference [25]. LLIFT [26] uses the LLM and static analysis combination to detect use before initialization bugs within the Linux kernel.

Taking inspiration from such works, we propose a strategy which uses LLMs and Static Analysis together, i.e., LASHED. We use hardware Common Weakness Enumerations (CWEs) [27], which provide examples of vulnerability categories to aid the generalizability of our approach. We use the LLM for three tasks: i) identifying security relevant assets, ii) removing false positives from static analysis, and iii) explaining the security issue posed by a reported violation. To identify security assets, the LLM uses the RTL source code and Hardware CWEs. The hardware static analysis tools formulate checks that could indicate the presence of certain CWEs. We use either linting or formal property verification for static analysis, depending on the nature of the CWE. The linting violations and failing assertions are returned to the LLM to prune out those that do not pose any security threat and provide explanations for the ones that do. Our contributions are:

- We present the first framework that combines Static Analysis and LLMs to detect security issues in RTL code. The details of this tool are described in Section II.
- We validate LASHED on four open source SoCs, described in Section III. Results are presented in Section IV.
- We investigate the impact of in-context learning and prompting to insist that the LLM reason through its assessments. The outcomes are analyzed in Section IV-A.

II. LASHED

LASHED takes RTL source and CWE information as inputs and outputs potential issues pertaining to the CWE. The CWEs covered are detailed in Section III-B. The output contains the bug code, its explanation, and its location. The framework for our approach is shown in Fig. 1. It can be broken down into three main steps, i.e., **Assets Identification**, **Static Analysis**, and **Contextualization**. We illustrate these steps through two motivating examples taken from the Hack@DAC 2021 buggy OpenPiton SoC shown in Fig. 2 and Fig. 3.

The vulnerability in Fig. 2(a) shows an instance of CWE 1191 where only the least significant 32 bits of the secret message are used to authenticate the JTAG access control module. This makes the access control susceptible to brute-force attacks. The vulnerability in Fig. 3(a) shows an instance of CWE 1233 where security-sensitive registers are missing lock bit protection in the Direct Memory Access (DMA) wrapper module. An adversary can modify them from software.

A. Assets Identification (AssetID)

We use an LLM to identify assets relevant to the CWE that LASHED is scanning for. We prompt the LLM with a system prompt that primes it as a hardware security expert searching for the CWE. This is followed by the description of the CWE from the MITRE website. There are numerous ways to structure the prompt and alter the included information which we investigate (see Section III-C). For instance, for prompt strategy *v1* the system prompt is followed by an example of the CWE in Verilog code derived from MITRE website, including identifying relevant assets and explaining why this example poses the CWE. After priming the LLM with the system prompt, the LLM is given a user prompt that contains the RTL source and instructions on what kinds of signals are to be identified. The identified assets are sent for static analysis. The components of the prompt are illustrated below:

```
system_prompt = "You are a hardware security expert. Your
task is to analyze Verilog code for potential CWE-<x>
bugs. CWE-<x> is <description of CWE>."

user_prompt = "What are the <relevant signals>? <Typical
nature of such signals>" + <RTL source code>."

if variation == 'v1':
    user_prompt = "<example of CWE in RTL> + <explanation
of security issue> + <assets identified in this
case>" + user_prompt
```

Asset identification of the motivating examples is shown in Fig. 2(b) and Fig. 3(b). The relevant access control signals for CWE-1191 example {`pass_data`, `data_d`} are correctly identified along with unrelated signals. For CWE-1233, the relevant security sensitive signals {`start_reg` ... `core_lock_reg`} and their “expected” lock signals {`reglk_ctrl[*]`} are identified. The reset conditions are also identified for assertion formation in the next step.

B. Static Analysis

Depending on the CWE, we decide whether a linting-based or assertion-based strategy is more appropriate. If the CWE

is typically present alongside some “structural” or “coding style” imperfections, we use lint checks. This is the case for CWEs 1191 and 1300. Conversely, if the CWE requires verifying whether a signal behaves appropriately depending on the value(s) of other signal(s) or if one signal flows to another, we use assertions. This applies to CWEs 1231, 1233 and 1244.

1) *Linting-based*: We select relevant checks from the ~1000 VC SpyGlass Lint [14] tags in the functional lint tag database based on our understanding of the CWE. For some CWEs, we develop custom lint checks using Verific [28], where, after obtaining the Abstract Syntax Tree (AST) of any module, we traverse it to check for some structural or stylistic element. If the checks’ results include the assets identified previously, the results are considered *violations*. These are sent to the next step, i.e., contextualization.

For CWE-1191, improper access control for debug occurs when a signal containing the user input (password) is not properly assigned a value. We check the code for the following: [*Width mismatch, Reverse Connected busses, Improper range index, Concatenation in array assign, Concatenation using unsized numbers, RHS has concatenation*] – if the assignment contains the previously identified access control signals, the signal and assignment are reported as a violation. Fig. 2(c) shows reported violations for the CWE-1191 example.

