VerilogLAVD: LLM-Aided Pattern Generation for Verilog CWE Detection

Anonymous ACL submission

Abstract

LLMs often fail in hardware vulnerability detection due to the intrinsic semantic concurrency of HDLs (Hardware Description Language), where vulnerabilities arise from the interaction of multiple concurrent execution statements rather than a single sequential execution path. Existing LLM-based methods struggle to capture the concurrency features. To address the problem, we propose VerilogLAVD, a **L**LM-**A**ided **V**ulnerability **D**etection framework by generating executable Traversal Detection Patterns (TDPs), i.e. the rules describing how to find the evidence of vulnerabilities in Verilog HDL. We first introduce a Unified Verilog Property Graph (VeriPG) that explicitly models parallel semantics by combining AST, CFG, and DDG. Furthermore, a semantic validation mechanism is designed to constrain and filter the LLM-generated TDPs. By executing these validated TDPs on VeriPG, our method produces stable and deterministic detection results. Experiments demonstrate that VerilogLAVD improves the F1 score by 133% compared to LLM-based methods. Furthermore, the framework successfully identifies real-world hardware vulnerabilities in open-source hardware design repositories.

1 Introduction

The growing complexity of modern hardware architectures has made security a critical concern throughout the design lifecycle. Ensuring compliance with security specifications of Hardware Description Languages (HDLs), such as Verilog, at the Register Transfer Level (RTL) is essential for building trustworthy hardware systems. However, conventional security verification techniques, such as formal verification and dynamic simulation(Orenes-Vera et al., 2021), often suffer from limited scalability and cannot exhaustively cover all possible design states(Akter et al., 2023). This shortfall can allow vulnerabilities to remain undetected until post-silicon stages, where fixing them

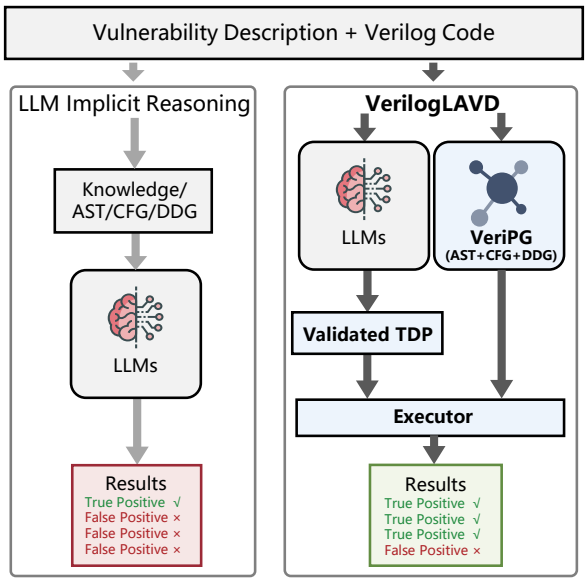

Figure 1: Comparison between LLM Implicit Reasoning and VerilogLAVD.

becomes far more costly(Karri et al., 2010). Existing early-stage static analysis tools, such as Synopsys Spyglass(Synopsys, 2015), focus mainly on functional correctness and depend heavily on manual, expert-defined rules, creating a high barrier to adoption and constraining scalability. Therefore, achieving automated and high-confidence vulnerability detection early in the design process remains a significant challenge.

Large language models (LLMs) have been widely applied in software vulnerability detection, but they often fail when used for hardware vulnerability detection. The core problem lies in the concurrent semantic nature of HDLs, unlike in the software domain, where vulnerabilities typically arise from the interaction of multiple concurrent execution statements rather than a single sequential execution path. Existing approaches usually feed code sequences or linearized graph representations directly into the LLM, relying entirely on the model's implicit reasoning for vulnerability detection(Fang et al., 2025; Saha et al., 2024; Lu et al.,

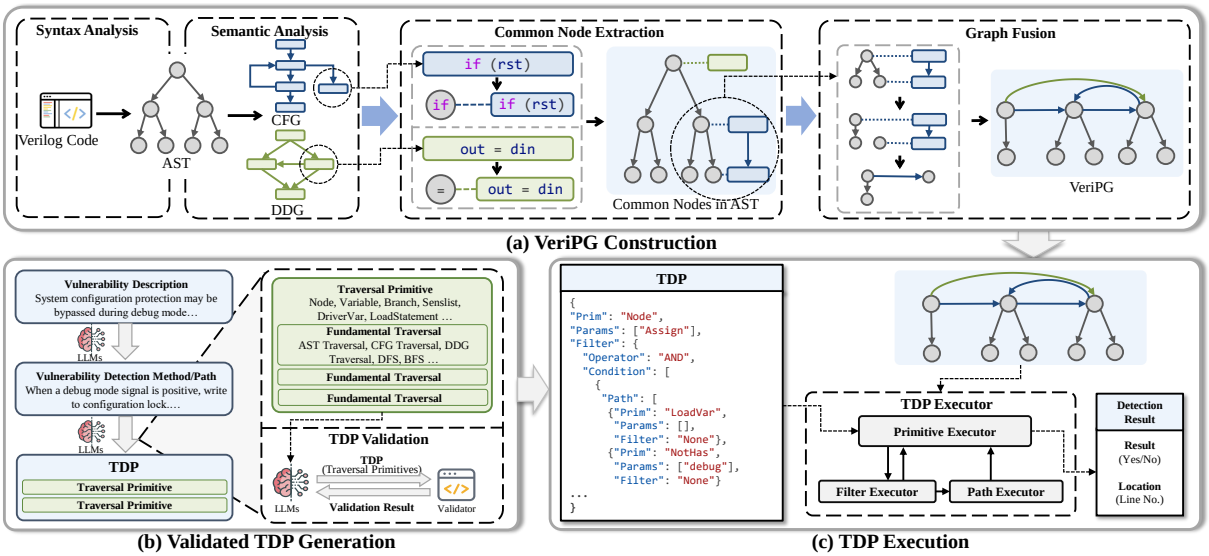

Figure 2: **Overview of the VerilogLAVD** framework. The workflow proceeds in three stages: (1) **VeriPG Construction**, which fuses AST, CFG, and DDG to model hardware concurrency; (2) **Validated TDP Generation**, where an LLM collaborates with a Validator to generate executable traversal patterns; and (3) **TDP Execution**, which runs the valid patterns on the graph to detect vulnerabilities.

