SECURITY-IMPACT-VALIDATED TRACES FOR MODULE-LEVEL RTL VULNERABILITY DETECTION WITH LARGE LANGUAGE MODELS

Anonymous authors
Paper under double-blind review

ABSTRACT

Detecting security weaknesses at the register-transfer level (RTL) can reduce costly post-fabrication repairs. Large language models (LLMs) offer automated detection, but require module-level data with reliable security labels. We present a Trace-aware detection approach and a data-construction pipeline combining Common Weakness Enumeration (CWE) scenario identification, structure-adaptive semantic mutation, whole-module syntax checks, and CWE-specific security-impact validation. The pipeline produces 2,444 deduplicated secure–vulnerable pairs across seven open-source platforms and eight hardware CWEs. Aligned views provide code-only inputs (L0) or code with Security Trace annotations (L1): validation-derived code blocks, control conditions, signal relationships, and statically resolvable dependencies, excluding endpoint values that could reveal labels. We compare code-only detection (No Trace), reference evidence (Gold Trace), and model-generated evidence (Self-Generated Trace), using source-lineage-isolated splits and platform holdout to assess detection and generalization. On 492 test modules, four models with source-ordered chunk aggregation achieve 96.54%–97.76% accuracy with Gold Trace; replacing it with generated Trace while retaining the same reference-trained classifiers yields 85.77%–92.68%. These results support the utility of structured security evidence while highlighting the gap between reference and generated Traces.

1 INTRODUCTION

Security mechanisms in modern systems-on-chip (SoCs), including access control, register protection, and debug authorization, are critical to hardware security (MITRE, n.d.). Detecting weaknesses at the register-transfer level (RTL) before fabrication reduces the need for costly downstream remediation (Ardeshiricham et al., 2017; Trippel et al., 2022). Existing approaches include information-flow tracking, hardware fuzzing, property-based verification, and static analysis (Ardeshiricham et al., 2017; Trippel et al., 2022; Rogers et al., 2025; Ayalasomayajula et al., 2024; Ahmad et al., 2025), but often require design-specific properties, annotations, test environments, or rules. Recent work explores large language models (LLMs) for hardware debugging and policy generation (Fu et al., 2023; Tarek et al., 2025b), vulnerability-data construction (Saha et al., 2024b), and detection through fine-tuning, program analysis, or retrieval-augmented agents (Tarek et al., 2025a; Long et al., 2026; Hasan et al., 2026).

Reliable learning and evaluation nevertheless require more than syntactically valid examples. Existing datasets range from generated finite-state machines to expanded SoC vulnerability seeds, while evaluations also use limited collections of real RTL. Relatively few studies jointly address real-design coverage, module-level context, and executable security-impact validation (Saha et al., 2024b; Tarek et al., 2025a; Hasan et al., 2026). A syntax check cannot establish that a mutation induces the security impact specified by the target Common Weakness Enumeration (CWE) entry. Moreover, RTL weaknesses may depend on control and data relationships across concurrent statements, which isolated code fragments can miss (Long et al., 2026).

Evaluation must also account for source reuse: variants of one module or reused hardware blocks can create highly similar training and test samples, inflating apparent generalization (Ding et al.,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

2025). Nor does a correct prediction alone establish a reliable explanation (Ullah et al., 2024). These concerns motivate our central question: **Can structured evidence derived from CWE-specific security-impact validation help LLMs detect module-level RTL vulnerabilities?**

We construct secure–vulnerable module pairs from real open-source RTL through CWE scenario identification and structure-adaptive semantic transformations, retaining pairs that pass whole-module syntax checks and target security-impact tests. Within each accepted validation scope, we organize code blocks, control conditions, signals, and statically resolvable dependencies into a **Security Trace (Trace)**. These annotations describe intra-module evidence, not complete SoC-level exploit paths.

Two aligned views support comparison: **L0** contains code and the detection task; **L1** adds reference Trace while preserving code, labels, and partitions. Endpoint-value fields that could reveal the label are omitted. We evaluate **No Trace** (code only), **Gold Trace** (dataset-provided reference evidence), and **Self-Generated Trace** (Self Trace, predicted from code), with complete modules as the supervision and prediction units. We call the Trace-generation model the Generator and the vulnerability classifier the Answerer. Gold and Self Answerers are trained using reference and generated Traces, respectively. We also test Gold Answerers with generated Traces, requiring no reference Trace at test time. Generated Traces remain model predictions rather than automatically validated evidence.

Our contributions are:

1. **A security-impact-validated RTL dataset and construction pipeline.** We provide 2,444 deduplicated secure–vulnerable pairs across seven platforms and eight hardware CWEs, accepted through both whole-module syntax checks and CWE-specific security-impact tests.
2. **Security Trace and aligned L0/L1 views.** Annotations linked to source code provide validation-derived security evidence for learning and controlled code-only versus Trace-assisted comparisons.
3. **Trace-aware module-level detection and empirical evaluation.** In an exploratory system-level comparison on the lineage-isolated test set, generated Trace paired with a Gold Answerer achieves 85.77%–92.68% accuracy across four models, exceeding three commercial models accessed through application programming interfaces (APIs; 50.41%–54.88%) and unadapted open-source baselines (48.78%–50.00%) on the same test set. Ablations further show the importance of correctly paired Trace for Answerer predictions.