For CWE-1300, we check for design structural elements that may cause vulnerability to side-channel attacks: [*If without else, Inferred Latches*]. If the code has these, we check whether the conditional statement contains any previously identified assets. The presence of these in a conditional statement without an `else` can result in information leaking.

2) *Assertion-based*: We develop a custom template for System Verilog Assertions (SVAs) for selected CWEs and populate the template with information from AssetID. The formal tool verifies these SVAs for the RTL code, and *falsified* assertions are sent to the next step, i.e., contextualization. We use VC Formal Property Verification (FPV) [29] for this.

For CWE 1231, we check for a signal containing lock bits that are modifiable when they should not be. From AssetID, we obtain the lock signal and the conditions under which it should be modified (the negation captures conditions when it should not be modifiable). For each lock signal and the corresponding conditions, we form the following template and populate it with the appropriate information:

```
@([CLK_SENSE] [CLK])
[CONDITIONS_FOR_STABLE_LOCK] => $stable([LOCK_SIGNAL]);
```

For CWE 1233, we check for a security-sensitive signal that is missing lock bit protection. From AssetID, we obtain the security-sensitive register that should be protected, the lock signal that should be protecting it, and the reset conditions under which this protection mechanism is not applicable. For each lock signal, we form and populate the following template:

```
@([CLK_SENSE] [CLK])
disable iff ([RESET_CONDITIONS]) [LOCK_SIGNAL] == '1 | =>
    $stable([SECURITY_SENSITIVE_REGISTER]);
```

Fig. 3(c) captures an example of this assertion formation.

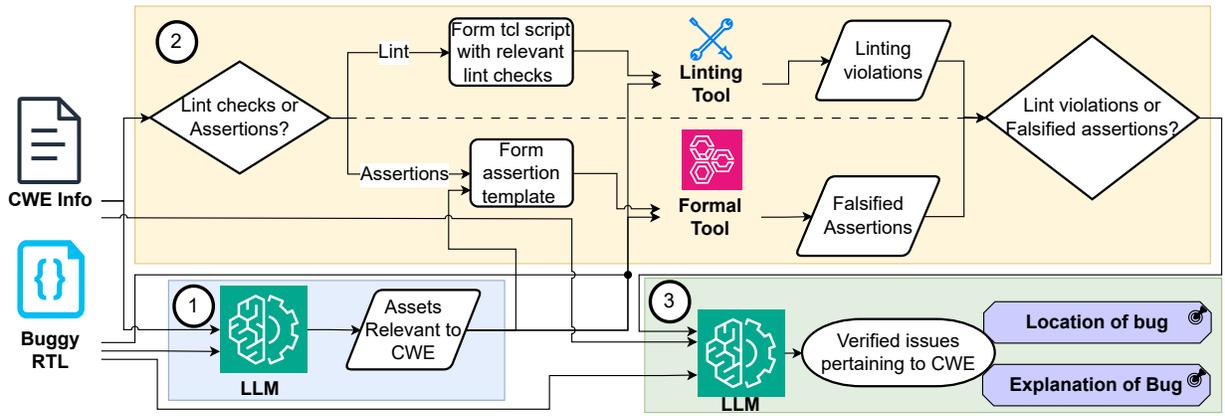


Fig. 1. LASHED framework. **1) Assets Identification (AssetID):** LLM identifies assets relevant to a CWE in the RTL. **2) Static Analysis:** Depending on CWE, a linting or assertion-based strategy is used. A linter or formal property verification tool checks for the lint tags identified or assertions formed. **3) Contextualization:** LLM reasons and prunes linting violations or falsified properties. LLM gives an explanation if there is a security issue.

```

1 if (pass_mode) begin
2   pass_data = { {60{8'h00}}, data_d };
3   state_d = PassChk;
4   pass_mode = 1'b0;

```

(a) CWE 1191 in JTAG access control mechanism. Only the least significant 32 bits of the secret message are used for authentication.

```

"access_control_related_signals": [ "pass_data",
  "pass_hash", "exp_hash", "hmac_ready", "data_d"
  ... ],

```

(b) Assets identification using gpt-4o-mini, prompt variation v0.

```

"relevant_violations": [ {
  "line_no": 158,
  "statement": "pass_data = {{60{8'b0}}}.data_d"; },
  "lhsexpr": "pass_data",
  "security_sensitive_signal": "pass_data" } ]

```

(c) Lint violation using VC SpyGlass Lint and Verific.

```

"insecure": true,
"explanation": The assignment to 'pass_data' defines a 2
56-bit wide data signal (with 60 zeros prepended to
'data_d'), ... If unauthorized access is not
controlled before 'pass_data' is evaluated or used,
an attacker could send malicious data to the device
and compromise intended security operations, making
it a potential violation of proper access control
in line with CWE 1191.

```

(d) Contextualization using gpt-4o-mini variation v0.