2024), as Figure 1 (Left) shows. However, LLMs lack sufficient understanding of RTL structures, which is evident in the following aspects:

Challenge 1. Absence of a unified cross-structure representation: The essence of vulnerability evidence lies in the structural patterns that span the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Dependency Graph (DDG). However, existing methods(Ahmad et al., 2022) lack a unified representation to model these cross-structural interactions, relying on isolated structural views. Consequently, they fail to detect defects like CWE-1234 (Bypassing Privilege), which necessitates simultaneously linking signal existence (AST) with control-flow tracing (CFG).

Challenge 2. Difficulty in capturing the concurrency of RTL languages: RTL semantics are inherently concurrent, meaning that vulnerability patterns often do not correspond to a single sequential execution path. However, existing static analysis methods typically confine their analysis to a serial execution order, failing to account for parallel interactions. As a result, critical timing dependencies are overlooked, such as the distinction between blocking and non-blocking assignments required to identify CWE-1280 (Access Control).

To address these challenges, we propose VerilogLAVD, a LLM-Aided vulnerability detection framework for Verilog code given the descriptions of CWE, as Figure 1 (Right) shows. To provide a unified cross-structure representation, we design a **Verilog Property Graph (VeriPG)** by combining AST, CFG, and DDG with specific structural fusion to model hardware concurrency comprehensively. (Section 2.2). To capture the concurrent semantics of RTL, we propose a **Traversal Detection Pattern (TDP)** generation method with an iterative generation-validation process. TDP is a lightweight traversal pattern that detects vulnerabilities in VeriPG by tracing timing-related edges using parallel-aware traversal primitives. Within this process, the **TDP Validator** ensures compliance with Verilog semantic requirements by modeling graph traversal as state transitions in a Finite State Machine (FSM), thereby constraining and filtering the LLM-generated TDP. (Section 2.3). Finally, the **TDP Executor** parses and executes the LLM-generated validated TDPs on VeriPG to yield the final vulnerability detection results (Section 2.4).

The main contributions of this work are:

- We introduce the VeriPG, a cross-structural Verilog property graph representation that explicitly models parallel signal.
- We introduce TDPs, a parallel-aware traversal pattern for VeriPG, utilizing an iterative interaction between the LLM and our proposed TDP Validator to generate reliable detection rules.
- Experimental results demonstrate that our approach outperforms pure LLM baselines. Furthermore, we validate its practical efficacy by successfully identifying real vulnerabilities within open-source hardware design repositories.

2 Methodology

2.1 Overview

To address the challenges of intrinsic RTL concurrency and the lack of unified cross-structure representation, we propose VerilogLAVD, a framework that utilizes LLMs to generate executable TDPs for automated vulnerability detection. Figure 2 presents the workflow proceeds in three primary phases: **(1)** constructing the VeriPG from Verilog code; **(2)** leveraging LLMs and the TDP Validator to generate validated TDPs from natural language descriptions; and **(3)** executing these TDP on the graph to detect security vulnerabilities.

2.2 VeriPG Construction

We construct the Unified VeriPG by fusing AST, CFG, and DDG to explicitly capture RTL concurrency. The AST serves as the structural backbone, providing unique node identifiers that act as anchors for graph integration.

Graph Definition: Formally, a VeriPG is defined as a directed property graph $G = (V, E, P_V, P_E)$, where V is the set of vertices (e.g., syntax nodes, statements), $E \subseteq V \times V$ is the set of edges typed by P_E as AST, CFG, or DDG, P_V is a function assigning properties (e.g., node type, name, line number) to each vertex.

Control Flow Refinement: Unlike sequential software CFGs, we construct the CFG on a per-procedural-block basis. To model hardware concurrency, we modify the topology so that parallel statement nodes share a common parent node, rather than following a linear order. Edges are annotated with `condition` properties to explicitly distinguish branching logic from concurrent execution flows.

Data Dependency Optimization: We refine the DDG by strictly distinguishing blocking (=) and non-blocking (<=) assignments. While blocking assignments maintain sequential dependencies, non-blocking assignments update concurrently and do not affect reads in the current cycle. Therefore, we explicitly prune erroneous dependency edges originating from non-blocking assignments to subsequent statements within the same block, ensuring the graph reflects true signal propagation timing.

Graph Fusion: We integrate the structures by aligning common nodes (e.g., assignments, control constructs) across the AST, CFG, and DDG using their unique AST identifiers. This superimposes explicit control-flow and data-dependency edges onto the syntactic skeleton, yielding a unified, cross-structural graph amenable to TDP traversal.

```
{
  "Prim": [Primitive Traversal Name],
  "Params": [Primitive Traversal Parameter List],
  "Filter": {
    "Operator": [Filter Condition Combine Operator (AND/OR)],
    "Condition": [
      {"Path": [Sequential-Executed TDP List]},
      [More TDPs or Paths],
      ...
    ]}
}
```

Figure 3: TDP Format.

CWE Vulnerability Description (CWE-1234)

System configuration protection may be bypassed during debug mode. . .If debug features supported by hardware or internal modes/system states are supported in the hardware design, modification of the lock protection may be allowed allowing access and modification of configuration information.

Vulnerability Detection Method (CWE-1234)

1. The code contains debug or test interfaces (e.g., such as "debug").
2. Debug access privileges pass the permission check.
3. The privilege protection logic involves modifications to the lock.

Vulnerability Detection Path (CWE-1234)

1. Traverse from Always to IfStatement via CFG.
2. From IfStatement to Identifier via AST.
3. Check if Identifier name contains "debug".
4. From IfStatement to assignment nodes (= or <=) via CFG.
5. From assignment nodes (= or <=) to Identifier via AST.
6. Check if Identifier name contains "lock".

Traversal Detection Pattern (CWE-1234)

```
{
  "Prim": "Node",
  "Params": ["If"],
  "Filter": {
    "Operator": "AND",
    "Condition": [
      {"Path": [
          {"Prim": "ConditionVar", "Params": [], "Filter": "None"},
          {"Prim": "Has", "Params": ["debug"], "Filter": "None"}]},
      {"Path": [
          {"Prim": "Assignment", "Params": [], "Filter": "None"},
          {"Prim": "DriverVar", "Params": [], "Filter": "None"},
          {"Prim": "Has", "Params": ["lock"], "Filter": "None"}]}]}
}
```

Figure 4: A Example of TDP Progressive Generation Process.

