HardSecBench: Benchmarking the Security Awareness of LLMs for Hardware Code Generation

Qirui Chen[1], **Jingxian Shuai**[1], **Shuangwu Chen**[1*], **Shenghao Ye**[1], **Zijian Wen**[1], **Xufei Su**[2], **Jie Jin**[3], **Jiangming Li**[4], **Jun Chen**[4], **Xiaobin Tan**[1] and **Jian Yang**[1]
[1]University of Science and Technology of China, China
[2]Xi'an Jiaotong University, China
[3]Nanjing University, China
[4]ZTE Corporation, China
{chenqirui, vexertron_shuai}@mail.ustc.edu.cn, chensw@ustc.edu.cn

Abstract

Large language models (LLMs) are increasingly used for hardware and firmware code generation, but existing studies primarily evaluate functional correctness while largely overlooking security. However, LLM-generated code that appears functionally sound may embed security flaws which could induce catastrophic damages after deployment. This critical research gap motivates us to design a benchmark for assessing security awareness under realistic specifications. In this work, we introduce **HardSecBench**, a benchmark with 924 tasks spanning Verilog Register Transfer Level (RTL) and firmware-level C, covering 76 hardware-relevant Common Weakness Enumeration (CWE) entries. Each task includes a structured specification, a secure reference implementation, and executable tests. To automate artifact synthesis, we propose a multi-agent pipeline that decouples synthesis from verification and grounds evaluation in execution evidence, enabling reliable evaluation. We evaluate diverse LLMs and find that they often satisfy functional requirements while leaving security risks. We also find that security results vary with prompting. These findings highlight pressing challenges and offer actionable insights for future advancements in LLM-assisted hardware design. Our data and code are available at https://github.com/chenqirui2002/HardSecBench.

1 Introduction

Large Language Models (LLMs) have been increasingly incorporated into practical hardware and firmware development for automated code generation, including Register Transfer Level (RTL) design and C-based firmware development [Liu *et al.*, 2024; Yu *et al.*, 2025b]. Extensive studies have focused on improving and evaluating the functional correctness of LLM-generated code [Liu *et al.*, 2023; Lu *et al.*, 2024]. However, as security intent is often not explicitly specified in LLM-assisted hardware development, the generated codes may pass functional validation yet still harbor potential security vulnerabilities that may induce catastrophic damages [Perry *et al.*, 2023; Fan *et al.*, 2025]. For example, in Figure 1, an LLM-generated configuration register successfully passes read/write tests, but violates the write-once lock policy when an alternative access interface fails to enforce the lock. As designs scale up and interfaces proliferate, such security violations may accumulate over the entire development lifecycle, thereby exacerbating systemic risks.

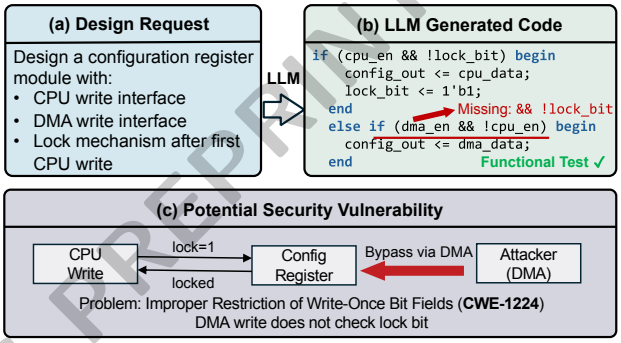


Figure 1: Potential security vulnerability in LLM-generated code. The LLM produces functionally correct code (b) from a design request (a), but the DMA interface fails to check the lock status, enabling attackers to bypass write-once protection (c).

However, there is still a lack of standardized and comprehensive benchmark to evaluate the systematic security of LLM-based hardware designs, as existing studies [Fan *et al.*, 2025; Hu *et al.*, 2025; Long *et al.*, 2025] usually validate the security of LLM-generated code on their own specific test sets. Two critical challenges hinder the development of a unified benchmark for systematic security assessment.

Trustworthy benchmark construction at scale. High-quality security benchmark samples are expensive to build and validate, making it difficult to expand evaluation to large benchmarks through expert effort alone [Qiu *et al.*, 2024]. To mitigate this inefficiency, LLM-assisted synthesis offers a practical way to generate benchmark tasks in large numbers. However, ensuring the reliability of the resulting benchmark and evaluation remains challenging, requiring rigorous qual-

* Corresponding author.

ity control to keep results trustworthy [Zhu *et al.*, 2025].

Unreliable security evaluation. Existing RTL security analysis workflows often rely on experts to craft SystemVerilog Assertions (SVAs) from security intent and debug violations through simulation [Orenes-Vera *et al.*, 2021; Afsharmazayejani *et al.*, 2024]. However, SVAs can miss vulnerabilities when security-relevant behaviors are not triggered. To reduce reliance on simulation scenarios, some workflows use early-stage static rule checking, which still demands security expertise and often emphasizes functional issues over security properties [Synopsys, 2015]. Recent work also adopts LLM-as-a-judge verifiers, but without timing-accurate simulation evidence their judgments can be unreliable and inconsistent across runs [Bhatt *et al.*, 2024].

To address these challenges, we design an automated pipeline that scales benchmark construction and enables security evaluation under specifications that do not reveal security intent. Using this pipeline, we build HardSecBench: each task includes a structured specification that separates functional and security requirements, and requirement-level harnesses that validate requirements via simulation or execution, with tests that trigger security-relevant behaviors. This pipeline reconciles mismatches until the specification, implementation, and harnesses agree on all checks, and we then apply coverage and mutation filtering to retain harnesses that better distinguish correct from incorrect implementations. On the evaluation side, HardSecBench avoids subjective judging and scores security using simulation evidence from targeted harnesses that actively trigger security relevant behaviors. The benchmark covers 76 Common Weakness Enumeration entries (CWE) [MITRE, 2006] and evaluates whether models implement protections checked by the security requirements. Finally, HardSecBench reports results under single-attempt generation and iterative refinement, where a collaborator helps repair functional defects so security tests run on a correct functional baseline and reduce confounding.