Fig. 2. Motivating example for CWE 1191 to illustrate LASHED flow for linter-based strategy.

For CWE 1244, we check for a privilege level signal that is escalated under conditions it should not have been. From AssetID, we obtain the privilege level signal, the correct conditions under which it should be escalated, the reset conditions of the module, the higher privilege level, and the signal containing the previous privilege level. A negation of

```

1 input logic [7 :0] reglk_ctrl_i; // register lock values
2 ...
3   else if(en && we)
4     case(address[7:3])
5       0: start_reg <= wdata;
6       ...
7       8: end_reg <= wdata;

```

(a) CWE 1233 in Direct Memory Access wrapper. Security sensitive registers are missing lock bit protection.

```

"relevant-signals": {
  "security_sensitive_signals_info": [
    {"lock_signal": "reglk_ctrl_i[0]",
     "security_sensitive_signal": "start_reg"},
    ...
    {"lock_signal": "reglk_ctrl_i[7]",
     "security_sensitive_signal": "core_lock_reg"},
    "reset_conditions": "~(rst_ni && ~rst_8)", ... }

```

(b) Assets identification using gpt-4o-mini, prompt variation v0.

```

@(posedge clk_i)
disable iff (~(rst_ni && ~rst_8)) reglk_ctrl_i[7] == '1
|=> $stable(core_lock_reg);

```

(c) Assertion formation for core_lock_reg signal in DMA wrapper using VC Formal Property Verification. This assertion was falsified.

```

"insecure": true,
"explanation": The signal 'core_lock_reg' is managed
but can be set to zero inadvertently, thereby
allowing modification of the system's important
registers. It lacks the necessary stability to
prevent unauthorized access.

```

(d) Contextualization using gpt-4o-mini variation v0.

Fig. 3. Motivating example for CWE 1233 to illustrate LASHED flow for assertion based strategy.

escalation conditions provides the conditions under which the privilege signal should not be escalated. For this privilege

signal, we form and populate the following assertion template:

```
@([CLK_SENSE] [CLK])
disable iff ([RESET_CONDITIONS])
~([CONDITIONS_FOR_PRIVILEGE_ESCALATION] !=>
  ([PRIVILEGE_SIGNAL] != [HIGH_PRIVILEGE] ||
  [PRIVILEGE_SIGNAL] == [PREVIOUS_PRIVILEGE] );
```

C. Contextualization

First, the LLM reasons whether the reported linting violations or falsified properties pose a security issue pertinent to the CWE under consideration. If it reasons that there is a security issue, the LLM is prompted to explain why. The explanation and static analysis violation are the final outputs of LASHED provided to the RTL designer. The components of the prompt to the LLM are illustrated below:

```
system_prompt = <same as Assets Identificaiton >
user_prompt = "Consider the following Verilog code: <RTL
source code> For each of the <static analysis outputs>,
determine whether the <output> poses a security issue
pertaining to CWE-<x> and provide an explanation if
that is the case. If the violation does not pose a
security issue, no explanation is needed. Here is the
output <output from Static Analysis>."
```

For experiments where we adopt prompt strategy v2 (details in Section III-C), we use the response of the LLM from the first contextualization request and re-prompt the LLM to reason through each of its suggested security issues, categorizing the violations as *insecure* only if very confident. Fig. 2(d) and Fig. 3(d) show examples of contextualization.

III. EXPERIMENTAL DETAILS

A. Dataset

Our dataset consists of 4 open-source RISC-V based SoCs: Hack@DAC 2021’s OpenPiton buggy SoC (H@DAC-21) [30], OpenTitan [31], Hummingbirdv2 E203 (E203) [32] and Veer-Wolf [33]. Their details are mentioned in Table I.

TABLE I
DATASET OF OPEN-SOURCE SoCs SCANNED FOR RELEVANT CWES.

SoC	Description	#Mods	#LoCs
H@DAC-21 [30]	OpenPiton SoC (CVA6 core) for Hack@DAC 2021 competition	63	15k
OpenTitan [31]	Silicon Root of Trust project (Ibex core)	359	171k
E203 [32]	Hummingbirdv2 E203 core and SoC	76	27k
VeerWolf [33]	FuseSoC-based platform for VeeR cores	34	11k

B. Hardware Common Weakness Enumerations (CWEs)

Security-related issues that arise because of hardware bugs are taxonomized as Common Weakness Enumerations (CWEs). MITRE [27] works with academia and industry to develop a list of CWEs that represent categories of vulnerabilities. A weakness is an element in a digital product’s software, firmware, hardware, or service that can be exploited for malicious purposes. We develop LASHED for 5 CWEs selected from the list of Most Important Hardware CWEs published by MITRE. We selected the ones that had coded examples on

TABLE II
CWES COVERED BY LASHED. SELECTED FROM MITRE’S LIST OF MOST IMPORTANT HARDWARE CWES.