2.3 TDP Generation

TDP Format: As illustrated in Figure 3, the TDP is designed as a composite structure to capture complex vulnerability patterns. It integrates **Primitive Traversal Operations** that abstract raw graph queries into high-level semantic actions (e.g., signal tracing), with **Traversal Parameters** that support diverse inputs—crucially including nested TDPs, to enable recursive data flow where inner results feed outer operations. Furthermore, a **Filter** component is employed to refine traversal outputs via boolean composition (AND/OR), retaining only those primitive operation results that satisfy specific constraint traversal paths.

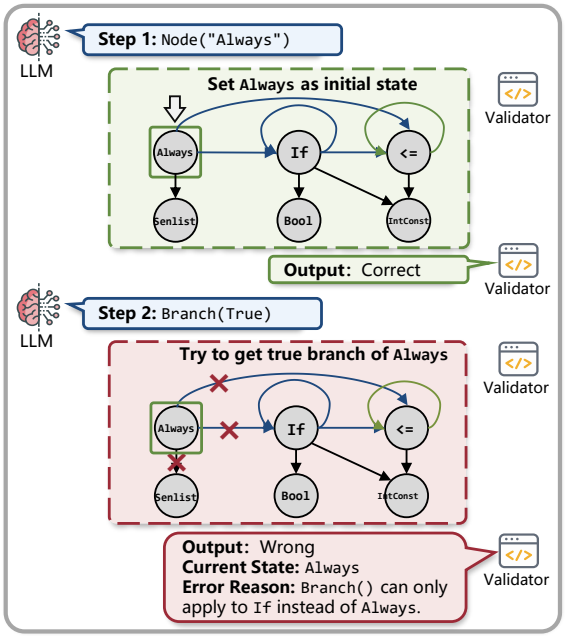

Figure 5: TDP Validator.

Progressive Generation: As exemplified by the CWE-1234 case in Figure 4, we implement a multi-stage workflow based on in-context learning to bridge the semantic gap between text and graph queries. The process evolves from logical analysis, which distills the CWE description into key signal relationships (Detection Method), to topological mapping, where abstract logic is translated into specific VeriPG structural flows (Detection Path). Finally, the system synthesizes these structural paths into concrete primitive operations, employing a Validator to enforce structural correctness via an iterative feedback loop. Throughout this procedure, we incrementally feed the VeriPG structure (nodes, edges, and connectivity), TDP format, available traversal primitives, and illustrative TDP examples into the LLM, thereby minimizing the context requirements for each individual generation step.

TDP Validator: To transform stochastic LLM outputs into reliable engineering artifacts, the Validator ensures generated TDPs comply with strict topological constraints of VeriPG. Specifically, it enforces two critical constraints: validating the structural existence of referenced nodes in the topology and confirming parameter legality against traversal primitives. This is achieved through a FSM that formally encodes the VeriPG schema, where an operation is deemed legal only if a valid transition exists between the current FSM state and the corresponding target node type state.

Figure. 5 illustrates this process. While the initial `Node("Always")` correctly initializes the FSM, the subsequent `Branch(True)` triggers a validation failure because branching is topologically restricted to `If` states. This mismatch against the current `Always` state prompts the Validator to return specific error feedback, guiding the LLM to refine the pattern that does not comply with constraints.

Category	CWEs	No.
Improper Access Control	1231, 1243, 1244, 1280, 1262	21
Improper Resource Operate	226, 1258, 1271	13
Improper Lock	1232, 1234	11
Side Channel	1255, 1300	11
Finite State Machine	1245	7
Non-Vulnerability	None	12

Table 1: Consist of Synthetic CWE Benchmark.

2.4 TDP Executor

The Executor translates validated TDPs into deterministic graph queries, transitioning from stochastic inference to mechanical execution. It coordinates three modular components to enforce semantic correctness: the **Primitive Executor** handles graph transitions and property lookups; the **Filter Executor** prunes paths violating constraints; and the **Path Executor** manages execution state. Crucially, this module provides white-box auditability by generating verifiable execution traces mapped to specific code lines. Furthermore, it maintains high efficiency through "Irrelevant Code Insensitivity," where computational overhead correlates with TDP complexity rather than the total design size, ensuring scalability for large designs.

3 Experimental Setup

3.1 Dataset

To evaluate the effectiveness and scalability of VerilogLAVD, we constructed a hybrid dataset consisting of a synthetic benchmark and a large-scale real-world hardware design repository.

3.1.1 Synthetic CWE Benchmark

Given the scarcity of publicly available hardware vulnerability datasets, we created a custom benchmark covering 13 CWE categories essential to RTL security, such as FSM logic errors and Access Control violations, as Table 1. We anchored this dataset in the official vulnerability descriptions and code examples provided by the MITRE CWE website. Using these official examples as a foundation, hardware security experts manually authored and expanded the test cases, incorporating complex control flows and data dependencies to mimic realistic

Vulnerability Category	DeepSeek-V3			DeepSeek-V3 +Knowledge			DeepSeek-V3 +VerilogLAVD			GPT-4o			GPT-4o +Knowledge			GPT-4o +VerilogLAVD		
	P(%)	R(%)	F1	P(%)	R(%)	F1	P(%)	R(%)	F1	P(%)	R(%)	F1	P(%)	R(%)	F1	P(%)	R(%)	F1
Improper Access Control	15.07	64.71	**0.24**	25.71	52.94	**0.35**	48.15	76.47	**0.59**	11.71	76.47	**0.20**	14.85	88.24	**0.25**	60.87	82.35	**0.70**
Improper Resource Operate	13.79	30.77	**0.19**	26.09	46.15	**0.33**	24.32	69.23	**0.36**	12.07	53.85	**0.20**	12.90	61.54	**0.21**	32.26	76.92	**0.45**
Improper Lock	31.25	45.45	**0.37**	35.29	54.55	**0.43**	43.75	63.64	**0.52**	24.24	72.73	**0.36**	21.88	63.64	**0.33**	46.15	54.55	**0.50**
Side Channel	18.75	27.27	**0.22**	26.67	36.36	**0.31**	83.88	45.45	**0.59**	8.16	36.36	**0.13**	12.50	72.73	**0.21**	53.85	63.64	**0.58**
Finite State Machine	11.11	14.29	**0.13**	17.65	42.86	**0.25**	45.45	71.43	**0.56**	18.75	42.86	**0.26**	27.27	42.86	**0.33**	54.55	85.71	**0.67**
Total	16.78	40.68	**0.24**	26.17	47.46	**0.34**	40.21	66.10	**0.50**	13.11	59.32	**0.21**	15.19	69.49	**0.25**	47.25	72.88	**0.57**

Table 2: Vulnerability detection performance evaluation of LLMs and VerilogLAVD. F1 scores are shown in bold, and the best-performing metrics are highlighted with a green background.

coding patterns. This process resulted in 81 verified Verilog designs, providing a balanced set of vulnerable and samples for controlled evaluation.

