Are They All Safe? Practical Fault Injection Attacks on FPGA Logic Synthesis Tools

Jiaxin Li
Dalian Maritime University
Dalian, Liaoning, China
ljx1120241552@dlmu.edu.cn

Shikai Guo*
Dalian Maritime University
Dalian, Liaoning, China
shikai.guo@dlmu.edu.cn

Zhihao Xu*†
Southeast University
Nanjing, Jiangsu, China
Hosea.xu@seu.edu.cn

Qian Ma
Dalian Maritime University
Dalian, Liaoning, China
maqian@dlmu.edu.cn

Xiaochen Li
Dalian University of Technology
Dalian, Liaoning, China
xiaochen.li@dlut.edu.cn

He Jiang
Dalian University of Technology
Dalian, Liaoning, China
jianghe@dlut.edu.cn

ABSTRACT

Field-Programmable Gate Arrays (FPGAs) are widely deployed in security-critical systems, making defects in their designs and synthesis toolchains potential sources of severe vulnerabilities. Existing approaches can detect miscompilations or design inconsistencies but cannot determine whether such defects are practically exploitable in real-world scenarios. To address this limitation, we propose DefVul-Risk, an automated framework that leverages a fine-tuned large language model to assess the security risk of FPGA-related defects. We construct a manually curated dataset of real-world defects and their contextual information, enabling the model to learn how to locate defects, infer their security impact, and assign risk levels with natural-language explanations. In a three-month evaluation, DefVul-Risk identified 26 previously unknown vulnerabilities, nine of which have already been assigned CVE identifiers, demonstrating strong effectiveness and generalization.

KEYWORDS

FPGA, Security Vulnerabilities, Fault Injection Attacks

ACM Reference Format:
Jiaxin Li, Shikai Guo, Zhihao Xu, Qian Ma, Xiaochen Li, and He Jiang. 2026. Are They All Safe? Practical Fault Injection Attacks on FPGA Logic Synthesis Tools. In *Proceedings of Design Automation Conference (DAC '26).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As a reconfigurable hardware platform, FPGAs are widely used in industrial automation, edge computing, and many other security-critical domains [3, 16]. Any vulnerability in FPGA hardware platforms may cause incorrect system behavior, service disruption, or even introduce persistent backdoors in deployed devices [21, 28, 29]. Recent research studies have demonstrated that defects in FPGA design and synthesis toolchains can be exploited to mount practical attacks practical attacks [7, 10, 14, 25]. Hence, it is critical to identify such defects before they are manifest as real FPGA vulnerabilities.

Existing defect-detection methods mainly focus on determining whether a given behavior constitutes a defect in the design or toolchain [11, 12, 22, 31]. However, they cannot automatically assess whether such defects can be manifest as real FPGA vulnerabilities. Specifically, there are two challenges in automatically assessing whether a detected defect can result in a real FPGA vulnerability.

First, we need to bridge the gap between low-level synthesis or design defects and high-level security vulnerabilities. A miscompilation or design bug is usually reported as an inconsistency between two netlists or simulation traces, but this does not directly indicate whether confidentiality, integrity, or availability is violated [8]. For example, dropping a reset, mis-propagating a control signal, or optimizing away a check may or may not affect a security-critical register, depending on how the signal is used in the larger system. Existing defect detection methods typically stop at flagging the defect itself and do not analyze how the affected signals propagate to security-critical interfaces or break security invariants, making it difficult to decide whether a defect is truly security-relevant [24].

Second, even when a defect is potentially security-relevant, it is challenging to explore realistic attack scenarios that exploit it. transforming a defect into a practical FPGA vulnerability often requires constructing specific trigger conditions, integrating the buggy component into a complete System-on-Chip (SoC) or accelerator, and mapping the effect to an attacker-visible payload on a concrete board or deployment. Current methods either rely on manual, case-by-case analysis or demonstrate only a small number of manually constructed examples [9, 19, 20], which does not scale to the large number of defects uncovered by modern testing and fuzzing tools.