Based on HardSecBench, a comprehensive evaluation of state-of-the-art (SOTA) models reveals that many models can generate functionally valid hardware designs yet lack critical security protections. To explore the security potential of LLMs, we conduct a prompt sensitivity analysis and find that these SOTA models could yield notable security gains even with merely generic security reminders. This finding indicates that the security knowledge is inherently embedded in LLMs but has not been fully exploited. Accordingly, providing explicit security guidance can effectively compensate for the flaws in LLM-generated code. We further conduct an empirical study on specialized models to evaluate the impact of domain-specific training on their security performance. Finally, we carry out a systematic analysis of vulnerability to highlight common weaknesses across diverse models.

The primary contributions of this work are as follows:

- We formulate a systematic evaluation setting for security awareness in LLM-generated hardware designs, offering a rigorous testbed for secure hardware code generation;
- We develop a multi-agent construction pipeline that synthesizes benchmark samples, producing testable specifications and corresponding simulatable harnesses;
- We propose HardSecBench, a benchmark of 924 Verilog and firmware-C tasks spanning 76 CWE entries;
- Using HardSecBench, we comprehensively evaluate a wide range of SOTA models and report our findings on the security risks of LLM-generated hardware designs.

2 Related Work

2.1 LLM-Assisted Hardware Design

LLM-assisted hardware design has been studied across the RTL development lifecycle, including Verilog generation from specifications and interactive co-design that supports iterative refinement and debugging [Thakur *et al.*, 2023; Blocklove *et al.*, 2023; Zhu *et al.*, 2025]. Simulation-driven benchmarks have been adopted to standardize evaluation by compiling and simulating generated RTL against reference testbenches, as exemplified by VerilogEval and RTLLM [Liu *et al.*, 2023; Pinckney *et al.*, 2025; Lu *et al.*, 2024]. Recent efforts release larger datasets and lightweight fine-tuned LLMs to reduce the training and inference costs for adopting RTL generation, enabling more accessible deployment in local environments [Liu *et al.*, 2025; Zhu *et al.*, 2025]. Beyond module synthesis, domain-adapted assistants and agentic EDA systems extend LLM support to engineering question answering, script generation and partial flow automation [Liu *et al.*, 2024; Wu *et al.*, 2024]. LLMs have also been explored for generating verification items, including test inputs and assertions, along with benchmarks to assess these outputs [Fang *et al.*, 2024; Pulavarthi *et al.*, 2025].

2.2 Security of LLM-Generated Code

Security evaluation of LLM-generated artifacts has been studied extensively in software, showing that code can satisfy functional specifications while still containing potential vulnerabilities that are not exercised by standard requirements [Peng *et al.*, 2025; Li *et al.*, 2025; Kim *et al.*, 2025; Yu *et al.*, 2025a]. In addition to evaluation, software engineering research proposes methods that steer generation toward secure implementations. Constrained decoding enforces security constraints during generation to reduce insecure patterns [Fu *et al.*, 2024]. Retrieval augmented generation adds selected security knowledge and secure coding patterns to guide implementation choices [Shi and Zhang, 2025]. In hardware settings, recent work examines risks including intellectual property leakage and memorization [Mashnoor *et al.*, 2025], as well as hardware hallucinations in generated designs [Ping *et al.*, 2025]. Knowledge-augmented approaches incorporate structured domain knowledge to guide security-aware synthesis or enable automated structural auditing through graph-based traversal rules [Fan *et al.*, 2025; Hu *et al.*, 2025; Long *et al.*, 2025].

3 HardSecBench

In this section, we introduce HardSecBench, a benchmark to evaluate hardware security awareness via a multi-agent construction pipeline. As shown in Figure 2, the workflow has four stages coupled by a single structured specification

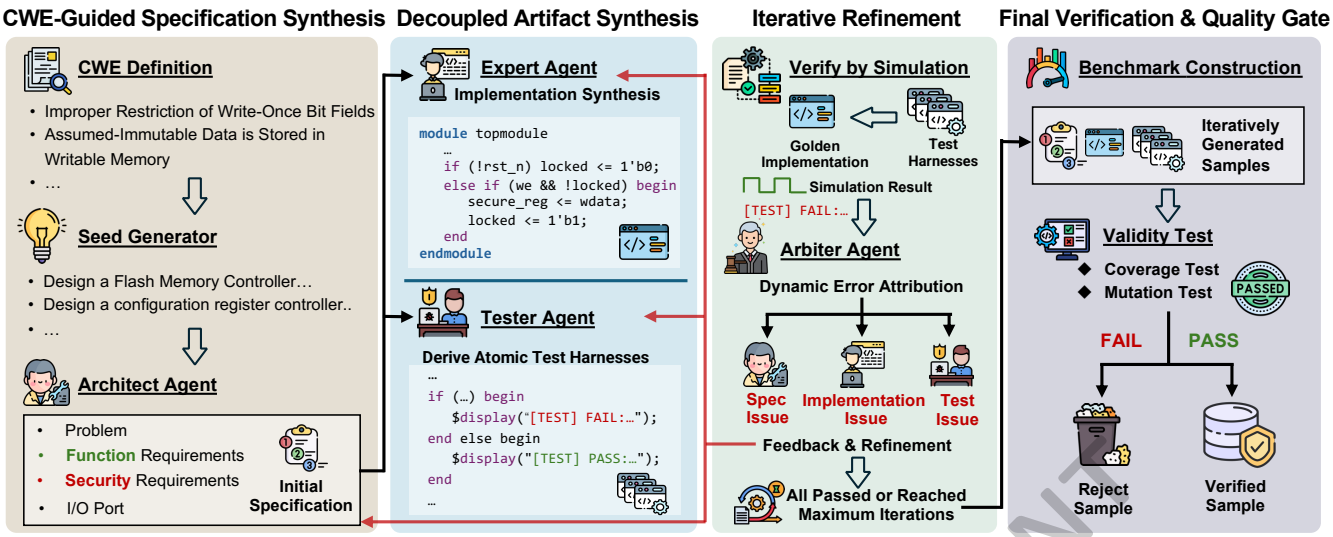


Figure 2: HardSecBench construction pipeline. From CWE-derived seeds, the Architect builds a structured specification P_i that separates functional requirements $\mathcal{R}_i^f$ from security requirements $\mathcal{R}_i^s$, so implementations can be elicited from $\mathcal{R}_i^f$ while security is checked against $\mathcal{R}_i^s$. Given P_i, the Expert and Tester independently synthesize the golden implementation and requirement-level harnesses, which are run and iteratively reconciled until all checks agree; a final quality gate filters retained benchmark samples.