3.1.2 Real-world Repository Dataset

To assess performance in realistic scenarios, we compiled a diverse collection of open-source hardware projects, ranging from the Hack@DAC challenge sets to major RISC-V implementations like OpenTitan (Ibex), CV32E40P, Mor1kx, and ORPSoC. This corpus spans various coding styles and totals approximately 1.08 million Source Lines of Code (SLOC)(flosse, 2012). Determining ground truth in such large-scale repositories is challenging; therefore, we relied on a rigorous manual audit. Hardware security experts inspected the tool's alerts against the source code to accurately label vulnerabilities, ensuring a reliable distinction between genuine defects, actionable code quality issues, and false positives.

3.2 Baselines

We compare VerilogLAVD against two categories of methods: (1) **LLM-based Approaches**: We select DeepSeek-V3(DeepSeek-AI, 2024) and GPT-4o(OpenAI, 2024) as backbones, evaluating them in two settings: *Direct Prompting* (Zero-shot) and a stronger *Knowledge-Augmented* baseline (providing code plus CWE definitions) to test reasoning capabilities beyond mere knowledge retrieval. We set the sampling temperature to 0.5. (2) **Static Analysis Methods**: We include CWEAT(Ahmad et al., 2022), a SOTA template-matching tool. This allows us to contrast our semantic reasoning approach against rigid syntactic matching, particularly in scenarios involving complex data flows.

3.3 Evaluation Metrics

We adopt a multi-dimensional assessment: (1) **Detection Quality**: We report Precision, Recall, and F1-Score on synthetic benchmarks, using F1 as the primary metric. For methods involving LLMs, we conducted 5 independent runs and reported the average results. (2) **Structural Reliability**: To measure LLM hallucinations, we introduce *Invalid Path Rate (IPTR)* for semantic mismatches and *Invalid Parameter Rate (IPMR)* for syntactic errors in generated TDPs. (3) **Practicality**: In real-world settings, we manually classify alerts (TP/FP/Indeterminate) of TDP execution results and analyze execution latency relative to both graph scale and traversal complexity.

4 Evaluation

To comprehensively evaluate the feasibility, reliability, and scalability of our proposed framework, we designed our experiments to answer the following four research questions:

RQ1: How effective is VerilogLAVD in detecting verilog vulnerabilities? We evaluate its performance against state-of-the-art LLM-based approaches and template-based static analysis to determine if graph-based reasoning outperforms LLM implicit reasoning and rigid pattern matching.

RQ2: How crucial is the Validator for generating reliable patterns? We investigate the contribution of this component to the overall system, specifically analyzing its ability to mitigate LLM hallucinations and ensure the semantic correctness of the generated traversal logic.

RQ3: What determines the execution efficiency of the VeriPG traversal engine? We examine the performance characteristics across varying design scales, analyzing whether execution overhead is driven by the graph size (nodes and edges) or the complexity of the traversal logic.

RQ4: Is VerilogLAVD practical for large-scale, real-world repositories? We assess the framework's capability to identify valid defects and actionable insights within complex, heterogeneous industrial codebases where ground truth is not pre-

CWE ID	CWEAT(%)	GPT-4o(%)	VerilogLAVD(%)
226	-	22.84	83.02
1231	-	71.73	72.82
1232	-	56.75	55.21
1234	75.00	88.08	65.20
1243	-	70.05	88.87
1244	-	66.96	65.89
1245	85.71	38.38	87.34
1255	-	17.56	83.62
1258	-	95.60	97.35
1262	66.67	34.23	62.03
1271	83.33	55.92	57.03
1280	60.00	57.60	84.91
1300	-	46.94	53.09

Table 3: Compare with CWEAT (Recall).

established.

4.1 Effectiveness of VerilogLAVD (RQ1)

Table 2 demonstrates that VerilogLAVD significantly outperforms both 'LLM-only' and 'LLM+Knowledge' baselines across diverse CWE categories. By anchoring detection in executable graph traversals rather than relying on unconstrained implicit reasoning, our method achieves an F1 score of 0.57 with the GPT-4o backbone, representing a 133.33% improvement over the knowledge-enhanced baseline. This superiority is consistent across model architectures; for instance, DeepSeek-V3 shows a marked performance increase, with its F1 score rising from 0.35 (knowledge-augmented) to 0.50. VerilogLAVD excels particularly in complex categories like FSM logic and Improper Access Control, where the ability to explicitly trace data dependencies via VeriPG captures subtle vulnerabilities that purely textual approaches consistently fail to identify.

The absolute performance (F1=0.57) reflects the inherent intricacy of aligning natural language with rigid hardware semantics. The remaining gap is primarily attributed to lexical diversity, where naming variations (e.g., 'reset' vs. 'rst') introduce alignment noise, and the complexity of arithmetic semantics, as validating precise value-domain constraints (e.g., boundary checks) extends beyond the scope of topological graph traversal.

As detailed in Table 3, to further validate our framework against traditional static analysis, we benchmarked VerilogLAVD against CWEAT. VerilogLAVD exhibits superior generalizability: while CWEAT relies on rigid predefined rules and fails to support 8 out of the 13 evaluated CWE categories, our approach effectively covers the entire spectrum. Moreover, in handling complex concurrent semantics such as CWE-1280 (Access Control), VerilogLAVD surpasses CWEAT by a significant margin (84.91% vs. 60.00% Recall), demonstrating the distinct advantage of combining LLM reasoning with graph traversals over fixed static patterns.

Method	IPTR	IPMR	Total
LLM+Knowledge	27.04%	13.83%	40.87%
VerilogLAVD (w/o Validator)	23.96%	12.88%	36.84%
VerilogLAVD (with Validator)	**3.49%**	**6.61%**	**10.10%**

Table 4: Misuse in TDP generation.

4.2 Impact of Validatator (RQ2)

We investigate the Validator's role in mitigating LLM structural hallucinations through ablation studies on the DeepSeek-V3 backbone. We compared three configurations: (1) Baseline (Knowledge-Augmented), (2) VerilogLAVD w/o Validator, and (3) VerilogLAVD w/ Validator. We measured Traversal Primitive Misuse, categorized into *IPTR* (incompatible primitives) and *IPMR*.