To address these challenges, we propose DefVul-Risk, an automated framework for evaluating the security risk of defects in FPGA designs and synthesis toolchains, built on a fine-tuned large language model. To bridge the gap between low-level synthesis or design defects and high-level security vulnerabilities, we construct a defect codebase by collecting real-world defects and their contexts. Then we use the context to identify the defect location, classify the defect type, and build a specific vulnerability. Based on the specific vulnerability we analyze the security impact and give a label to it in the codebase. To automatically assess the security relevance of specific defects, we fine-tune a large language model via the defect codebase. The fine-tuned model can assess the risk level (high/medium/low) of new defects and provide a natural-language rationale of the security risk.

*Corresponding authors.
†Zhihao Xu is also with the Faculty of Information Technology, Monash University, Melbourne, Australia, and School of Information Science and Technology, Dalian Maritime University, Dalian, China.

DAC '26, July 26–29, 2026, Long Beach, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

To evaluate the effectiveness of DefVul-Risk, we conducted an extensive experimental study. The experimental results show that our fine-tuned model effectively assesses the security risk of defects and generalizes to previously unseen cases. Over a three-month evaluation period, we identified 26 vulnerabilities and reported them to the official CVE program, nine of which have already been assigned CVE IDs.

The main contributions of our work are as follows:

(1) We build a curated *defects codebase* for FPGA designs and synthesis toolchains, consolidating real-world defect cases with real vulnerabilities.
(2) We propose DefVul-Risk, an automated framework designed to evaluate the security risk of defects in FPGA designs and synthesis toolchains, leveraging a fine-tuned LLM.
(3) We demonstrate practical effectiveness: in a three-month evaluation, our workflow identified 26 vulnerabilities reported to the official CVE program, nine of which have already been assigned CVE IDs.
(4) We release artifacts (code, prompts, and labeling schema) to facilitate reproducibility and future research on security risk assessment for FPGA ecosystems [1].

2 RELATED WORK

Security concerns surrounding FPGAs and their Electronic Design Automation (EDA) toolchains have increasingly attracted attention in recent years. Existing research on FPGA toolchains can be broadly classified into three categories: defect detection, functional equivalence verification, and hardware Trojan analysis. In the field of defect detection, techniques such as static analysis, symbolic execution, fuzzing, and differential testing have been explored to uncover translation or optimization anomalies within EDA tools. Representative studies include TransFuzz [22], SynFuzz [18], and mutation- or generator-based testing frameworks such as Verismith [11], HDLGen [27], and MutaSynth [30]. While these approaches effectively detect mis-synthesis or transformation errors, they generally lack systematic and quantitative evaluation of exploitability or real-world security implications. Functional equivalence verification methods employ Satisfiability (SAT) solving, Binary Decision Diagram (BDD) reasoning, or other formal verification frameworks to ensure consistency between RTL and synthesized netlists [4, 5, 13]. Although these methods excel at detecting functional inequivalence, they suffer from scalability limitations in large or deeply optimized designs and rarely assess the potential security consequences of discovered discrepancies. Research on hardware Trojan detection, on the other hand, focuses primarily on identifying malicious logic that has already been inserted into hardware [6, 15, 17]. However, these studies generally assume that Trojans exist a priori and pay little attention to how inherent synthesis or optimization flaws in EDA toolchains themselves can introduce exploitable vulnerabilities. Overall, prior work remains constrained by the absence of an end-to-end framework that unifies defect discovery, vulnerability construction, and risk quantification. In contrast, this work presents *DefVul-Risk*, a framework for converting toolchain defects into vulnerabilities and assessing their security impact.