2 Related Work

2.1 RTL/SoC Security Analysis and Validation

Hardware information-flow analysis checks compliance with security policies (Hu et al., 2021), through compile-time security types (Zhang et al., 2015) or RTL-level information-flow tracking, including CellIFT (Ardeshiricham et al., 2017; Solt et al., 2022). Hardware fuzzing explores behavior using generated stimuli, including software fuzzing of translated RTL (Trippel et al., 2022) and coverage-guided instruction generation with golden-model comparison in TheHuzz (Kande et al., 2022). HardFails highlights software-exploitable, cross-layer weaknesses and limitations of design-time analysis (Dessouky et al., 2019). Our pipeline instead uses targeted intra-module simulation to validate constructed samples and organize their evidence; it is not a new fuzzing or information-flow algorithm.

2.2 RTL Vulnerability Data and Security-Oriented Data Construction

MITRE Hardware CWE defines weakness categories (MITRE, n.d.), while Don't CWEAT It examines their amenability to early static analysis (Ahmad et al., 2022). SoC vulnerability benchmarks (Tarek et al., 2023) and resources containing SystemVerilog security properties and source snapshots (Rogers et al., 2025) support scenario and validation-target identification. For LLM training, Vul-FSM generates vulnerable finite-state machines with fidelity checks (Saha et al., 2024b), whereas BugWhisperer expands CVA6/SoC vulnerability seeds using proprietary LLMs (Tarek et al., 2025a). We construct paired samples directly from real RTL and retain complete-module context.

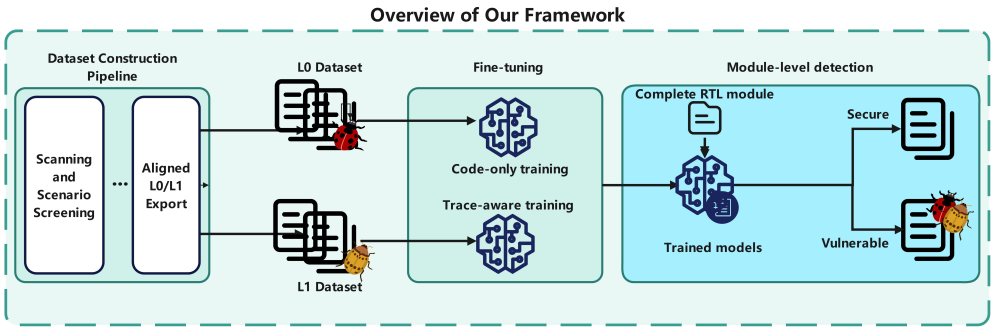

Figure 1: L0/L1 support code-only and Trace-aware fine-tuning. Detection settings differ: No Trace uses code alone; Gold Trace adds reference evidence; Self-Generated Trace generates evidence from code. Trace-aware Answerers receive code and Trace.

Validation must establish more than syntactic correctness. Encarsia uses transformations based on real processor bugs and formal methods to check observable architectural deviations (Bölcskei et al., 2025). Our approach combines CWE-specific intra-module simulation with paired data and aligned security evidence, without claiming whole-design formal guarantees. Separately, PrimeVul documents evaluation distortions from duplicate samples and dataset construction (Ding et al., 2025); our lineage-aware partitions keep pairs intact while isolating related source designs.

2.3 LLM-Based RTL Security Analysis and Structured Evidence

LLMs support hardware debugging (Fu et al., 2023; Akyash & Kamali, 2024), assertion and policy generation (Kande et al., 2024; Ayalasomayajula et al., 2024; Tarek et al., 2025b), and broader SoC security tasks (Saha et al., 2024a). Detection approaches include fine-tuned BugWhisperer (Tarek et al., 2025a), LASHED's filtering and explanation of static-analysis warnings (Ahmad et al., 2025), VerilogLAVD's semantically validated, LLM-generated traversal patterns over program-analysis structures (Long et al., 2026), and retrieval-augmented multi-agent SecureRAG-RTL (Hasan et al., 2026). Unlike executable detection patterns, our Traces annotate sample-specific evidence within accepted validation scopes.

Research on intermediate reasoning and process supervision examines how intermediate steps support predictions (Wei et al., 2022; Lightman et al., 2024); our source-linked Traces are neither free-form chain-of-thought nor manually annotated step-level supervision. Studies of vulnerability explanations and reasoning faithfulness—whether explanations reflect the basis of a prediction—caution that correct predictions need not imply reliable rationales (Ullah et al., 2024; Turpin et al., 2023; Matton et al., 2025). We therefore distinguish reference from generated Trace and evaluate detection utility without equating it with evidence correctness or faithful reasoning.

3 Method

3.1 Framework Overview

Our framework constructs validated secure–vulnerable RTL pairs and aligned L0/L1 views, then uses these data to fine-tune code-only and Trace-aware models for module-level detection (Figure 1). Downstream evaluation assesses dataset utility, not sample validity.