Results in Table 4 confirm that the iterative validation loop is critical. It reduces the overall misuse rate to 10.10%, compared to 40.87% for the baseline and 36.84% without validation. Most notably, the *IPTR* dropped to 3.49%, representing an 87.10% reduction. This demonstrates that the feedback effectively aligns stochastic LLM outputs with the topological constraints of VeriPG.

4.3 Efficiency Across Verilog Design Scales (RQ3)

We evaluate scalability by analyzing the correlation between code volume, TDP complexity, and execution latency. As shown in Figure 6, there is no linear dependency between Lines of Code (LOC) and execution time. Unlike conventional static analysis, VerilogLAVD demonstrates "Irrelevant Code Insensitivity", runtime correlates strongly with the number of executed primitives (Figure 7) rather than the total design size. This confirms that the computational cost is governed by the complexity of the detection logic (depth/breadth of traversal) rather than the size of the Verilog file. A case study on CWE-1280 further illustrates this efficiency. By executing a targeted path search across AST, CFG, and DDG rather than simulating the entire block, the TDP identifies complex race conditions with

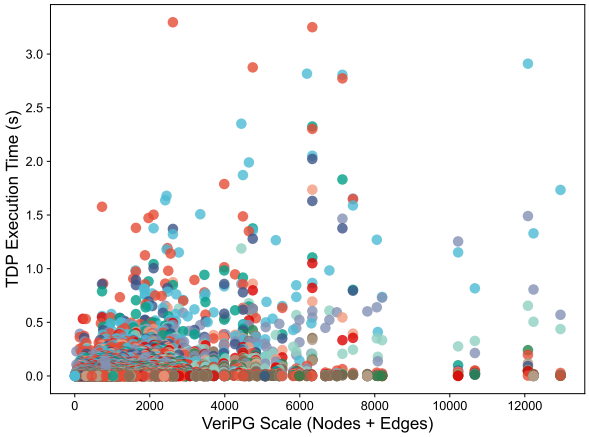

Figure 6: Relation between TDP Execution Time and VeriPG Scale.

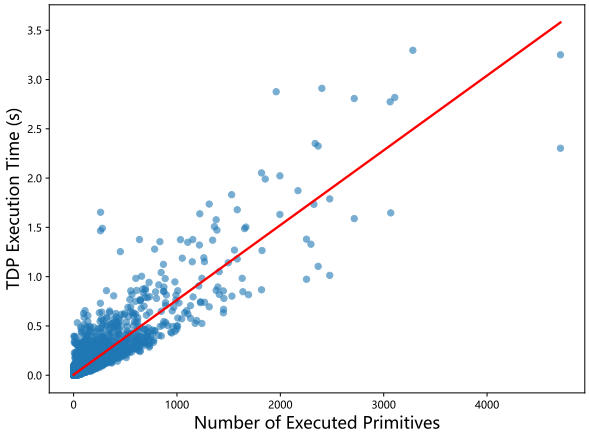

Figure 7: Relation between TDP Execution Time and Number of Executed Primitives.

millisecond-level latency. This structural bounding allows the framework to maintain consistent performance even in large-scale industrial designs.

4.4 Real-world Applicability (RQ4)

To answer RQ4, we deployed VerilogLAVD on a heterogeneous corpus comprising 11 open-source repositories, ranging from CTF benchmarks (Hack@DAC) to industrial-grade RISC-V cores (e.g., OpenTitan/Ibex, ORPSoC), totaling approximately 1.08 million lines of code (Table 5). Given the absence of ground truth for the industrial repositories, we conducted a rigorous manual audit of the 107 reported alerts to evaluate their validity and engineering utility. The verification results are summarized in Figure 8.

This result yields three key insights: (1) **High FSM Precision:** The framework achieves dominant performance on Improper FSM Transitions (CWE-1245) with a strict precision of 66.1%

Repository	Files	LoC(k)	1234	1245	1262	1271	1280
hackatdac21	514	329.1	0	7	1	9	3
hackatdac18	374	119.2	3	34	1	6	4
e203_hbirdv2	138	89.1	1	1	4	0	2
orpsoc-cores	240	434.3	5	10	0	3	0
Cores-VeeR-EH1	9	5.1	0	1	0	0	0
ibex	42	26.3	0	0	0	0	0
cv32e40p	179	46.6	0	2	0	1	0
scr1	1	0.3	0	0	0	0	0
serv	75	10.0	0	1	0	0	0
riscv	54	17.9	0	3	0	0	4
mor1kx	42	38.1	0	0	0	1	0

Table 5: Detection result in real-world open-source repository.

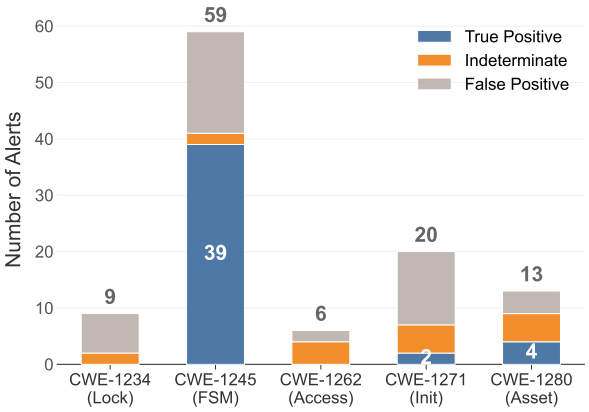

Figure 8: Classification result of alert examples in real-world.

(39/59 TPs). This confirms that explicit CFG modeling enables accurate reasoning about missing transitions where traditional tools fail. (2) **Context-Dependent Utility:** While Access Control (CWE-1280) shows lower strict precision, it uncovers valuable "Indeterminate" warnings (e.g., missing local resets in the e203 chip). Including these actionable alerts raises the useful detection rate for CWE-1280 to 69.2%. (3) **Static Analysis Limits:** Uninitialized Values (CWE-1271) exhibited higher false positives (2/20 TPs) due to the lack of runtime simulation data. However, the tool remains valuable by narrowing the audit scope for human experts.