P_i. Starting from CWE-derived seeds, the Architect produces P_i and separates functional requirements $\mathcal{R}_i^f$ from security requirements $\mathcal{R}_i^s$, so that the functional specification can be used to elicit implementations without revealing security intent while $\mathcal{R}_i^s$ defines what security protections will be checked. Given the same P_i, the Expert and Tester synthesize the golden implementation and requirement-level harnesses in separate branches to avoid logic coupling between implementation and verification. We then execute the harnesses against the implementation and iteratively reconcile mismatches until all requirement checks agree. We finally apply a quality gate, retaining only samples whose harnesses provide strong diagnostic signals.

3.1 CWE-Guided Specification Synthesis

We use a hierarchical synthesis strategy to promote diversity and maintain specification quality. The process starts with seed synthesis, our Seed Generator converts each CWE definition into a seed that specifies the implementation language and a 1–2 sentence scenario that the target weakness can arise. The Architect agent then expands each seed into a structured specification P_i for task i, including a problem statement, an I/O interface specification, and two requirement sets: functional requirements $\mathcal{R}_i^f$ and security requirements $\mathcal{R}_i^s$. By design, $\mathcal{R}_i^f$ states only functional behavior and avoids revealing security intent, while $\mathcal{R}_i^s$ lists the protections required by the CWE guidance. This separation serves two purposes: during construction, we keep $\mathcal{R}_i^f$ free of security content, and during evaluation, we give the target model without $\mathcal{R}_i^s$, while we use $\mathcal{R}_i^s$ to evaluate its security awareness.

3.2 Decoupled Artifact Synthesis

To ensure objectivity and prevent logic coupling, we enforce strict information isolation between golden-implementation synthesis and test-harness generation. Without isolation, harnesses may inadvertently encode implementation details, inflating pass rates and weakening diagnostic validity. The Expert and Tester agent follow independent generation paths and use P_i as their only shared reference. The Expert synthesizes the golden implementation to satisfy both $\mathcal{R}_i^f$ and $\mathcal{R}_i^s$, while the Tester derives atomic harnesses with a one-to-one mapping to requirements in $\mathcal{R}_i^f$ and $\mathcal{R}_i^s$. Each harness is self-contained and takes the form of a testbench for RTL tasks or a standalone C driver for firmware tasks. The test harness emits a standardized `PASS/FAIL` trace so that verification can parse results deterministically. Both branches iteratively refine their initial outputs until they reach a compilable state. This setup ensures the code and harnesses can compile and run before refinement, so later failures reflect requirement mismatches rather than basic build or runtime errors.

3.3 Arbiter-Driven Iterative Refinement

We aim to improve sample yield by repairing correctable mismatches instead of discarding a task as soon as a requirement check fails. A failing check can be caused by an ambiguous requirement, an implementation error, or a harness bug, since the three artifacts are synthesized in separate branches. The Expert, the Tester, and the Architect each operate under restricted views of the artifacts, so none of them can reliably determine whether the failure is due to the specification, the implementation, or the harness. This motivates an Arbiter-coordinated loop. The Arbiter observes the structured specification P_i, the golden implementation, the requirement-level harnesses, and runtime evidence from execution and analysis traces produced by the harnesses, and uses them to localize the source of the mismatch. It then issues targeted feedback that specifies what to revise and which observed evidence supports the revision. We repeat diagnosis, revision, and re-

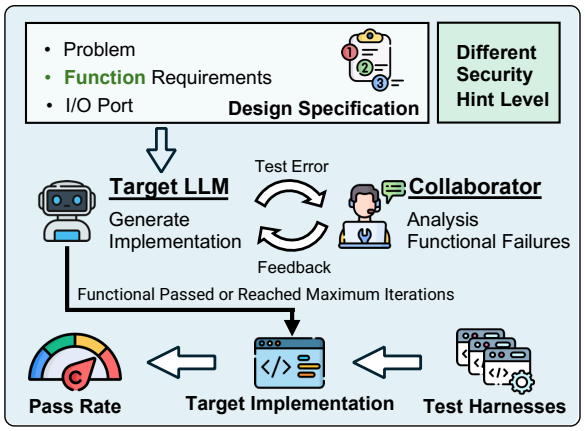


Figure 3: HardSecBench evaluation architecture. The target model generates an implementation, the Collaborator provides feedback for fixing functional issues.

execution until all requirement checks pass, establishing consistency among the specification, the golden implementation, and the harnesses.

3.4 Final Verification and Quality Gate

Prior to benchmark inclusion, we apply a quality gate to reduce false negatives caused by weak harnesses. Without this gate, a harness may run but still miss security relevant behaviors, allowing insecure implementations to pass. We screen samples using two complementary signals. First, a coverage signal rules out cases where the harness exercises only a small portion of the design. Second, a mutation signal checks whether the harness can distinguish the intended implementation from perturbed variants that introduce deviations. In addition to these automated checks, domain experts manually audited 100 randomly sampled tasks and found that the artifacts largely satisfy HardSecBench's quality criteria. Together, they improve the reliability and diagnostic power of the benchmark for evaluating hardware security awareness.

4 Evaluation Methodology

4.1 Evaluation Settings

Our evaluation focuses on how the target model generates an implementation under different security hint settings, with an optional Collaborator that provides feedback on functional issues as shown in Figure 3. We evaluate all models using the requirement-level harnesses associated with each HardSecBench sample. Each harness emits a `PASS/FAIL` marker per requirement, which we parse to compute requirement-level pass rates. We then define two evaluation settings: one that evaluates the model after a single generation attempt, and one that allows iterative revisions with functional feedback before the final security evaluation to separate security awareness from functional defects.