CWE	Description
1191	On-Chip Debug and Test Interface With Improper Access Control
1231	Improper Prevention of Lock Bit Modification
1233	Security-Sensitive Hardware Controls with Missing Lock Protection
1244	Internal Asset Exposed to Unsafe Debug Access Level or State
1300	Improper Protection of Physical Side Channels

MITRE’s website and were present in the H@DAC-21 SoC to have some ground-truth for validation of initial prototyping. The 5 CWEs covered are described in Table II.

C. Prompt Variations

The performance of LLMs is dependent on the quality of prompts and examples of the correct solutions to the task at hand. To study the extent to which in-context learning [34] and insistence on reasoning helps in LASHED’s performance, we guide the LLM through 4 prompt variations. The 4 variations are formed using combinations of 2 improvements. The *first improvement* helps in the Assets Identification and Contextualization phases by providing a comprehensive example of a hardware security bug that captures the CWE. The *second improvement* helps in the Contextualization phase by asking the LLM to think again about its initial assessments of whether the reported violation poses a security risk.

1) *Variation v0 (baseline)*: is the zero-shot implementation for LASHED. It contains no in-context learning or request to re-evaluate outputs and forms the baseline to compare improvements in performance with variations v1, v2 and v3.

2) *Variation v1*: uses the *first improvement* only. The examples and descriptions of bugs for each CWE are taken from the MITRE’s website. Each example appears in the prompt after the LLM is given its role and information about the CWE it is going to look for. It consists of the bug in RTL form, the explanation of the security issues because of the bug and the relevant security assets. Here is an example for the prompt variation v1 appended to the baseline v0 for CWE 1231 (full example in Appendix Section B):

Here is an example of CWE-1231 with code from the register locks module:

```
always @(posedge clk_i) begin
  if (rst_ni && jtag_unlock && rst_9)) begin
    for (j=0; j < 6; j=j+1) begin
      reglk_mem[j] <= 'h0;
    <Explanation of the security issue>
```

In this example the lock signal is `reglk_mem` and the correct conditions for changing lock signals are `(rst_ni && jtag_unlock)`.

3) *Variation v2*: uses the *second improvement* only. The model is “given time to think” so that it can double check

its initial assessment. It is asked to use inner monologue to go over its reasoning process. It looks the same for all CWEs and appears in the user prompt for Contextualization. First, the LLM is asked to go over the violations and assess which of them are actually insecure and which are not. This output is then sent back to the LLM to simulate a chat. An example of the instruction to ‘reason’ for CWE-1231 is shown below:

Go over the previously provided response and reason about the provided explanation for each falsified property. Only categorize the falsified property as insecure if you are confident in your assessment. Here is the ‘falsified_properties’ object:
 <string of falsified assertions information>

4) Variation v3: uses both, first and second improvements.

D. Large Language Models (LLMs)

We use 2 OpenAI LLMs, *gpt-4o-mini-2024-07-18* and *gpt-4o-2024-08-06* [35], to conduct out experiments and evaluate if using a more powerful LLM makes a difference for our setup. *gpt-4o-mini* is OpenAI’s most advanced model in the small models category. *gpt-4o* is OpenAI’s most advanced GPT model and is slower and more advanced than *gpt-4o*. Both these models have the ability to give structured outputs consistent with the object structure provided.

IV. RESULTS

We evaluated LASHED on the 4 SoCs for 5 CWEs, 4 prompt variations and 2 LLMs. The results are summarized in Table III. In total, 545 instances are flagged across the 160 experiments out of which 51% are potential CWEs. 2026 assertions were formed and 12,554 assets were checked in the process. Since we do not know the real number of bugs (we can only confirm or deny a specific bug’s presence after flagging), it is not possible to calculate a proper Recall or Accuracy score. Therefore, we rely on Precision i.e. (# true positives / # predicted positive (flagged)) as the metric to evaluate performance. For our recommended combination of using *gpt-4o* with v3, 35 of the 40 flagged instances are plausible CWEs, providing a precision of 87.5%. On average, for a given SoC being searched for a particular CWE, choosing the appropriate prompt variation, there are 3.4 violations reported. We evaluated the violations manually, with our author-confirmed violations providing the True Positives count in Table III.

There is significant variation in LASHED’s success based on the CWE. It performs the best on CWEs 1191, 1244 and 1300 with precisions of 1, 0.8 and 0.91 and worse on CWEs 1231 and 1233 with precisions of 0.28 and 0.45. CWE 1231 was harder to identify because of poor asset identification. It was difficult for the LLM to identify the lock bit signal that should remain stable and instead kept identifying control and status registers. CWE 1233 was harder to accurately identify because of the range of possibilities of how a security sensitive signal may be protected in RTL. It may be protected by an if condition or by being ‘anded’ (&) with another signal or by being assigned a signal which may have some protection.