4.5 Case Study

We present a representative RTL vulnerability case in which data-flow analysis is critical for accurate detection. Specifically, detecting CWE-1280 requires analyzing data dependencies: this vulnerability arises when an access control condition is used before being properly initialized, potentially leading to nondeterministic execution behavior, as Figure 9 shows. To efficiently detect such issues, our approach uses customized traversal primitives for the CFG and DDG. Figure 10 shows the corre-

```
...
1 always @ (posedge clk or negedge rst_n)
2 begin
3   if (!rst_n)
4     data_out = 0;
5   else
6     data_out = (grant_access) ? data_in : data_out;
7     grant_access = (usr_id == 3'h4) ? 1'b1 : 1'b0;
8 end
...
```

Figure 9: Vulnerability Code (CWE-1280).

```
...
"Func": "Variable",
...
    {"Func": "LoadStatement", "Params": []},
    {"Func": "DriverStatement", "Params": ['=']}
    {"Func": "Exist", "Params": []}
...
```

Figure 10: TDP Example (CWE-1280).

sponding TDP consists of 4 key steps:

1. The process initiates with the **`Variable`** primitive to scan the module scope. A filter is applied to identify critical control signals (e.g., `grant_access`) that govern data flow logic.
2. From the identified signal, the traversal follows DDG edges using **`LoadStatement`** to locate where the signal is consumed. In Figure 9, this maps to the conditional usage in the line 6.
3. The traversal find the definition or assignment of the signal using **`DriverStatement`**. Crucially, the TDP specifies the parameter `['=']`, restricting the search to blocking assignments. This step identifies the problematic line 7, due to its blocking nature, creates the race condition.
4. Finally, the **`Exist`** primitive evaluates the boolean result of this specific path. If a signal is found to be used in a conditional logic and driven by a blocking assignment in a manner that violates sequential safety, the pattern returns true, flagging the vulnerability.

This case demonstrates that our structure-bounded TDPs transform LLM reasoning from opaque implicit reasoning into verifiable graph-based analysis, enabling precise detection of both data-flow and control-flow–dependent vulnerabilities.

5 Related Work

Traditional Hardware Verification. Existing methods face distinct limitations: simulation-based validation (Dessouky et al., 2019; Rajendran et al., 2023) and fuzzing (Hossain et al., 2023; Al-Shaikh et al., 2023) suffer from labor intensiveness and limited coverage; formal verification (Aftabjahani et al., 2021; Orenes-Vera et al., 2021) battles state space explosion and high expertise barriers. While static analysis (Ahmad et al., 2022) and IFT (Solt et al., 2022; Zhao et al., 2024) offer precision, they incur high computational overhead or require custom detector development. Similarly, ML-based methods (Fan et al., 2024; Yasaei et al., 2022) are constrained by the scarcity of large-scale datasets.

LLM-aided Verification. Recent works utilize LLMs for implicit reasoning (Akyash and Kamali, 2024; Pearce et al., 2025; Zhang et al., 2024), SVA generation (Orenes-Vera et al., 2023), and IFT optimization (Mashnoor et al., 2025). However, these approaches often struggle to comprehend the inherent structural concurrency of RTL or produce unstable formal properties. Crucially, prior work has not investigated integrating LLMs with RTL-structure-based analysis. Our approach bridges this gap by leveraging LLMs to generate traversal rules, significantly reducing manual effort and enhancing scalability for early-stage design.

6 Conclusion

In this paper, we proposed VerilogLAVD, a neuro-symbolic framework addressing the lack of effective security analysis in early-stage hardware design. By constructing the unified VeriPGand leveraging the TDP within an iterative generation-verification loop, our approach captures critical RTL concurrency semantics that evade pure LLMs. Experimental results confirm its superiority, achieving F1 score improvements of 0.31 and 0.25 over pure LLM and RAG baselines, respectively, validating the necessity of graph reasoning.

Limitations

Our work has two primary limitations. First, regarding scalability, constructing the VeriPG for massive, full-chip designs is computationally intensive; our current evaluation focuses on module-level and subsystem-level analysis. Second, our approach is bound by LLM capabilities, specifically the context window size when reasoning about extremely long vulnerability traces, and the inherent non-determinism of LLM outputs which requires rigorous verification loops.

References

Sohrab Aftabjahani, Ryan Kastner, Mark Tehranipoor, Farimah Farahmandi, Jason Oberg, Anders Nordstrom, Nicole Fern, and Alric Althoff. 2021. Special

session: Cad for hardware security-automation is key to adoption of solutions. In *2021 IEEE 39th VLSI Test Symposium (VTS)*, pages 1–10. IEEE.

Baleegh Ahmad, Wei-Kai Liu, Luca Collini, Hammond Pearce, Jason M Fung, Jonathan Valamehr, Mohammad Bidmeshki, Piotr Sapiecha, Steve Brown, Krishnendu Chakrabarty, and 1 others. 2022. 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*, pages 1–9.

Sonia Akter, Kasem Khalil, and Magdy Bayoumi. 2023. A survey on hardware security: Current trends and challenges. *IEEE Access*, 11:77543–77565.

Mohammad Akyash and Hadi Mardani Kamali. 2024. Self-hwdebug: Automation of llm self-instructing for hardware security verification. In *2024 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pages 391–396. IEEE.

Hasan Al-Shaikh, Arash Vafaei, Mridha Md Mashahedur Rahman, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. Sharpen: Soc security verification by hardware penetration test. In *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, pages 579–584.

DeepSeek-AI. 2024. Deepseek-v3 technical report. *Preprint*, arXiv:2412.19437.

Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2019. {HardFails}: insights into {software-exploitable} hardware bugs. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 213–230.

Ruchao Fan, Yongming Tang, Hao Sun, Jiyuan Liu, and He Li. 2024. An efficient ml-based hardware trojan localization framework for rtl security analysis. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, pages 1–7.

Zhigang Fang, Renzhi Chen, Zhijie Yang, Yang Guo, Huadong Dai, and Lei Wang. 2025. Lintllm: An open-source verilog linting framework based on large language models. *arXiv preprint arXiv:2502.10815*.

flosse. 2012. flosse/sloc. https://github.com/flosse/sloc. Accessed: 2025-11-24.

Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. 2023. Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Ramesh Karri, Jeyavijayan Rajendran, Kurt Rosenfeld, and Mohammad Tehranipoor. 2010. Trustworthy hardware: Identifying and classifying hardware trojans. *Computer*, 43(10):39–46.

Guilong Lu, Xiaolin Ju, Xiang Chen, Wenlong Pei, and Zhilong Cai. 2024. Grace: Empowering llm-based software vulnerability detection with graph structure and in-context learning. *Journal of Systems and Software*, 212:112031.