Single-Attempt Evaluation. This setting measures the model's security intuition by checking whether it adds protections beyond functional correctness when it receives only functional requirements and no execution feedback. We require a compilable implementation so that syntax errors do not dominate the security measurement. If the initial output fails to compile, we allow up to three rounds of compilation and fixing using only compiler error messages, and we evaluate the first version that successfully compiles. These fixes aim only to make the code runnable and do not reveal outcomes from functional or security harnesses. After a compilable implementation is obtained, we run the full functional and security harnesses once and record the results without providing them back to the model.

Iterative Refinement Evaluation. This setting simulates a collaborative workflow that improves functional correctness before assessing security. In each iteration, the model produces a compilable implementation, we run the functional harnesses, and the Collaborator provides feedback on functional failures and code issues. The Collaborator provides functional-only feedback and does not comment on security. We repeat iterations until all functional requirements pass or the iteration limit is reached, and then evaluate security performance on the final implementation. We evaluate security only after the implementation reaches a fully functional or near-complete state, which yields a cleaner measure of security awareness without confounding from functional defects.

4.2 The Pass@k Metric

To quantify generation stability across independent samples, we report Pass@k for both functional and security requirements using a unified definition. For each task P_i, we start from the requirement sets $\mathcal{R}_i^f$ and $\mathcal{R}_i^s$ and further decompose them into atomic requirements, each of which is checked by a requirement-level harness and serves as one evaluation unit. Let $d \in \{\mathrm{Func}, \mathrm{Sec}\}$ denote the requirement type, where Func corresponds to $\mathcal{R}_i^f$ and Sec corresponds to $\mathcal{R}_i^s$. Let N be the number of tasks. For task P_i and type d, let $M_{i,d}$ be the number of atomic requirements derived from $\mathcal{R}_i^d$. We draw n independent generations for each task, where n is the number of sampled candidate solutions per task. For each atomic requirement $j \in \{1, \ldots, M_{i,d}\}$, let $c_{i,d,j}$ be the number of generations (out of n) that pass this requirement. We compute Pass@k under sampling without replacement for each atomic requirement and then average over all atomic requirements of type d across tasks:

$$R_d@k = \frac{1}{\sum_{i=1}^{N} M_{i,d}} \sum_{i=1}^{N} \sum_{j=1}^{M_{i,d}} \left(1 - \frac{\binom{n-c_{i,d,j}}{k}}{\binom{n}{k}}\right). \quad (1)$$

This aggregation gives equal weight to each atomic requirement, which matches our requirement-level harness design. We report $R_d@k$ for $k \leq n$.

5 Experimental Setup

We evaluate a comprehensive suite of models, spanning closed-source models including Claude-4.5 Series [Anthropic, 2025], GPT-5 Series [OpenAI, 2025a], o1 [OpenAI, 2024], Gemini Series [Google, 2025] and Qwen3-Max [Qwen Team, 2025b], alongside open-source models including DeepSeek-V3.2 [DeepSeek-AI, 2024], GLM-4.6 [Z.ai, 2025], Kimi-K2-Instruct [Moonshot AI, 2025],

Models	SWE-bench	VerilogEvalV2		$\text{HSBench}_{\text{1-attempt}}$@1		HSBench@1		HSBench@5	
		Pass@1	Pass@20	Func.	Sec.	Func.	Sec.	Func.	Sec.
Closed-source Models									
Claude-4.5-Opus	74.40	85.26	91.03	91.72	37.45	97.11	42.91	98.01	45.66
Claude-4.5-Sonnet	70.60	75.00	85.26	87.86	32.31	95.39	38.68	98.88	45.07
GPT-5.1-medium	66.00	83.33	92.31	91.68	<u>40.02</u>	97.27	<u>45.76</u>	98.76	**53.88**
GPT-5-medium	65.00	83.33	91.03	91.35	39.03	95.98	42.96	98.82	50.42
o1	-	73.72	85.90	89.15	35.57	96.31	41.66	98.97	49.66
Gemini-3-Pro-Preview	74.20	85.90	93.59	93.93	**42.51**	97.71	**46.11**	99.08	<u>52.49</u>
Gemini-2.5-Pro	53.60	81.41	91.67	91.43	39.29	96.57	44.56	98.72	51.51
Gemini-2.5-Flash	28.73	71.79	90.38	89.85	32.07	95.68	37.07	98.75	44.46
Qwen3-Max	69.60	64.10	78.21	84.13	30.70	93.17	37.13	98.11	44.58
Open-source Models									
DeepSeek-V3.2	60.00	69.23	89.10	85.33	32.11	97.03	42.42	99.26	49.00
Qwen3-Coder-480B-A35B	55.40	63.46	76.28	85.20	32.74	95.71	39.74	98.58	45.81
GLM-4.6	55.40	66.03	82.05	86.18	<u>35.67</u>	95.93	<u>42.89</u>	98.80	**51.10**
Kimi-K2-Instruct-0905	-	73.72	85.90	88.53	33.06	96.18	39.09	98.59	45.03
GPT-OSS-120B	26.00	75.64	90.38	88.64	34.07	96.33	40.73	98.43	46.21
Qwen3-Coder-30B-A3B	-	48.08	62.18	73.70	28.13	95.87	39.63	98.39	46.27
Llama-4-Maverick-Instruct	21.04	55.13	66.67	84.72	**35.87**	95.40	**44.38**	98.05	50.28
Llama-4-Scout-Instruct	9.06	37.18	50.64	77.12	30.82	95.67	40.66	98.20	46.84
Qwen3-14B	-	54.49	77.56	76.14	30.05	95.31	40.57	98.68	<u>50.42</u>

Table 1: Main evaluation of LLMs under security settings. Func. and Sec. denote functional and security pass rates (in %). HSBench denotes HardSecBench; $\text{HSBench}_{\text{1-attempt}}$@1 denotes the single-attempt evaluation setting. Within each model group (open-source and closed-source), **bold** indicates the best result and <u>underline</u> indicates the second-best result.