3.2 Dataset Construction Methodology

The pipeline checks candidate relevance, applies CWE-aligned mutations, and validates their security impact (Figure 2). Authors specify occurrence scenarios and mutation semantics; executable rules implement these specifications.

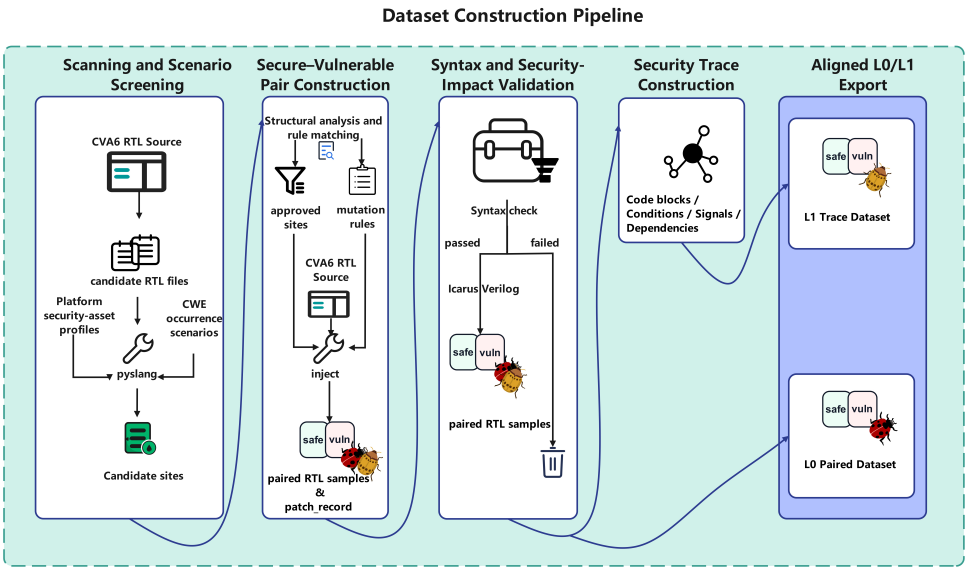

Figure 2: Construction illustrated with CVA6. `pyslang` parsing supports separate screening and rule matching. Pairs must pass both whole-module syntax checks and CWE-specific impact tests; validation evidence supplies Trace annotations for aligned L0/L1 export.

3.2.1 CWE Scenario Identification

Per-platform file manifests select Verilog/SystemVerilog design files from open-source RTL projects, excluding testbenches, design-verification code, formal-verification support code, and software from mutation candidates. Using `pyslang`, we record source locations, modules, abstract syntax tree (AST) types, identifiers, assignment or condition expressions, and enclosing guards.

Because platforms name similar security objects differently, we organize scanned module and signal names for locks, permission controls, and protected registers into platform security-asset profiles. These profiles translate author-defined occurrence scenarios, grounded in MITRE CWE definitions (MITRE, n.d.), into platform-specific screening criteria. We apply the criteria to candidate sites and their associated controls and statements within the same module, retaining matching locations and CWEs. This checks scenario applicability, not vulnerability existence; a site may initially match multiple scenarios.

3.2.2 Security-Semantic Pair Construction

We first examine the code structure and security objects protected at each candidate site, then select mutation rules that can introduce the target CWE weakness. For lock protection, for example, the transformation depends on whether the lock condition appears in a branch guard or an assignment expression. Each site is assigned to one CWE; multiple applicable rules are applied separately to construct distinct secure–vulnerable pairs.

3.2.3 CWE-Specific Security-Impact Validation

After `pyslang` syntax checks on both complete modules, we construct testbenches within CWE-specific intra-module scopes and compare behavior under identical stimuli using Icarus Verilog. Scope follows the security property, not a fixed source-line window: a lock-controlled write enable can be checked at its local gate expression, whereas access control spanning multiple stages requires the relevant intra-module control path.

We retain syntactically valid pairs exhibiting the expected security-impact difference. The validated scope bounds the Trace. Local combinational tests may use scalarized signal abstractions; conclusions are limited to the checked expressions or paths, abstractions, and stimuli, not whole-system security.

3.2.4 SECURITY TRACE AND DATA REPRESENTATION

Trace annotations contain source-linked `code_blocks`, signal `nodes`, control `conditions`, statically resolvable dependency `edges`, and endpoint metadata. They describe evidence within the validated scope, not complete chip-level information flow or free-form reasoning. We remove `value_observations` from the main training and evaluation representation because it could reveal the security outcome.

L0 stores the detection `question` and complete-module `code` as input, and `verdict` and `cwe_id` as targets. L1 adds a reference Trace annotation to the same record. During task preparation, this annotation becomes Generator supervision or additional Gold Answerer input. Labels, outcome summaries, and pair/provenance metadata are excluded from detection inputs.

L0/L1 example. The excerpts below show a CWE-1233 vulnerable sample from CVW's `csrm` module after removing `value_observations`. Ellipses abbreviate content; `<same as L0>` indicates identical inputs. Actual records retain the complete module.