TABLE III
 RESULTS SUMMARY. PRECISION = TRUE POSITIVES (TPs) / FLAGGED, FALSE DISCOVERY RATE (FDR) = FALSE POSITIVES / FLAGGED, ASSETS = NUMBER OF ASSETS IDENTIFIED, ASSERTIONS = NUMBER OF ASSERTIONS FORMED FROM CUSTOM TEMPLATES FOR EACH CWE. THE BEST RESULTS ARE EMBOLDENED.

CWE	Variation	Flagged	TPs	Precision	FDR	Assets	Assertions
1191	v0	12	12	1.00	0.00	511	-
	v1	11	11	1.00	0.00	580	-
	v2	13	13	1.00	0.00	511	-
	v3	9	9	1.00	0.00	580	-
			45	45	1.00	0.00	2182
1231	v0	15	3	0.20	0.80	33	33
	v1	8	6	0.75	0.25	11	11
	v2	24	2	0.08	0.92	33	33
	v3	6	4	0.67	0.33	11	11
			53	15	0.28	0.72	88
1233	v0	143	55	0.38	0.62	669	587
	v1	114	48	0.42	0.58	406	355
	v2	68	35	0.51	0.49	669	587
	v3	89	50	0.56	0.44	406	355
			414	188	0.45	0.55	2150
1244	v0	5	3	0.60	0.40	24	12
	v1	3	3	1.00	0.00	22	15
	v2	1	1	1.00	0.00	24	12
	v3	1	1	1.00	0.00	22	15
			10	8	0.80	0.20	92
1300	v0	8	7	0.88	0.13	2053	-
	v1	9	8	0.89	0.11	1968	-
	v2	3	3	1.00	0.00	2053	-
	v3	3	3	1.00	0.00	1968	-
			23	21	0.91	0.09	8042
		545	277	0.508	0.49	12554	2026

A. Analysis

1) *Impact of Prompt Variations:* Prompt variations were successful in improving performance. As shown in Fig. 4-(Variation), variation v1 which introduces an example of CWE to guide the LLM, results in finding less false positives. The number of true positives decreases slightly from 80 to 76 while the precision improves from 0.44 to 0.52 in comparison to the baseline v0. Introducing v2, which asks the LLM to reason about its initial assessment during contextualization, shows improvement as well but to a lesser extent. The number of true positives decreases from 80 to 54 but the precision increases from 0.44 to 0.50 in comparison to the baseline v0. v2 significantly reduced false positives from 103 to 55. The best performing variation is v3 with highest precision of 0.62, a 42% improvement over the baseline. As v3 is a combination of v1 and v2, it seems that v1 can improve the ‘search’ of LASHED by identifying assets that are more relevant to the CWE and then v2 prunes out the remaining false positives. This can be seen in Fig. 5-(Variation) with an increase in removal of violations in contextualization for v3.

2) *Which SoCs were better analyzed?:* LASHED’s performance significantly varies with the SoC as illustrated in Fig. 4-

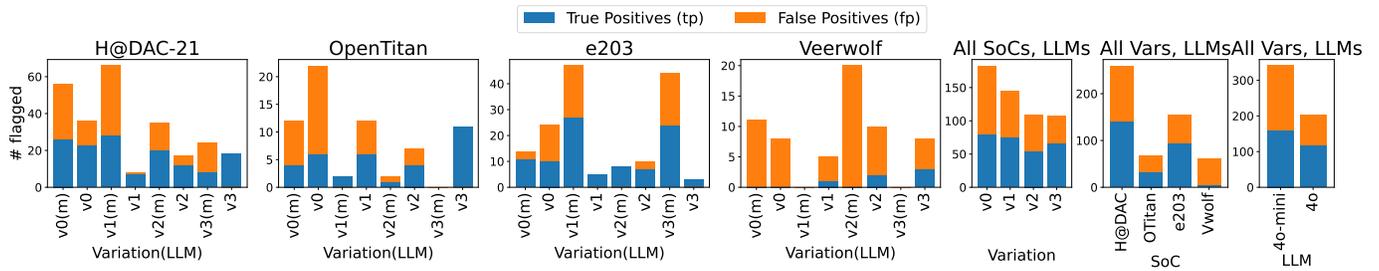


Fig. 4. Classification of instances flagged by LASHED. Stacked bar shows the number of instances identified by LASHED that pose security issues as true positives (tp) and those that do not as false positives (fp). The numbers are summed up for all CWEs.