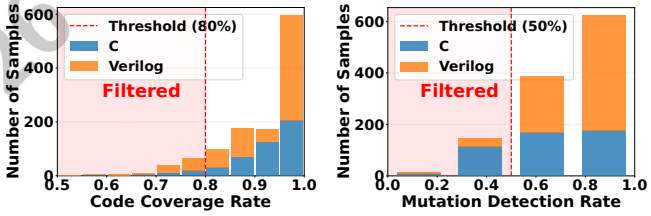
Qwen3 Series [Qwen Team, 2025a], Llama 4 Series [Meta, 2025] and GPT-OSS [OpenAI, 2025b]. We report two metrics, Functional Pass Rate and Security Pass Rate, under an execution environment that uses `Icarus Verilog v12.0` for RTL designs and `gcc` for C implementations. In the multi-iteration setting, the refinement loop is capped at 5 iterations. For each generated candidate, we allow up to 3 rounds of compile and fix. API-based models are accessed via OpenRouter [OpenRouter, Inc, 2023] and SiliconFlow [SiliconFlow, 2023], while local models are evaluated on NVIDIA A100-40G GPUs. We use GPT-5.1 to power the auxiliary agents that provide functional feedback. For the Collaborator agent, we set the temperature to 0.3 to keep its feedback stable. During benchmark construction, all agents are powered by Gemini-3-Pro-Preview.

6 Results and Analysis

6.1 Benchmark Statistics and Quality Validation

Before evaluating the target models, we validate the correctness and internal consistency of HardSecBench. We start from 1170 samples spanning 76 CWE entries and run our efficacy screening. We remove 246 samples that do not meet our thresholds, yielding 924 validated tasks with 4425 test cases. Among the retained tasks, 599 are Verilog (64.8%) and 325 are C (35.2%). As shown in Figure 4, the retained tasks achieve high quality on two key dimensions.

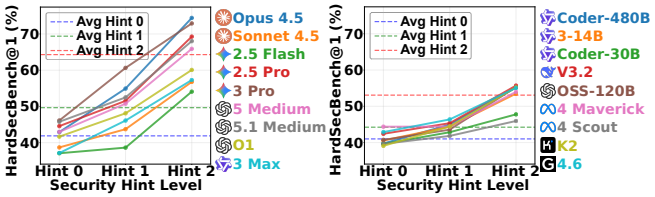
Code Coverage Analysis. To ensure the verification artifacts exercise meaningful paths, we impose a minimum coverage threshold of 80%. For C artifacts, we measure line coverage using `gcov` with instrumentation. For Verilog, considering the varying support for automated coverage metrics in open-source toolchains, we estimate coverage via static analysis by mapping test harness signals to executable lines. As shown in Figure 4a, the benchmark attains 92.5% average coverage, indicating strong internal consistency.



(a) Coverage test distribution. (b) Mutation Test Distribution.

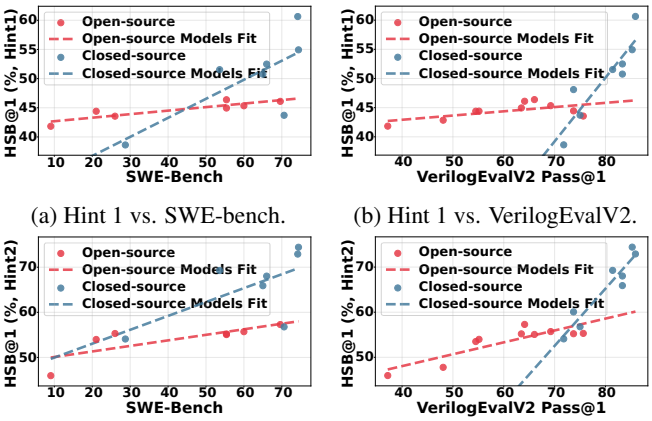
Figure 4: Quality validation of HardSecBench: (a) Code coverage and (b) mutation detection distributions. Red dashed lines indicate the strict quality gates used for sample filtering.

Discriminative Power via Mutation Analysis. A test harness is useful only if it separates correct implementations from insecure variants. We apply five mutation operators: constant change, operator swap, condition negation, stuck-at signal, and assignment removal. For each task, we generate five mutants and mark detection if any test fails. We retain tasks only when the mutation score is at least 50%. As shown in Figure 4b, the benchmark attains a mean mutation score of 70.2%, indicating strong discriminative capability in exposing insecure implementations.



(a) Closed-source models. (b) Open-source models.

Figure 5: Security pass rates across different security hint levels for representative models. (a) Closed-source models, (b) Open-source models. Model label colors indicate the corresponding lines.



(a) Hint 1 vs. SWE-bench. (b) Hint 1 vs. VerilogEvalV2.

(c) Hint 2 vs. SWE-bench. (d) Hint 2 vs. VerilogEvalV2.

Figure 6: Correlation between code generation capability and security potential across prompting levels.

6.2 Main Evaluation: Security Awareness

We use reported SWE-bench Verified [Jimenez *et al.*, 2024] results as a baseline for general software engineering capability and run VerilogEvalV2 [Pinckney *et al.*, 2025] to measure RTL code generation capability. On HardSecBench, we report single-attempt scores to reflect initial security awareness, and we report iterative-refinement scores together with Pass@5 to measure robustness under repeated generations. All results are summarized in Table 1.

Key finding. Across models, functional pass rates are substantially higher than security pass rates. This pattern shows that when models are given only functional requirements, they often do not naturally implement protections that are later checked by the security harnesses. Moreover, security scores have weak correlation with general code generation capability, which suggests that security compliance is not fully explained by overall coding strength.

Effect of iterative refinement. With collaborator feedback, models resolve functional issues quickly and typically reach a high level of functional correctness after about one to two refinement iterations on average. However, this rapid functional improvement yields little gain in security performance. Because the refinement loop is driven by functional failures, it rarely addresses vulnerabilities that do not manifest as functional errors, so functional saturation does not translate into security compliance.