L0

```
{"input": {"question": "Analyze the following complete ...",
           "code": "module csrm ... endmodule"},
 "output": {"verdict": "vulnerable", "cwe_id": "CWE-1233"}}
```

L1

```
{"input": {"question": "<same as L0>", "code": "<same as L0>"},
 "output": {"verdict": "vulnerable", "cwe_id": "CWE-1233",
   "trace": {
     "code_blocks": [{"block_id": "B1", ...}],
     "trace_graph": {
       "nodes": [{"node_id": "N1", "signal": "WritePMPCFGM[i]",
                  "code_block_ids": ["B1"]}],
       "edges": [],
       "conditions": [{"condition_id": "C1", "expression": "...",
                       "controlled_node_ids": ["N1"], ...}],
       "terminal_node_id": "N1", "path_complete": true
     }}, ...}}
```

The write-enable node is itself the endpoint (`terminal_node_id`), so no intermediate edges are needed. `path_complete` refers only to completeness within the extracted validation scope.

3.3 DOWNSTREAM MODEL TRAINING AND EVALUATION

We fine-tune four open-source models using quantized low-rank adaptation (QLoRA); configurations appear in Section 4.2. Chunks are overlapping code segments used for model processing, distinct from the security-relevant `code_blocks` in a Trace. We name three input-processing strategies: **Independent-Max (IM)** takes the maximum chunk vulnerability score as the module score; **Ordered-Hierarchical (OH)** aggregates source-ordered chunk representations using a lightweight Transformer with positional encodings; **Full-context** encodes the complete input and is used only for the two Qwen models. All strategies use module-level supervision; chunks have no independent ground-truth labels.

For applicable model–context combinations, L0 trains a code-only Answerer. L1 trains a Gold Answerer on code and reference Traces, or first trains a Generator on code-to-Trace prediction and then a separate Self Answerer on code and generated Traces. Chunked Generator outputs support both IM and OH Answerers. Self-Generated Trace detection does not access reference Traces at test time.

All Answerers use paired module-level binary cross-entropy:

$$\mathcal{L}_{\text{ans}} = -\frac{1}{2}\left[\log(1-p_s) + \log p_v\right], \tag{1}$$

where p_s and p_v are independently predicted vulnerability probabilities for the secure and vulnerable versions, computed after aggregation for chunk-based strategies. The Generator uses autoregressive

cross-entropy only over the target Trace:

$$\mathcal{L}_{\text{trace}} = -\frac{1}{T}\sum_{j=1}^{T} \log P_\phi(t_j \mid c, t_{<j}), \tag{2}$$

where c is input code, ϕ denotes Generator parameters, and t_j is target token j in a length-T Trace. Generator and Self Answerer training are separate; Answerer gradients do not propagate through generated Traces to the Generator.

4 Experiments

We examine dataset coverage and source separation, downstream utility, and held-out-platform transfer. Trace controls test utility; other comparisons assess implementation and training choices. These complement sample validation (Section 3).

4.1 Dataset Characteristics and Quality

Dataset size and class distribution. The dataset covers seven platforms and eight CWEs, with 2,444 secure–vulnerable pairs and 4,888 complete-module records per L0/L1 view (Figure 3a). Class counts are constrained by applicable hardware scenarios and validation requirements. For example, CWE-1252 requires verified coexistence of write and execute permissions for the same memory region, limiting eligible construction sites to yield 15 retained pairs.

Lineage-aware splitting. A source lineage groups variants from the same or highly similar source modules; each group is assigned wholly to one partition. To audit source separation, we compare naive and lineage-aware splits. Lineage grouping lowers median nearest-training source similarity from 1.000 to 0.065 and test pairs reaching 0.95 from 239/243 to 0/246 (Figure 3b). We use lineage-aware partitions throughout the main experiments.

(a) Dataset coverage

2,444 pairs / 7 platforms / 8 CWEs

Platform	Pairs	CWE	Pairs
Carfield	1,224	CWE-1262	996
OpenTitan	418	CWE-1191	413
CV32E40S	261	CWE-1313	321
Caliptra	238	CWE-1221	299
CVA6	204	CWE-1231	186
OpenPiton	82	CWE-1260	119
CVW	17	CWE-1233	95
		CWE-1252	15

(b) Source similarity by split

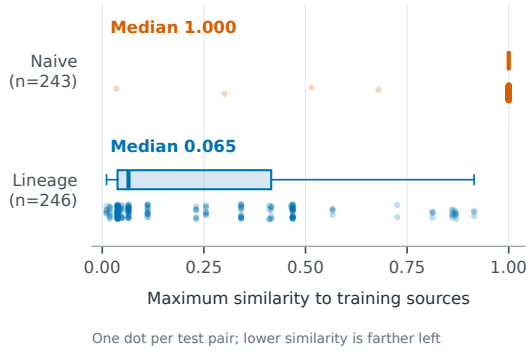

Figure 3: Dataset coverage and split audit. (a) Platform/CWE counts are independent marginals. (b) Maximum normalized token 5-gram Jaccard similarity to training per test pair. Boxes show quartiles and medians; whiskers extend to observations within 1.5 interquartile ranges. All points are retained, with vertical jitter. Naive/lineage splits contain 243/246 pairs: this compares split protocols, not identical test sets.