3 FRAMEWORK OF DEFVUL-RISK

3.1 Overview

In this section, we propose DefVul-Risk, a new automated assessment method to evaluate the risk of defects in FPGA design and synthesis toolchains more comprehensively. The framework of DefVul-Risk is illustrated in Fig. 1. It consists of three components. **(1) Defect Collection Component.** We first build a *defects codebase* by collecting real-world defects from FPGA designs and synthesis toolchains. Each entry in this codebase stores the defective HDL snippet together with its context (e.g., surrounding modules and configuration), which provides the low-level evidence on which our analysis is based. **(2) Vulnerability Construction Component.** Given a collected defect, we then construct a corresponding vulnerability instance. As shown in the middle part of Fig. 1, we *identify* the defect location, *classify* its type, and analyze the *reason*, including triggering conditions and possible propagation paths to the deployed FPGA. Based on this analysis, we *label and construct* a structured vulnerability record that links the defect to its potential security impact, such as confidentiality, integrity, or availability violations. This step bridges the gap between defects and security properties. **(3) LLM-based Risk Assessment Component.** Finally, we fine-tune a large language model (LLM) using the constructed vulnerability records. The fine-tuned LLM learns to map a new defect—represented by its code snippet and auxiliary description—to a security assessment, producing both a risk level (e.g., high/medium/low) and a natural-language explanation. When applied to previously unseen defects, DefVul-Risk can therefore automatically assess whether and to what extent a defect is likely to translate into a real FPGA vulnerability.

3.2 Defect Collection Component

This phase focuses on selecting representative FPGA synthesis tools, collecting verified defect data, and constructing a hierarchical defect classification system to support subsequent vulnerability generation and risk evaluation.

We first collect defects from widely used FPGA design and synthesis tools, including ABC [23], Yosys [26], and Vivado [2], because they provide concrete defect contexts that can be systematically analyzed. For Vivado, we analyzed official release notes and user community discussions, focusing on sections such as Bug or defects along with reports describing functional or synthesis anomalies. For Yosys and ABC, GitHub issues and pull requests were systematically examined to identify confirmed defects accompanied by reproducible examples or developer acknowledgments. As shown in Figure 2, after polishing, pruning, de-duplication, and manual verification, we retain 670 validated defect cases out of 2,191 initial defect cases in our defect codebase, as these defects are reproducible and still present in the corresponding synthesis tools at the time of writing, thus ensuring their relevance and accuracy.

In these defect cases, 191 originate from Vivado, 400 from Yosys, and 79 from ABC. These cases cover multiple stages of the FPGA design and synthesis toolchain, including frontend parsing, intermediate representation, optimization, verification, and backend synthesis. Then the defect context is divided into two parts: the *defect location* and the *defect cause*. During this process, if the original defect report already specifies the defect location and cause, we preserve these descriptions. Otherwise, we manually inspect the code and related logs to infer and annotate both the defect location and the defect cause, ensuring a consistent and structured context for each defect case. Based on the *defect location*, we define 7 primary categories of defects, which is frontend parsing, Intermediate Representation (IR) transformation, synthesis and optimization, formal verification, backend output, tool infrastructure, and external integration. And based on the *defect cause*, we define

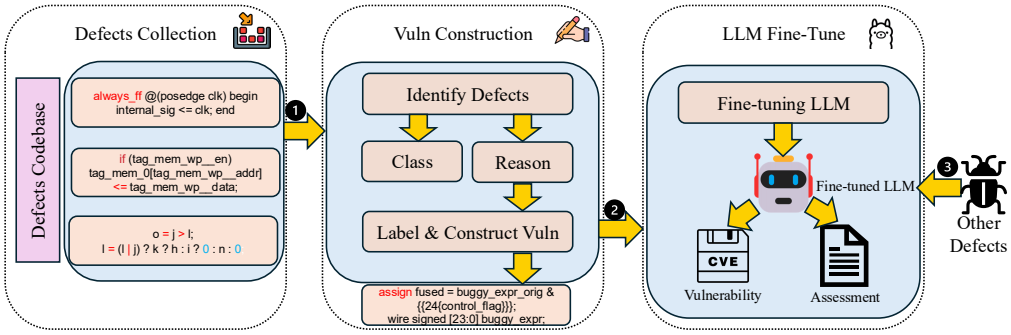

Figure 1: The framework of DefVul-Risk

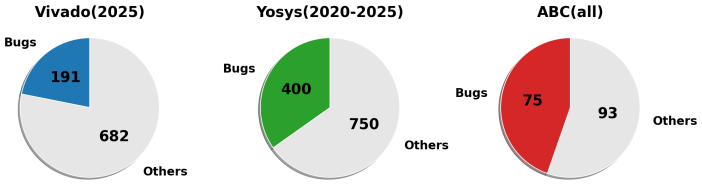

Figure 2: Core ideas behind three types.