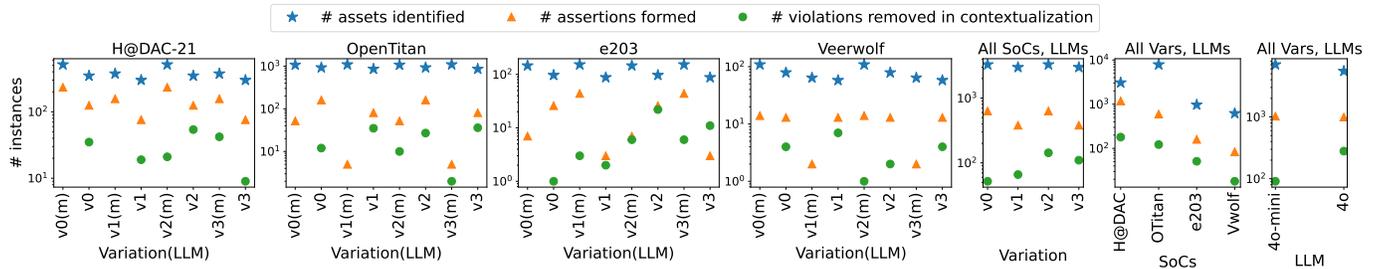


Fig. 5. Intermediary outputs during LASHED’s operation. # assets identified is number of security relevant signals identified during Assets Identification, # assertions formed is the number of assertions formed during Static Analysis. # violations removed in contextualization is the number of violations that the LLM reasoned as not posing a security threat. The numbers are summed up for all CWEs.

(SoC). It performed the best on e203 with a precision of 0.61 and the worst on Veerwolf with a precision of 0.10. The main culprit for poor performance on Veerwolf is the large number of false positives for CWE-1231. LASHED kept misidentifying a lot of the control and status registers as lock registers. Hack@DAC-21 performs the second best with a precision of 0.55, which is expected because the guiding examples we use in *v1* are inspired from the Hack@DAC-21 SoC. OpenTitan has a precision close to that of Hack@DAC-21, 0.50, which validates our approach – OpenTitan is significantly larger than Hack@DAC-21 in size and complexity.

Another area LASHED had performance issues in with Veerwolf and OpenTitan came with the forming of meaningful assertions. The ratio of assertion formation to number of assets identified for Hack@DAC-21 is significantly larger, as shown in Fig. 5-(SoC). This is perhaps due to the guiding examples being based on Hack@DAC-21.

3) *Which LLM performed better?*: LASHED performed better with *gpt-4o* (precision 0.58) compared to *gpt-4o-mini* (precision 0.47). *gpt-4o* flags less violations 204 vs. 341, less true positives 118 vs. 159, but also, less false positives 86 vs. 182, than *gpt-4o-mini*. This difference is highlighted when considering *v3* only: *gpt-4o* detects more true positives 35 vs. 32 and has a higher precision 0.88 vs. 0.47.

V. DISCUSSION

Our work shows that a combination of LLMs and static analysis can provide the ‘best of both worlds’, with each tool helping to overcome the limitations of the other. Detecting vulnerabilities with only LLMs leads to a large number of false positives and limited confidence on the outputs because there

is no verification. Conversely, detecting vulnerabilities with formal tools requires a lot of expertise and precise information which can be hard to obtain. We demonstrate that the limited confidence on the outputs of LLMs can be improved by verification through static analysis tools and the requirement of specific information can be provided by the LLM to some extent. The precision of 0.51 for LASHED (for all experiments combined) can be interpreted as a 51% confidence in the identification of a security issue.

Linting checks and assertions are very different in their nature. Lint rules are harder to map to security issues and assertions are harder to form correctly. Which of these work better in our flow is still an open question. The lint checks that appeared often in flagged instances were related to improper range indexes, concatenations and ‘if’ statements missing ‘else’ statements. While the confidence of reported violations is high, there are more false positives than would be ideal. The main culprit is liberal asset identification by the LLM. Providing the CWE information is not enough to constrain the number of assets the LLM identifies. Providing information regarding the operation of the specific modules and the security objectives could improve results. Another reason for false positives is the difficulty in forming correct assertions. Incorrect conditions identified by the LLM lead to failing assertions which flag violations erroneously. Another limitation lies in the manual evaluation of flagged violations. We assessed all violations through visual inspection which has a possibility of being incorrect. Each violation, however, is accompanied by a failing assertion or violation which gives credence to the classification.

VI. RELATED PRIOR WORKS

Few works have explored using both LLMs and Static Analysis components to identify RTL security bugs. SoCureLLM [12] is an LLM-driven approach for large-scale System-on-Chip security verification and policy generation. While they use LLMs for security policy generation and security policy violation, they do not use static tools for security violation in their solution. Flag-RTL [11] uses LLMs for bug detection in RTL but uses static analysis differently. A front end parser localizes the LLM's search to particular parts of the code, but no static analysis tool is used for verification. Self-HWDebug [13] automates LLM self-instructing for hardware security verification. This work uses known bugs and their CWEs to generate instructions for debugging and repair and there is no use of static analysis. While these works are related to LASHED, there are significant differences which do not allow for a direct comparison. None of the works use LLMs and Static Analysis together in equal levels of significance and none explore their tools on unknown bugs.