4.2 Experimental Setup

We use Qwen2.5-Coder-7B, Qwen3-8B, DeepSeek-Coder-6.7B, and Mistral-7B-v0.3 with QLoRA: 4-bit NormalFloat (NF4), 16-bit brain floating point (BF16), and low-rank adaptation (LoRA) rank 16. The training budget is five epochs, with seed 42, AdamW learning rate 10^{-4}, and an effective batch of eight pairs; Qwen3 thinking is disabled. Saved model weights are selected on validation

data: the highest macro-averaged area under the receiver operating characteristic curve (AUROC) for Answerers, and the lowest Trace loss for Generators. Decision thresholds are set on validation data and remain unchanged during testing. IM/OH use 12,000-token code chunks with 768-token overlap; Qwen Full-context allows up to 131,072 tokens. Self Trace uses greedy decoding, with limits of 8,192 output tokens and 24,576 total input/output tokens. We report module-level accuracy, F1 score, and Matthews correlation coefficient (MCC). Trace generation failures count as incorrect predictions; AUROC is computed only for modules with valid generated Traces.

Pairings with different training and test Trace sources in the main table and Table 2C are exploratory cross-input analyses conducted after inspecting initial test results, reusing frozen models and validation thresholds. Each experiment uses one run with seed 42; curriculum learning adds L0 training compute, while Self Trace pipelines also require Generator training and inference.

4.3 Overall Detection Performance

Deployment provides code, not Gold Trace: a Generator supplies Self Trace to a Gold Answerer. Table 1 compares this L1 pipeline with unadapted and commercial API models on 492 modules. OH requires a trained aggregator unavailable to unadapted models, which therefore use IM and zero thresholds; L1 pipelines use generated Traces, trained OH, and frozen validation thresholds. This system-level comparison is complemented by matched L0 and aggregation controls (Section 4.4).

Table 1: Detection results for unadapted models, commercial APIs, and L1-fine-tuned pipelines. L1 uses Self Trace + Gold Answerer. Bold marks the best value per metric.

Model	Accuracy (%)	F1 (%)	MCC
Unadapted models (IM)			
Qwen2.5	50.00	66.67	0.000
Qwen3	48.78	39.71	-0.026
DeepSeek	50.00	66.67	0.000
Mistral	50.00	66.67	0.000
Commercial API models (zero-shot)			
GPT-5.6 Sol	54.27	63.30	0.0980
Qwen3.8-Max (0902)	54.88	38.33	0.1156
DeepSeek V4 Pro	50.41	2.40	0.0453
L1-fine-tuned pipelines (OH)			
Qwen2.5-L1	91.06	91.13	0.821
Qwen3-L1	**92.68**	**92.11**	**0.863**
DeepSeek-L1	85.77	85.94	0.716
Mistral-L1	87.60	88.47	0.761

Across all four models, L1 pipelines improve accuracy from 48.78%–50.00% to 85.77%–92.68%. This supports the downstream utility of the complete trained pipeline; the following controls examine which design choices contribute.

Commercial APIs receive the same frozen L0 code chunks and predict binary labels zero-shot; a module is flagged if any chunk is flagged (OR aggregation). Their accuracy is 50.41%–54.88%, below all four L1 pipelines.

4.4 Trace Utility, Evidence Dependence, and Design Analysis

On the same 492 modules, Table 2 tests Trace utility (A), evidence dependence (B), and Answerer training source (C); aggregation (D) and curriculum (E) are secondary. Gold and Self Answerers are trained on code paired with Gold and Self Trace, respectively; the table separately identifies the Trace supplied at test time.

A. Does Trace-assisted training improve detection? With model and OH aggregation matched, Gold Trace reaches 96.54%–97.76% accuracy; generated Trace exceeds L0 by 5.28–27.85 percentage points. The former evaluates reference evidence, whereas the latter demonstrates utility without supplied test-time Traces, at the additional cost of a Generator.

Table 2: Ablation results: Accuracy (%), higher is better. Bold indicates the higher value within each matched comparison for each model; ties are both bold.