Nowfel Mashnoor, Mohammad Akyash, Hadi Kamali, and Kimia Azar. 2025. Llm-ift: Llm-powered information flow tracking for secure hardware. In *2025 IEEE 43rd VLSI Test Symposium (VTS)*, pages 1–5. IEEE.

OpenAI. 2024. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/. Accessed: 2025-07-11.

Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. 2021. Autosva: Democratizing formal verification of rtl module interactions. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 535–540. IEEE.

Marcelo Orenes-Vera, Margaret Martonosi, and David Wentzlaff. 2023. Using llms to facilitate formal verification of rtl. *arXiv preprint arXiv:2309.09437*.

Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. 2025. Asleep at the keyboard? assessing the security of github copilot's code contributions. *Communications of the ACM*, 68(2):96–105.

Sree Ranjani Rajendran, Shams Tarek, Benjamin M Hicks, Hadi M Kamali, Farimah Farahmandi, and Mark Tehranipoor. 2023. Hunter: Hardware underneath trigger for exploiting soc-level vulnerabilities. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE.

Dipayan Saha, Katayoon Yahyaei, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. 2024. Empowering hardware security with llm: The development of a vulnerable hardware database. In *2024 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 233–243. IEEE.

Flavien Solt, Ben Gras, and Kaveh Razavi. 2022. {CellIFT}: Leveraging cells for scalable and precise dynamic information flow tracking in {RTL}. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2549–2566.

Synopsys. 2015. Spyglass: Early design analysis tools for socs. https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html. Accessed: 2025-07-11.

Rozhin Yasaei, Luke Chen, Shih-Yuan Yu, and Mohammad Abdullah Al Faruque. 2022. Hardware trojan

detection using graph neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(1):25–38.

Jian Zhang, Chong Wang, Anran Li, Weisong Sun, Cen Zhang, Wei Ma, and Yang Liu. 2024. An empirical study of automated vulnerability localization with large language models. *arXiv preprint arXiv:2404.00287*.

Yiqiang Zhao, Gonsen Qu, Qizhi Zhang, Yao Li, Zhengyang Li, and Jiaji He. 2024. Static gate-level information flow for hardware information security with bounded model checking. In *2024 IEEE 42nd VLSI Test Symposium (VTS)*, pages 1–7. IEEE.

A Traversal Primitive

The VerilogLAVD framework introduces three distinct categories of traversal primitives (Generic, Boolean, and VeriPG-specific) to bridge the gap between the abstract reasoning of LLMs and the rigorous precision required for hardware analysis.

A.1 Generic graph traversal

Generic Graph Traversals provide the fundamental capability to navigate the underlying graph structure, enabling the detector to move seamlessly across the AST, CFG, and DDG to correlate syntax with execution flow.

Name	Description
Node	Retrieve node IDs of a specified type. Input: node type; Output: a list of node IDs of that type.
ASTOffspring	Retrieve descendant node IDs of a specified type in the AST. Input: node ID and descendant node type; Output: a list of descendant node IDs.
CFGOffspring	Retrieve descendant node IDs of a specified type in the CFG. Input: node ID and descendant node type; Output: a list of descendant node IDs.
DDGOffspring	Retrieve descendant node IDs of a specified type in the DDG. Input: node ID and descendant node type; Output: a list of descendant node IDs.

Table 6 Description of Generic Graph Traversals

A.2 Boolean operation

Boolean Operations function as the core logical mechanism for filtering, allowing the system to compose complex conditions such as existence checks or value comparisons to precisely define vulnerability matches and reduce false positives.

Name	Description
Exist	Check whether a list is non-empty. Input: a list; Output: a boolean value.
Has	Check whether a list contains a specified traversal. Input: variable name; Output: a boolean value. In a Path object, a preceding node must first retrieve the variable using Var.
Count	Return count of input list, Input: list; Output: list length.
Equal	Check whether two input are equal. Input: two object. Output: Compare results
Merge	Combine two list. Input: two lists. Output: Merge results

Table 7 Description of Boolean Operations

A.3 VeriPG-specific traversal

VeriPG-specific Traversals address the limitation of generic algorithms in understanding hardware concurrency by encapsulating Verilog-specific semantics, including the critical distinction between blocking and non-blocking assignments.

Name	Description
Always	Get all Always nodes; output is a list of their IDs.
If	Get all If nodes; output is a list of their IDs.
For	Get all For nodes; output is a list of their IDs.
Blocking	Get all BlockingSubstitution assignment nodes; output is a list of their IDs.
Nonblocking	Get all NonblockingSubstitution assignment nodes; output is a list of their IDs.
Assign	Get all Assign assignment nodes; output is a list of their IDs.
Switch	Get all Switch nodes; output is a list of their IDs.
Variable	Get all declared signal names; output is a list of signal names.
SensVar	Get sensitive variables of a specified Always node; output is a list.
SensEdge	Get sensitivity edges of a specified Always node; output is a dictionary of variable-edge pairs.
ConditionVar	Get condition variables of a specified If node; output is a list.
BranchWhenVarTrue	Get true branches for condition variable in If node; output is list of nodes.
GetBranch	Get the true or false branch nodes of a given If node; output is a list.
LoopWhenVarTrue	Get loops where the condition variable is true; output is a list.
LoopStatement	Get loop statements of a specified For node; output is a list.
DriverVar	Get driven variable (i.e., LHS) of a given assignment node; output is the variable name.
LoadVar	Get loading variables (i.e., RHS) of a given assignment node; output is a list.
LoadValue	Get loaded values from the RHS of a given assignment node; output is a list.
ChildAST	Get the AST subtree (left or right) of an assignment node; output is the corresponding child node.
LeftAST	Get the left subtree of a given assignment node's AST; output is the node.
RightAST	Get the right subtree of a given assignment node's AST; output is the node.
Case	Get Case nodes of a specified Switch node; output is a list.
CaseCount	Get the number of Case nodes for a given Switch node.
TestExpression	Get the test expression of a specified Switch node.
Default	Get the default expression of a specified Switch node.
DriverStatement	Get assignment statements for a given driven variable; output is a list.
LoadStatement	Get assignment statements where given loading variable appears; output is list.
Dimension	Get the dimension of a specified variable; output is the dimension value.
Length	Get total bit length of a variable; output is the total number of bits.
StateSum	Get the total number of states for a variable.