VII. CONCLUSION

This work combines LLMs and Static Analysis for hardware security bug detection. LASHED, by using LLMs equipped with in-context learning and requests for 'thinking again', has reasonable success in finding unknown bugs with an empirical precision of 0.88. On average, LASHED reported 3.4 violations per CWE for a given SoC, out of which 1.7 were evaluated to be plausible security issues. This is demonstrated over 5 of the most important Hardware CWEs, with the best performance on CWE 1191 (100% precision) and worst on CWE 1231 (28% precision). We catered the static analysis approach to the nature of the CWEs to show 2 techniques that do similarly well i.e., linting and assertions. Future work could investigate the application of reasoning models such as OpenAI o1 not available during our experimentation, as well as examine support for more CWEs.

VIII. ACKNOWLEDGMENTS

This research work is supported in part by a gift from Intel Corporation. This work does not in any way constitute an Intel endorsement of a product or supplier. We thank Verific Design Automation for generously providing academic access to linkable libraries, examples, and documentation for their RTL parsers.

REFERENCES

- [1] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "HardFails: Insights into Software-Exploitable Hardware Bugs," 2019, pp. 213–230. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky>
- [2] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *Proceedings of the 47th Design Automation Conference*, ser. DAC '10. New York, NY, USA: Association for Computing Machinery, Jun. 2010, pp. 12–17. [Online]. Available: <https://dl.acm.org/doi/10.1145/1837274.1837280>
- [3] S. Ray, N. Ghosh, R. Masti, A. Kanuparthi, and J. Fung, "INVITED: Formal Verification of Security Critical Hardware-Firmware Interactions in Commercial SoCs," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, Jun. 2019, pp. 1–4, iSSN: 0738-100X.
- [4] J. He, X. Guo, T. Meade, R. Dutta, Y. Zhao, and Y. Jin, "SoC interconnection protection through formal verification," *Integration*, vol. 64, pp. 143–151, Jan. 2019. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S016792601830289X>
- [5] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, May 2021, pp. 1286–1303, iSSN: 2375-1207.
- [6] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing Hardware Like Software," 2022, pp. 3237–3254. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity22/presentation/trippel>
- [7] A. Ardeshircham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*, Mar. 2017, pp. 1691–1696, iSSN: 1558-1101.
- [8] W. Hu, A. Ardeshircham, and R. Kastner, "Hardware Information Flow Tracking," *ACM Computing Surveys*, vol. 54, no. 4, pp. 83:1–83:39, May 2021. [Online]. Available: <https://dl.acm.org/doi/10.1145/3447867>
- [9] B. Ahmad, W.-K. Liu, L. Collini, H. Pearce, J. M. Fung, J. Valamehr, M. Bidmeshki, P. Sapiecha, S. Brown, K. Chakrabarty, R. Karri, and B. Tan, "Don't CWEAT It: Toward CWE Analysis Techniques in Early Stages of Hardware Design," in *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 1–9. [Online]. Available: <https://doi.org/10.1145/3508352.3549369>
- [10] 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, Feb. 2021.
- [11] B. Ahmad, B. Tan, R. Karri, and H. Pearce, "FLAG: Finding Line Anomalies (in code) with Generative AI," Jun. 2023, arXiv:2306.12643 [cs]. [Online]. Available: <http://arxiv.org/abs/2306.12643>
- [12] S. Tarek, D. Saha, S. K. Saha, M. Tehranipoor, and F. Farahmandi, "SoCureLLM: An LLM-driven Approach for Large-Scale System-on-Chip Security Verification and Policy Generation," 2024, publication info: Preprint. [Online]. Available: <https://eprint.iacr.org/2024/983>
- [13] M. Akyash and H. M. Kamali, "Self-HWDebug: Automation of LLM Self-Instructing for Hardware Security Verification," May 2024, arXiv:2405.12347. [Online]. Available: <http://arxiv.org/abs/2405.12347>
- [14] V. S. Lint, "Synopsys VC SpyGlass Lint," 2022. [Online]. Available: <https://www.synopsys.com/verification/static-and-formal-verification/vc-spyglass/vc-spyglass-lint.html>
- [15] jasperlint, "Jasper Superlint App," 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/jaspergold-superlint-app.html
- [16] "VC Formal," 2022. [Online]. Available: <https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html>
- [17] Cadence, "Jasper RTL Apps | Cadence," Jul. 2022. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html
- [18] D. Hansson, "Continuous Linting with Automatic Debug," in *2014 15th International Microprocessor Test and Verification Workshop*, Dec. 2014, pp. 70–72, iSSN: 2332-5674.
- [19] A. McNutt and G. Kindlmann, "Linting for Visualization: Towards a Practical Automated Visualization Guidance System," 2018.
- [20] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald, "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, no. 4, pp. 19:1–19:36, Oct. 2009. [Online]. Available: <https://dl.acm.org/doi/10.1145/1592434.1592436>
- [21] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating Large Language Models Trained on Code," Jul. 2021, arXiv:2107.03374 [cs]. [Online]. Available: <http://arxiv.org/abs/2107.03374>