Experimental setting		Qwen2.5	Qwen3	DeepSeek	Mistral
A. Code-only versus Trace-assisted detection (OH)					
L0		82.11	87.40	57.93	77.24
L1: Gold Trace + Gold Answerer		**96.95**	**97.76**	**97.76**	**96.54**
L1: Self Trace + Gold Answerer		91.06	92.68	85.77	87.60
B. Trace input controls (Gold Answerer, OH)					
Original Gold Trace		**96.95**	**97.76**	**97.76**	**96.54**
Trace removed		50.00	50.00	50.20	50.00
Trace shuffled		51.42	51.02	52.03	51.22
C. Same test input, different Answerers (OH)					
Test input	Answerer				
Gold Trace	Gold Answerer	**96.95**	**97.76**	**97.76**	**96.54**
Gold Trace	Self Answerer	90.24	91.26	88.82	85.98
Self Trace	Gold Answerer	**91.06**	**92.68**	**85.77**	**87.60**
Self Trace	Self Answerer	86.59	85.57	81.30	81.50
D. Input organization					
Data	Input organization				
L0	IM	**84.35**	81.10	**63.62**	44.31
L0	OH	82.11	**87.40**	57.93	**77.24**
L0	Full-context	75.41	77.44	—	—
L1 Gold	IM	**98.17**	96.75	96.95	95.94
L1 Gold	OH	96.95	**97.76**	**97.76**	**96.54**
L1 Gold	Full-context	97.97	96.75	—	—
E. Direct training versus curriculum learning (OH)					
Train/test Trace	Training order				
Gold Trace	L1 only	**96.95**	**97.76**	**97.76**	**96.54**
Gold Trace	L0 then L1	95.12	**97.76**	96.34	**96.54**
Self Trace	L1 only	86.59	85.57	**81.30**	81.50
Self Trace	L0 then L1	**87.40**	**92.89**	79.27	**83.94**

B. Does detection depend on matched evidence? Holding the Gold Answerer, code, labels, and threshold fixed, we empty the test Trace or replace it with a Trace from another pair (seed 42), without retraining. Accuracy falls from about 97% to 50% after removal and 51%–52% in all four shuffling tests. This indicates dependence on matched Trace input, but input-distribution shifts mean it does not alone establish security-semantic understanding.

C. Which training Trace source suits generated evidence? With test inputs fixed, Gold Answerers outperform Self Answerers across models, leading by 4.47–7.11 percentage points on generated Traces. This exploratory comparison motivates the main-table pairing, not an independently validated selection rule. Both Answerer types perform better with reference Traces, suggesting room to improve generated evidence.

D. Does input organization change the findings? IM versus OH has model-dependent effects on L0: Qwen2.5/DeepSeek favor IM and Qwen3/Mistral favor OH. With L1 Gold, both reach 95.94%–98.17% accuracy, differing by at most 1.22 percentage points per model. Across the four models, OH has higher mean accuracy than IM for both L0 and L1 Gold.

E. Does prior L0 training help? Curriculum training initializes the L1 Answerer's adapter and OH aggregator from its L0 model; it does not retrain the Generator. With five L1 epochs in both settings, adding L0 training yields no gain under Gold Trace. Under Self Trace, it improves three models but reduces DeepSeek accuracy; Qwen3 gains most (7.32 percentage points). Curriculum benefits are thus model-dependent, not a prerequisite for Trace utility.

4.5 Cross-Platform Generalization to OpenTitan

To test transfer beyond training platforms, we reserve OpenTitan exclusively for testing and exclude its lineages from the other platforms. Lineage-separated training/validation/test sets contain 1,699/190/418 pairs (836 test modules). All four models use OH and select checkpoints and thresholds only on non-OpenTitan validation data. Table 3 compares Gold Trace + Gold Answerer with Self Trace + Self Answerer, not the main table's generated-Trace + Gold Answerer pairing.

Table 3: Cross-platform detection on OpenTitan. Accuracy and F1 are percentages; bold marks the higher value for each model and metric.

	Gold Trace + Gold Answerer			Self Trace + Self Answerer		
Model	Accuracy	F1	MCC	Accuracy	F1	MCC
Qwen2.5	**98.80**	**98.79**	**0.976**	90.55	90.89	0.813
Qwen3	**98.45**	**98.42**	**0.969**	88.88	88.59	0.779
DeepSeek	**95.33**	**95.23**	**0.908**	82.66	82.84	0.653
Mistral	**96.77**	**96.73**	**0.936**	91.03	91.12	0.821

Gold and Self routes reach 95.33%–98.80% and 82.66%–91.03% accuracy, respectively, supporting transfer to this platform held out from task training. The 5.74–12.67-point gap compares complete pipelines: both Trace and Answerer change, so it cannot be attributed solely to Trace generation.

4.6 Validation Performance During Training

Does reference evidence help the Answerer reach high validation accuracy earlier? Figure 4 compares code-only L0 and L1 Gold with matched models, OH aggregation, and data partitions.

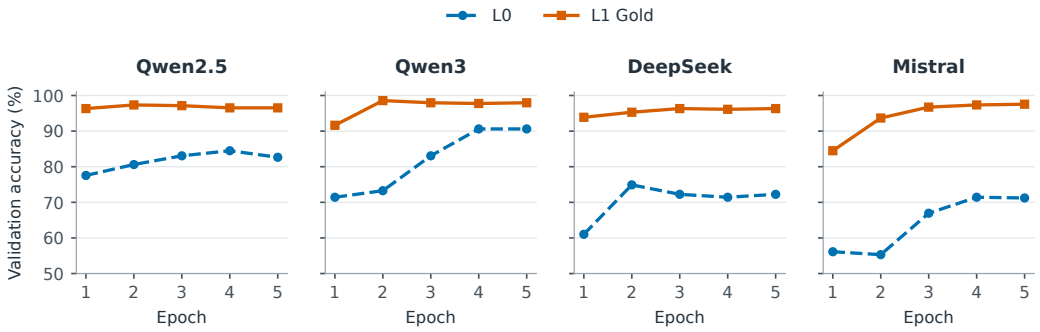

Figure 4: Validation accuracy over five epochs: 1,953 training pairs, 490 validation modules, OH, seed 42. Thresholds are calibrated per epoch; points are actual epoch results, not running maxima. L1 uses Gold Trace.