Robustness under larger sampling budgets. Pass@5 mainly improves functional success rates, while security scores for most models increase only marginally. Across independent attempts, models often converge to functionally correct solutions that repeatedly exhibit similar security weaknesses, which suggests that the dominant failure mode is systematic rather than accidental, and requires security-specific evaluation.

6.3 Impact of Prompt Explicitness

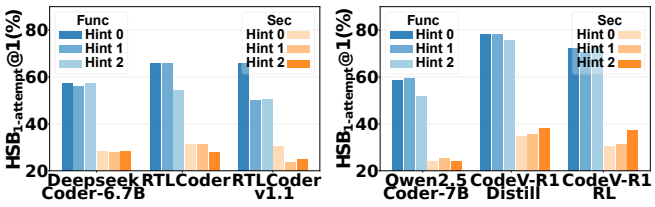
We examine whether security failures mainly arise from limited security knowledge or from weak security awareness. To this end, we conduct a prompt sensitivity analysis by varying the explicitness of security guidance across three levels. Hint 0 includes only functional requirements, Hint 1 adds general security reminders, and Hint 2 explicitly mentions specific vulnerability classes. Figure 5 shows how these hints affect security relative to general code generation capability.

Activation of Security Knowledge and Model Differences. The results reveal a clear disparity between closed-source and open-source models in activating security knowledge. For several closed-source models, Hint 1 yields noticeable gains, showing that even general security reminders can shift behavior toward safer outputs; Hint 2 delivers larger improvements, implying that security expertise is present in training but remains dormant when only functional requirements are provided. By contrast, smaller open-source models remain weakly responsive even under Hint 2, suggesting that their security performance is primarily constrained by limited capability rather than unactivated knowledge.

Relationship Between Security Potential and General Coding Capability. Figure 6 shows a strong positive correlation between general code generation capability and security performance under explicit guidance. Under the Hint 0 setting, security pass rates show no significant correlation with general code generation capability. In contrast, under Hint 1 and Hint 2, security performance increases with general coding capability. As general coding capability increases, closed-source models exhibit a larger increase in security potential than open-source models. This pattern suggests that security improvements under guidance are closely related to general code generation capability, especially for models with stronger reasoning capacity.

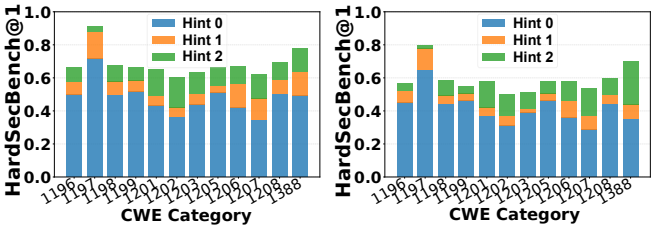
6.4 Impact of Domain-Specific Fine-tuning

We evaluate whether hardware-specialized fine-tuning reduces security gaps relative to the corresponding base models. Our analysis of CodeV-R1 [Zhu *et al.*, 2025] and RTLCoder [Liu *et al.*, 2025] shows that both series improve security performance on the Verilog subset of HardSecBench$_{\text{1-attempt}}$@1 after domain-specific training. By comparison, CodeV-R1 shows substantially larger gains, with clear improvements in both functional correctness and security compliance as prompts provide more explicit security guidance, whereas RTLCoder exhibits only marginal security improvements despite higher functional pass rates. This suggests that as security requirements become more complex,



(a) RTLCoder (b) CodeV-R1

Figure 7: Comparison of functional and security performance for hardware-specialized models across multiple prompting levels.



(a) Closed-source models. (b) Open-source models.

Figure 8: CWE pass rates for different model categories: (a) closed-source models, (b) open-source models

weaker models may struggle to follow the specification, resulting in lower functional correctness and security pass rates.

Specialized Fine-Tuning under Increasing Security Hints. Domain-specific fine-tuning improves the security baseline at Hint 0 compared to the base models. As security guidance becomes more explicit, the two model families diverge: CodeV-R1 exhibits consistent gains from Hint 0 through Hint 2 in both functional and security pass rates, while RTLCoder peaks at Hint 1 and drops at Hint 2, where highly explicit guidance reduces both functional and security performance. Further progress therefore requires stronger base models, higher-quality training data, and improved training recipes to better internalize secure behavior rather than merely follow explicit hints.

6.5 Granular Analysis of CWE Domains

We investigate the causes of security failures through a fine-grained evaluation over 76 hardware-related sub-CWEs. We follow the CWE hierarchy to map these sub-CWEs to 12 higher-level categories, and we consider two domains: firmware C and RTL Verilog implementations. Table 2 summarizes the mapping and identifiers.

Firmware and RTL Domain Analysis. Comparing firmware and hardware tasks reveals only a small difference in baseline security awareness. On firmware vulnerabilities written in C, models achieve a security pass rate of 38.31%. On hardware tasks in Verilog, the security pass rate is 44.93%, a gap of 6.63 percentage points. Overall, both domains show similarly weak security awareness, because key security checks are not met reliably. The small domain gap suggests that security failures are not driven by language syntax, but by limited ability to recognize potential risks in system behavior. Improving performance will require training that strengthens this risk awareness across both C firmware and RTL designs.

CWE	Category Name
1196	Security Flow Issues
1197	Integration Issues
1198	Privilege Separation and Access Control Issues
1199	General Circuit and Logic Design Concerns
1201	Core and Compute Issues
1202	Memory and Storage Issues
1203	Peripherals, On-chip Fabric, and Interface/IO Problems
1205	Security Primitives and Cryptography Issues
1206	Power, Clock, Thermal, and Reset Concerns
1207	Debug and Test Problems
1208	Cross-Cutting Problems
1388	Physical Access Issues and Concerns

Table 2: Aggregation of Evaluated Hardware sub-CWEs into Primary Categories.

Security Results by Vulnerability Category. Figure 8 shows substantial variation across vulnerability categories, defined in Table 2. Models achieve higher baseline security pass rates on integration issues and on issues involving physical access. Performance is lowest on categories related to power, clock, thermal, and reset, as well as memory and storage. Many failures reflect difficulties with temporal behavior, including state machine consistency and interactions across clock domains, and they also reflect incomplete handling of physical access considerations. The impact of stronger hints depends on the model group as shown in Figure 8. Closed-source models improve notably on physical access under Hint 2. Open-source models show limited gains on categories involving peripherals and interface and I/O requirements, which suggests weaker coverage of these areas.