- [22] S. Thakur, B. Ahmad, H. Pearce, B. Tan, B. Dolan-Gavitt, R. Karri, and S. Garg, “VeriGen: A Large Language Model for Verilog Code Generation,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 29, no. 3, pp. 46:1–46:31, Apr. 2024. [Online]. Available: <https://doi.org/10.1145/3643681>
- [23] B. Ahmad, S. Thakur, B. Tan, R. Karri, and H. Pearce, “On Hardware Security Bug Code Fixes by Prompting Large Language Models,” *IEEE Transactions on Information Forensics and Security*, vol. 19, pp. 4043–4057, 2024. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/10462177>
- [24] Z. Li, S. Dutta, and M. Naik, “LLM-Assisted Static Analysis for Detecting Security Vulnerabilities,” Nov. 2024, arXiv:2405.17238. [Online]. Available: <http://arxiv.org/abs/2405.17238>
- [25] P. J. Chapman, C. Rubio-González, and A. V. Thakur, “Interleaving Static Analysis and LLM Prompting,” in *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*. Copenhagen Denmark: ACM, Jun. 2024, pp. 9–17. [Online]. Available: <https://dl.acm.org/doi/10.1145/3652588.3663317>
- [26] H. Li, Y. Hao, Y. Zhai, and Z. Qian, “Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach,” *Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach (Artifact)*, vol. 8, no. OOPSLA1, pp. 111:474–111:499, Apr. 2024. [Online]. Available: <https://dl.acm.org/doi/10.1145/3649828>
- [27] T. M. C. (MITRE), “CWE - CWE Most Important Hardware Weaknesses,” 2022. [Online]. Available: https://cwe.mitre.org/scoring/lists/2021_CWE_MHW.html
- [28] Verific, “Verific Design Automation,” 2022. [Online]. Available: <https://www.verific.com/>
- [29] V. Formal, “VC Formal: Formal Verification Solution | Synopsys,” 2024. [Online]. Available: <https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html>
- [30] “HACK-EVENT/hackatdac21,” Apr. 2024, original-date: 2023-07-15T20:58:02Z. [Online]. Available: <https://github.com/HACK-EVENT/hackatdac21>
- [31] lowRISC contributors, “Open source silicon root of trust (RoT) | OpenTitan,” 2023. [Online]. Available: <https://opentitan.org/>
- [32] N. S. Technology, “Hummingbirdv2 E203 Core and SoC - GitHub,” May 2022, original-date: 2020-07-29T06:28:49Z. [Online]. Available: https://github.com/riscv-mcu/e203_hbirdv2
- [33] chipsalliance, “VeeRwolf,” Nov. 2024, original-date: 2019-08-07T15:24:36Z. [Online]. Available: <https://github.com/chipsalliance/VeeRwolf>
- [34] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba, “Large Language Models Are Human-Level Prompt Engineers,” Nov. 2022, arXiv:2211.01910 [cs]. [Online]. Available: <http://arxiv.org/abs/2211.01910>
- [35] OpenAI, “GPT-4o,” May 2024. [Online]. Available: <https://openai.com/index/hello-gpt-4o/>

APPENDIX

A. Scalability and Cost

On average, each experiment took 163 seconds to run. This time includes the complete flow of running the LLM and static analysis tools from the identification of relevant RTL to the outputs of location and explanation of bugs. In total, 160 experiments were run in 7.26 hours. There is a relation between the lines of code and the time taken. The dependency is linear in the log-log scale and produces follows the relation $time \propto loc^{0.53}$ approximately. This shows that the time taken grows proportional to the square root of the amount of code being analyzed, indicating a scalable approach. On average, each experiment cost \$0.055 while using gpt-4o-mini and \$0.379 while using gpt-4o. In total all experiments cost \$35 to run, catering to 40.2M input tokens.

B. Prompt variation $v1$

The examples and descriptions of bugs for each CWE are taken from the MITRE’s website. Each example appears in

the prompt after the LLM is given its role and information about the CWE it is going to look for. It consists of the bug in RTL form, the explanation of the security issues because of the bug and the relevant security assets. Here is an example for the prompt variation $v1$ appended to the baseline $v0$ for CWE 1231.

Here is an example of CWE-1231 with code from the register locks module:

```

"""
always @(posedge clk_i) begin
    if ( (rst_ni && jtag_unlock && rst_9)) begin
        for (j=0; j < 6; j=j+1) begin
            reglk_mem[j] <= 'h0;
        end
    end
end
"""

```

Register locks help prevent SoC peripherals’ registers from malicious use of resources. The registers that can potentially leak secret data are locked by register locks. In the vulnerable code, the `reglk_mem` is used for locking information. If one of its bits toggle to 1, the corresponding peripheral’s registers will be locked. A critical issue arises within the reset controller module. Specifically, the reset controller can inadvertently transmit a peripheral reset signal to the register lock within the user privilege domain. This unintentional action can result in the reset of the register locks, potentially exposing private data from all other peripherals, rendering them accessible and readable. In this example the lock signal is `reglk_mem` and the correct conditions for changing lock signals are `(rst_ni && jtag_unlock)`.