Table 8 Description of VeriPG-specific traversals

B Prompt Templates

B.1 Vulnerability Detection Method Extraction

The extraction prompt template for the vulnerability detection method is as follows:

Prompt Template of Vulnerability Detection Method Extraction

System Prompt:
You are now a hardware security engineer specializing in RTL vulnerability analysis.
Based on the given description of a Verilog-related vulnerability, you will provide the corresponding Verilog code vulnerability constraint condition. This condition should be usable to identify the vulnerability within Verilog code.
User Prompt:
Definition of a vulnerability constraint condition:

- A condition is a natural language description. Each condition corresponds to a condition that a signal or keyword should satisfy. One condition describes one type of vulnerability pattern.
- The condition is a description of the behavior of a Verilog keyword, which can be used to determine whether a certain type of vulnerability exists in the Verilog code.
- The content of the natural language description should be determinable through direct textual analysis of the code.
- Please generate constraints as concisely as possible and do not include code snippets.

Here is the vulnerability description that needs to be converted into a condition:*{CWE description}*
Please analyze the vulnerability description above and derive the corresponding vulnerability constraint condition.

B.2 Vulnerability Detection Path Mapping

The prompt template for the vulnerability detection path mapping is as follows:

Prompt Template of Vulnerability Detection Path Extraction

System Prompt:
You act as an expert hardware security engineer specializing in static vulnerability analysis of Verilog code. Your task is to generate detection paths containing specific keywords based on provided Verilog vulnerability detection method. These paths will be used for retrieval within VeriPG, a graph structure generated from Verilog parsing.
User Prompt:
VeriPG Overview: VeriPG is a unified graph structure integrating the Abstract Syntax Tree (AST), Data Dependency Graph (DDG), and Control Flow Graph (CFG) derived from Verilog. It is used for modeling and static analysis of Verilog vulnerabilities. The list of keyword nodes is as follows: *{Description of Traversal Primitives}*
Vulnerability Detection Method to Analyze: *{Vulnerability Detection Method}*
Please first extract the necessary VeriPG keyword nodes from the detection method above, and then describe these detection method using these keyword nodes and their corresponding attributes.
Output Requirements:

- Output only the final detection path using the keyword nodes. When describing signal assignments, specify the signal name using the name attribute of the Identifier node whenever possible.
- The output format must be a natural language description.
- If multiple detection paths exist, enclose each path within <path>path content</path> tags.
- The output detection paths should be as concise as possible while accurately describing the issue.

B.3 TDP Generation

The generation prompt template for the vulnerability traversal rule is as follows:

Prompt Template of Vulnerability Rule Generation

System Prompt:
You are now a hardware security engineer specializing in RTL code vulnerability analysis.
Based on the given Verilog vulnerability detection path, you need to construct a vulnerability traversal rule by composing VeriPG traversal primitives. This traversal rule will be used within VeriPG to perform vulnerability detection.
User Prompt:
Overview of VeriPG:
VeriPG is a graph-based structure composed of an AST, DDG, and CFG parsed from Verilog. It is used to model Verilog code and perform static vulnerability analysis. Each node in VeriPG corresponds to a Verilog keyword. Node types include: DeclVars, Input, Output, Reg, Wire, Width, IntConst, Assign, SensList, Sens, Identifier, If, For, Switch, Case, Default, <= (Nonblocking), = (Blocking), Land, Lor, Unot.
Overview of VeriPG Traversal Primitives:
VeriPG traversal rules should be constructed by nesting the following traversal primitives. Traversal primitives are the fundamental methods to explore VeriPG. Each node has a globally unique node ID and variable name. The traversal primitive details are as follows: *{Description of Traversal Primitives}*
Overview of VeriPG Vulnerability Traversal Rules:
Vulnerability traversal rules are composed by nesting traversal primitives and saved in JSON format. The composition rules are as follows: *{TDP Format}*
You are required to compose a VeriPG vulnerability traversal rule using the above format, so that it can be used to locate the vulnerable nodes in the VeriPG graph. Vulnerability Constraint Path to be Detected: *{Vulnerability Detection Path}*
Output Example: *{TDP Example} {Explanation of TDP Example}*
Output Requirements:

- Please generate the vulnerability traversal rule, and then validate the rule by sequentially verifying each traversal primitive (Prim) used in the rule.
- The validation must be done using a Python-based rule validation tool. The tool takes a list of consecutive traversal primitive names as input and returns whether the current sequence is valid.

C Algorithm of TDP Executor

This section presents the pseudocode for the execution engine, demonstrating the deterministic process of translating static rules into dynamic detection queries.

Algorithm 1 delineates the Primitive Executor, the engine's atomic unit, which employs recursive parsing to handle nested queries and invokes the Filter Executor immediately following graph traversal to ensure strict adherence to predefined constraints.

Algorithm 1 Primitive Executor

Input: G (VeriPG), P (Primitive Object in Vulnerability Rule), N (Current Nodes in VeriPG)
Output: Result (Executing Result)

```
if Parameter_P is Primitive/Path Object then
    Get true Parameter from Primitive/Path executor.
end if
Result = Execute(G, Name_P, Parameter, N)
if Filter_P is not null then
    Result = FilterExecutor(G, Filter_P, N)
end if
return Result
```

Algorithm 2 details the Path Executor, which functions as a sequence manager; by iteratively processing ordered steps and passing the output of the preceding step as the input context for the next, this chaining mechanism mimics the cognitive workflow of a manual code audit, ensuring logical continuity throughout the detection process.

Algorithm 2 Path Executor

Input: G (VeriPG), PT (Path Object in Vulnerability Rule), N (Current Nodes in VeriPG)
Output: Result (Executing Result)

```
Result = N
for each step in PO do
    Result = PrimitiveExecutor(G, step, Result)
end for
return Result
```

Finally, Algorithm 3 defines the Filter Executor, the engine's logical filtering component, which applies composite Boolean conditions to the node set and retains only those nodes satisfying all criteria, thereby facilitating precise vulnerability localization and minimizing false positives.

Algorithm 3 Filter Executor

Input: G (VeriPG), FT (Filter Object in Vulnerability Rule), N (Current Nodes in VeriPG)
Output: Result (Executing Result)

```
for each node in N do
    for each condition in FT do
        Get condResult from Primitive/Path executor.
        CondResults.append(condResult)
    end for
    if CombineCondResults(CondResults, Operator_FT) then
        Result.append(node)
    end if
end for
return Result
```