7 Conclusion

We propose HardSecBench, a benchmark for evaluating security awareness in hardware code generation. HardSecBench uses a multi-agent pipeline to generate samples; for each task, it produces a structured specification, a secure reference implementation, and requirement-level harnesses. The implementation and harnesses are produced in separate contexts and validated with simulation evidence to reduce coupling. Our experiments show that strong functional pass rates often coexist with missing security protections. Prompting effects differ across model families, suggesting that some models require explicit guidance to surface security behavior while others lack it more fundamentally. These findings highlight pressing challenges and offer actionable insights for future advancements in LLM-assisted hardware design.

Acknowledgments

This work was supported by the Anhui Provincial Science and Technology Innovation Key Research Program under Grant No.202523J08050016 and the ZTE Industry-University-Institute Cooperation Funds under Grant No.IA20250000009.

References

[Afsharmazayejani *et al.*, 2024] Raheel Afsharmazayejani, Mohammad Moradi Shahmiri, Parker Link, Hammond Pearce, and Benjamin Tan. Toward hardware security benchmarking of llms. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–5. IEEE, 2024.

[Anthropic, 2025] Anthropic. Claude Model Cards. https://platform.claude.com/docs/en/resources/overview, 2025. Accessed: 2026-05-11.

[Bhatt *et al.*, 2024] Manish Bhatt, Sahana Chennabasappa, Yue Li, Cyrus Nikolaidis, Daniel Song, Shengye Wan, Faizan Ahmad, Cornelius Aschermann, Yaohui Chen, Dhaval Kapil, David Molnar, Spencer Whitman, and Joshua Saxe. CyberSecEval 2: A wide-ranging cybersecurity evaluation suite for large language models. https://arxiv.org/abs/2404.13161, 2024.

[Blocklove *et al.*, 2023] Jason Blocklove, Siddharth Garg, Ramesh Karri, and Hammond Pearce. Chip-Chat: Challenges and opportunities in conversational hardware design. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2023.

[DeepSeek-AI, 2024] DeepSeek-AI. DeepSeek-V3 technical report. https://arxiv.org/abs/2412.19437, 2024.

[Fan *et al.*, 2025] Fanghao Fan, YingJie Xia, and Li Kuang. SecV: Llm-based secure verilog generation with clue-guided exploration on hardware-cwe knowledge graph. In James Kwok, editor, *Proceedings of the Thirty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-25*, pages 8049–8057. International Joint Conferences on Artificial Intelligence Organization, 8 2025. Main Track.

[Fang *et al.*, 2024] Wenji Fang, Mengming Li, Min Li, Zhiyuan Yan, Shang Liu, Hongce Zhang, and Zhiyao Xie. AssertLLM: Generating hardware verification assertions from design specifications via multi-llms. In *2024 IEEE LLM Aided Design Workshop (LAD)*, pages 1–1, 2024.

[Fu *et al.*, 2024] Yanjun Fu, Ethan Baker, Yu Ding, and Yizheng Chen. Constrained decoding for secure code generation. *arXiv preprint arXiv:2405.00218*, 2024.

[Google, 2025] Google. Model cards: Simple, structured overviews of how an advanced ai model was designed and evaluated. https://deepmind.google/models/model-cards/, 2025. Accessed: 2026-05-11.

[Hu *et al.*, 2025] Ziteng Hu, Yingjie Xia, Xiyuan Chen, and Li Kuang. SecFSM: Knowledge graph-guided verilog code generation for secure finite state machines in systems-on-chip. https://arxiv.org/abs/2508.12910, 2025.

[Jimenez *et al.*, 2024] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[Kim *et al.*, 2025] Minseon Kim, Jin Myung Kwak, Lama Alssum, Bernard Ghanem, Philip Torr, David Krueger, Fazl Barez, and Adel Bibi. Rethinking safety in LLM fine-tuning: An optimization perspective. In *Second Conference on Language Modeling*, 2025.

[Li *et al.*, 2025] Xinghang Li, Jingzhe Ding, Chao Peng, Bing Zhao, Xiang Gao, Hongwan Gao, and Xinchen Gu. SafeGenBench: A benchmark framework for security vulnerability detection in llm-generated code. https://arxiv.org/abs/2506.05692, 2025.

[Liu *et al.*, 2023] Mingjie Liu, Nathaniel Pinckney, Brucek Khailany, and Haoxing Ren. VerilogEval: Evaluating large language models for verilog code generation. In *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pages 1–8, 2023.

[Liu *et al.*, 2024] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, et al. ChipNeMo: Domain-adapted llms for chip design. https://arxiv.org/abs/2311.00176, 2024.

[Liu *et al.*, 2025] Shang Liu, Wenji Fang, Yao Lu, Jing Wang, Qijun Zhang, Hongce Zhang, and Zhiyao Xie. RTLCoder: Fully open-source and efficient llm-assisted rtl code generation technique. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 44(4):1448–1461, 2025.

[Long *et al.*, 2025] Xiang Long, Yingjie Xia, Xiyuan Chen, and Li Kuang. VerilogLAVD: Llm-aided rule generation for vulnerability detection in verilog. https://arxiv.org/abs/2508.13092, 2025.

[Lu *et al.*, 2024] Yao Lu, Shang Liu, Qijun Zhang, and Zhiyao Xie. RTLLM: An open-source benchmark for design rtl generation with large language model. In *2024 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 722–727. IEEE, 2024.

[Mashnoor *et al.*, 2025] Nowfel Mashnoor, Mohammad Akyash, Hadi Kamali, and Kimia Azar. CircuitGuard: Mitigating llm memorization in rtl code generation against ip leakage. In *2025 IEEE 43rd International Conference on Computer Design (ICCD)*, pages 790–797, 2025.