Qwen2.5, Qwen3, DeepSeek, and Mistral reach 95% validation accuracy with Gold Trace at epochs 1, 2, 2, and 3; no L0 model does so within five epochs. Reference evidence thus reduces epochs to this validation target.

5 Conclusion

We construct 2,444 secure–vulnerable RTL pairs across seven platforms and eight CWEs, with CWE-specific security-impact validation and aligned code-only and Security Trace views. Experiments show higher detection accuracy with reference Traces than with code alone; generated Traces remain less effective. These findings support structured security evidence as a useful aid to module-level RTL vulnerability detection, within the evaluated settings. Platform coverage, CWE coverage, class imbalance, and intra-module validation limit the dataset's scope. Future work will expand coverage, investigate cross-module security impact, and study transfer to unseen platforms and CWE types without target-CWE-specific fine-tuning.

AI USE STATEMENT

We used OpenAI Codex as a generative artificial intelligence (AI) assistant to discuss and refine the research methodology and experimental design; implement and debug portions of the data-construction and validation pipeline; clean and structure data; retrieve and organize related literature; and draft, translate, and edit the manuscript. The authors defined the CWE occurrence scenarios and intended mutation semantics. Codex assisted in implementing the corresponding screening predicates and mutation code; the authors reviewed the implementation and made the final decisions. This assistance does not replace the automated syntax and security-impact checks used to accept samples. The authors made the final decisions regarding the research questions, CWE scope, data-construction principles, experimental protocol, technical choices, and conclusions. The authors reviewed all AI-assisted code, analyses, text, and other artifacts incorporated into this work and take full responsibility for the final content, claims, and experimental results.

ETHICS STATEMENT

This study uses public open-source RTL projects and involves neither human participants nor private data. The research-derived samples have potential dual use and are intended for defensive analysis and hardware-security evaluation. They are not claimed to be exploitable flaws in deployed products. We did not attack deployed hardware or provide exploitation procedures for operational systems. Redistribution must respect upstream licenses and attribution requirements.

REPRODUCIBILITY STATEMENT

Sections 3 and 4 describe data construction and evaluation. The supplementary material provides 20 anonymized secure–vulnerable pairs from the main training partition (10 each from Caliptra and OpenTitan), with aligned L0/L1 views, recorded validation evidence, and a data-integrity checker. This illustrative subset covers six CWEs; it does not reproduce the full experiments or establish small-data training performance. Implementation code and model weights are not included. We will release the full dataset and split manifests via GitHub upon acceptance, in accordance with upstream licenses, and include the repository link in the accepted paper.

REFERENCES

Baleegh Ahmad, Wei-Kai Liu, Luca Collini, Hammond Pearce, Jason M. Fung, Jonathan Valamehr, Mohammad Bidmeshki, Piotr Sapiecha, Steve Brown, Krishnendu Chakrabarty, Ramesh Karri, and Benjamin Tan. Don't CWEAT it: Toward CWE analysis techniques in early stages of hardware design. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–9, 2022. doi: 10.1145/3508352.3549369.

Baleegh Ahmad, Hammond Pearce, Ramesh Karri, and Benjamin Tan. LASHED: LLMs and static hardware analysis for early detection of RTL bugs. *arXiv preprint arXiv:2504.21770*, 2025. URL https://arxiv.org/abs/2504.21770.

Mohammad Akyash and Hadi Mardani Kamali. Self-HWDebug: Automation of LLM self-instructing for hardware security verification. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 391–396, 2024. doi: 10.1109/ISVLSI61997.2024.00077.

Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pp. 1691–1696, 2017. doi: 10.23919/DATE.2017.7927266.

Avinash Ayalasomayajula, Rui Guo, Jingbo Zhou, Sujan Kumar Saha, and Farimah Farahmandi. LASP: LLM assisted security property generation for SoC verification. In *ACM/IEEE International Symposium on Machine Learning for CAD (MLCAD)*, pp. 1–7, 2024. doi: 10.1145/3670474.3685967.

Matej Bölcskei, Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. Encarsia: Evaluating CPU fuzzers via automatic bug injection. In *34th USENIX Security Symposium (USENIX Security 25)*, pp. 2577–2593. USENIX Association, 2025. URL https://www.usenix.org/conference/usenixsecurity25/presentation/bolcskei.

Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. HardFails: Insights into software-exploitable hardware bugs. In *28th USENIX Security Symposium (USENIX Security 19)*, pp. 213–230. USENIX Association, 2019. URL https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky.

Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David Wagner, Baishakhi Ray, and Yizheng Chen. Vulnerability detection with code language models: How far are we? In *IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 1729–1741, 2025. doi: 10.1109/ICSE55347.2025.00038.

Weimin Fu, Kaichen Yang, Raj Gautam Dutta, Xiaolong Guo, and Gang Qu. LLM4SecHW: Leveraging domain-specific large language model for hardware debugging. In *Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pp. 1–6, 2023. doi: 10.1109/AsianHOST59942.2023.10409307.

Touseef Hasan, Blessing Airehenbuwa, Nitin Pundir, Souvika Sarkar, and Ujjwal Guin. SecureRAG-RTL: A retrieval-augmented, multi-agent, zero-shot LLM-driven framework for hardware vulnerability detection. *arXiv preprint arXiv:2603.05689*, 2026. URL https://arxiv.org/abs/2603.05689.

Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. Hardware information flow tracking. *ACM Computing Surveys*, 54(4):1–39, 2021. doi: 10.1145/3447867.

Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. TheHuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3219–3236. USENIX Association, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/kande.

Rahul Kande, Hammond Pearce, Benjamin Tan, Brendan Dolan-Gavitt, Shailja Thakur, Ramesh Karri, and Jeyavijayan Rajendran. (security) assertions by large language models. *IEEE Transactions on Information Forensics and Security*, 19:4374–4389, 2024. doi: 10.1109/TIFS.2024.3372809.

Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In *International Conference on Learning Representations (ICLR)*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/aca97732e30bcf1303bc22ac3924fd16-Abstract-Conference.html.

Xiang Long, Yingjie Xia, Li Kuang, Yao Wan, and ZiHao Liu. VerilogLAVD: LLM-aided pattern generation for Verilog CWE detection. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics*, pp. 28292–28308, 2026. doi: 10.18653/v1/2026.acl-long.1304.

Katie Matton, Robert Ness, John Guttag, and Emre Kiciman. Walk the talk? measuring the faithfulness of large language model explanations. In *International Conference on Learning Representations (ICLR)*, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/hash/b5ec50eb177908f21f78ed0d76ed525c-Abstract-Conference.html.

MITRE. CWE-1194: Hardware design. Common Weakness Enumeration, n.d. URL https://cwe.mitre.org/data/definitions/1194.html. Accessed 2026-09-14.

Jayden Rogers, Niyaz Shakeel, Xiao Tan, Samantha Espinosa, Divya Mankani, Cade Chabra, Kaki Ryan, and Cynthia Sturton. Hardware security benchmarks for open-source SystemVerilog designs. In *IEEE Secure Development Conference (SecDev)*, pp. 196–203, 2025. doi: 10.1109/SecDev66745.2025.00031.

Dipayan Saha, Shams Tarek, Katayoon Yahyaei, Sujan Kumar Saha, Jingbo Zhou, Mark Tehranipoor, and Farimah Farahmandi. LLM for SoC security: A paradigm shift. *IEEE Access*, 12:155498–155521, 2024a. doi: 10.1109/ACCESS.2024.3427369.

Dipayan Saha, Katayoon Yahyaei, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. Empowering hardware security with LLM: The development of a vulnerable hardware database. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 233–243, 2024b. doi: 10.1109/HOST55342.2024.10545393.

Flavien Solt, Ben Gras, and Kaveh Razavi. CellIFT: Leveraging cells for scalable and precise dynamic information flow tracking in RTL. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2549–2566. USENIX Association, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/solt.

Shams Tarek, Hasan Al Shaikh, Sree Ranjani Rajendran, and Farimah Farahmandi. Benchmarking of SoC-level hardware vulnerabilities: A complete walkthrough. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 1–6, 2023. doi: 10.1109/ISVLSI59464.2023.10238612.

Shams Tarek, Dipayan Saha, Sujan Kumar Saha, and Farimah Farahmandi. BugWhisperer: Fine-tuning LLMs for SoC hardware vulnerability detection. In *IEEE VLSI Test Symposium (VTS)*, pp. 1–5, 2025a. doi: 10.1109/VTS65138.2025.11022958.

Shams Tarek, Dipayan Saha, Sujan Kumar Saha, Mark Tehranipoor, and Farimah Farahmandi. SoCureLLM: An LLM-driven approach for large-scale system-on-chip security verification and policy generation. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pp. 335–345, 2025b. doi: 10.1109/HOST64725.2025.11050068.

Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3237–3254. USENIX Association, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/trippel.

Miles Turpin, Julian Michael, Ethan Perez, and Samuel Bowman. Language models don't always say what they think: Unfaithful explanations in chain-of-thought prompting. In *Advances in Neural Information Processing Systems*, volume 36, pp. 74952–74965, 2023. doi: 10.52202/075280-3275.

Saad Ullah, Mingji Han, Saurabh Pujar, Hammond Pearce, Ayse Coskun, and Gianluca Stringhini. LLMs cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In *IEEE Symposium on Security and Privacy*, pp. 862–880, 2024. doi: 10.1109/SP54263.2024.00210.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pp. 24824–24837, 2022. doi: 10.52202/068431-1800.

Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 503–516, 2015. doi: 10.1145/2694344.2694372.