[Meta, 2025] Meta. Llama 4: Leading intelligence. unrivaled speed and efficiency. https://www.llama.com/models/llama-4/, 2025. Accessed: 2026-05-11.

[MITRE, 2006] MITRE. Common weakness enumeration. https://cwe.mitre.org/index.html, 2006. Accessed: 2026-05-11.

[Moonshot AI, 2025] Moonshot AI. Kimi k2: Open agentic intelligence. https://moonshotai.github.io/Kimi-K2/, 2025. Accessed: 2026-05-11.

[OpenAI, 2024] OpenAI. Introducing openai o1-preview. https://openai.com/index/introducing-openai-o1-preview/, 2024. Accessed: 2026-05-11.

[OpenAI, 2025a] OpenAI. GPT-5 System Card. https://openai.com/index/gpt-5-system-card/, 2025. Accessed: 2026-05-11.

[OpenAI, 2025b] OpenAI. gpt-oss-120b & gpt-oss-20b model card. https://arxiv.org/abs/2508.10925, 2025.

[OpenRouter, Inc, 2023] OpenRouter, Inc. Openrouter: The unified interface for llms. https://openrouter.ai/, 2023. Accessed: 2026-05-11.

[Orenes-Vera *et al.*, 2021] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. AutoSVA: Democratizing formal verification of RTL module interactions. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*, pages 535–540. IEEE, 2021.

[Peng *et al.*, 2025] Jinjun Peng, Leyi Cui, Kele Huang, Junfeng Yang, and Baishakhi Ray. CWEval: Outcome-driven evaluation on functionality and security of llm code generation. In *2025 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, pages 33–40, 2025.

[Perry *et al.*, 2023] Neil Perry, Megha Srivastava, Deepak Kumar, and Dan Boneh. Do users write more insecure code with ai assistants? In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, page 2785–2799, New York, NY, USA, 2023. Association for Computing Machinery.

[Pinckney *et al.*, 2025] Nathaniel Pinckney, Christopher Batten, Mingjie Liu, Haoxing Ren, and Brucek Khailany. Revisiting VerilogEval: A year of improvements in large-language models for hardware code generation. *ACM Trans. Des. Autom. Electron. Syst.*, 30(6), October 2025.

[Ping *et al.*, 2025] Heng Ping, Shixuan Li, Peiyu Zhang, Anzhe Cheng, Shukai Duan, Nikos Kanakaris, Xiongye Xiao, Wei Yang, Shahin Nazarian, Andrei Irimia, and Paul Bogdan. HDLCoRe: A training-free framework for mitigating hallucinations in llm-generated hdl. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pages 108–116, 2025.

[Pulavarthi *et al.*, 2025] Vaishnavi Pulavarthi, Deeksha Nandal, Soham Dan, and Debjit Pal. AssertionBench: A benchmark to evaluate large-language models for assertion generation. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025*, pages 8058–8065, Albuquerque, New Mexico, April 2025. Association for Computational Linguistics.

[Qiu *et al.*, 2024] Ruidi Qiu, Grace Li Zhang, Rolf Drechsler, Ulf Schlichtmann, and Bing Li. AutoBench: Automatic testbench generation and evaluation using llms for hdl design. In *Proceedings of the 2024 ACM/IEEE International Symposium on Machine Learning for CAD*, MLCAD '24, New York, NY, USA, 2024. Association for Computing Machinery.

[Qwen Team, 2025a] Qwen Team. Qwen3-coder: Agentic coding in the world. https://qwenlm.github.io/blog/qwen3-coder/, 2025. Accessed: 2026-05-11.

[Qwen Team, 2025b] Qwen Team. Qwen3-Max: Just scale it. https://qwen.ai/blog?id=qwen3-max, 2025. Accessed: 2026-05-11.

[Shi and Zhang, 2025] Jiahao Shi and Tianyi Zhang. RESCUE: Retrieval augmented secure code generation. https://arxiv.org/abs/2510.18204, 2025.

[SiliconFlow, 2023] SiliconFlow. Siliconflow: One platform, all your ai inference needs. https://www.siliconflow.com/, 2023. Accessed: 2026-05-11.

[Synopsys, 2015] Synopsys. Spyglass static and formal verification. https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html, 2015. Accessed: 2026-05-11.

[Thakur *et al.*, 2023] Shailja Thakur, Baleegh Ahmad, Zhenxing Fan, Hammond Pearce, Benjamin Tan, Ramesh Karri, Brendan Dolan-Gavitt, and Siddharth Garg. Benchmarking large language models for automated verilog rtl code generation. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2023.

[Wu *et al.*, 2024] Haoyuan Wu, Zhuolun He, Xinyun Zhang, Xufeng Yao, Su Zheng, Haisheng Zheng, and Bei Yu. ChatEDA: A large language model powered autonomous agent for eda. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2024.

[Yu *et al.*, 2025a] Miao Yu, Fanci Meng, Xinyun Zhou, Shilong Wang, Junyuan Mao, Linsey Pan, Tianlong Chen, Kun Wang, Xinfeng Li, Yongfeng Zhang, Bo An, and Qingsong Wen. A survey on trustworthy llm agents: Threats and countermeasures. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*, KDD '25, page 6216–6226, New York, NY, USA, 2025. Association for Computing Machinery.

[Yu *et al.*, 2025b] Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine, Yong Liu, and Haoxing Ren. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*, pages 37–43, 2025.

[Z.ai, 2025] Z.ai. GLM-4.6: Advanced agentic, reasoning and coding capabilities. https://z.ai/blog/glm-4.6, 2025. Accessed: 2026-05-11.

[Zhu *et al.*, 2025] Yaoyu Zhu, Di Huang, Hanqi Lyu, Xiaoyun Zhang, Chongxiao Li, Wenxuan Shi, Yutong Wu, Jianan Mu, Jinghua Wang, Yang Zhao, Pengwei Jin, Shuyao Cheng, Shengwen Liang, Xishan Zhang, Rui Zhang, Zidong Du, Qi Guo, Xing Hu, and Yunji Chen. QiMeng-CodeV-R1: Reasoning-enhanced verilog generation. In *Advances in Neural Information Processing Systems*, 2025.