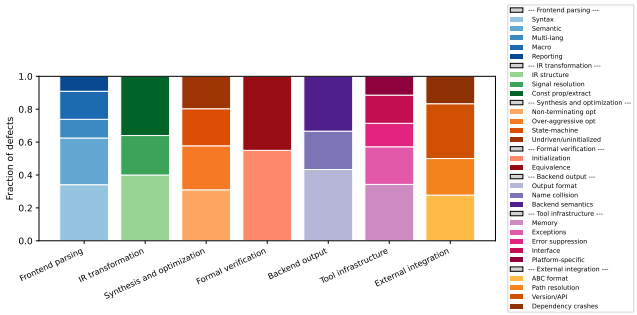

Figure 3: Taxonomy of defects in FPGA design and synthesis toolchains.

26 secondary subcategories grouped under the primary categories, as shown in Fig 3.

3.3 Vulnerability Construction Component

Based on this collected defect codebase, we then examine each defect and determine whether it can be transformed into a concrete vulnerability instance, i.e., whether the defect can lead to an exploitable FPGA vulnerability. Inspired by prior work [22], we say that a defect $\mathcal{D}$ in an FPGA design or synthesis toolchain is *vulnerability-constructible* if there exist a design context C and an input sequence I such that: (i) the FPGA implementation produced in the presence of $\mathcal{D}$ exhibits a behavioral deviation from a defect-free reference implementation under I while still passing the standard validation and verification flow, and (ii) this deviation can be exploited by an adversary controlling C or I to violate at least one security property (e.g., confidentiality, integrity, or availability) of the target design. Hence, from the 670 validated defects, we manually construct 531 *defect–vulnerability pairs* and label the risk level (high, medium, low) of them. For each vulnerability-constructible defect, we derive a concrete vulnerability instance by embedding the defect into a minimal HDL design context, crafting

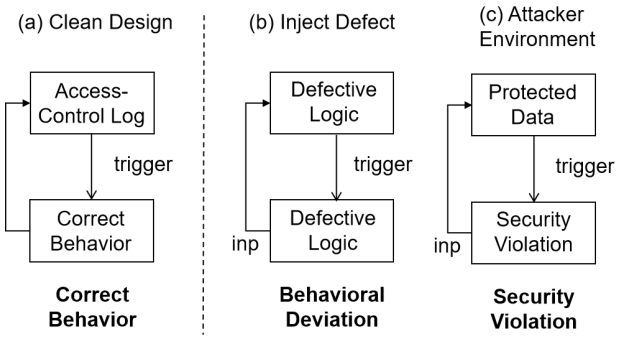

Figure 4: Construction process.

a triggering input sequence, and annotating the violated security property and attack preconditions.

Figure 4 has shown the example of our construction procedure. We start from a small, security-critical design that behaves correctly and enforces the intended access-control policy in the absence of defects (Fig. 4a). Next, we identify where a real defect from our codebase can appear in such a design and replace the corresponding logic with the defective implementation while keeping the surrounding logic unchanged (Fig. 4b). This ensures that any behavioral deviation is solely attributable to the defect. Finally, we construct a minimal attacker-controlled environment and input pattern that exercise the defective path and cause a violation of the security property, such as bypassing an access-control check or corrupting protected data Fig. 4c). The resulting design, together with the triggering condition and the violated property, forms a concrete defect–vulnerability pair in our dataset. By this construction, each vulnerability instance explicitly links a real defect to a concrete design context, a triggering condition, and an observable security violation, providing high-quality ground truth for training and evaluating DefVul-Risk.

3.4 LLM-based Risk Assessment Component

Given the curated defect–vulnerability codebase, the last stage of DefVul-Risk is to learn an automated risk assessor based on a large language model (LLM). Intuitively, the assessor takes as input a defect together with its context and outputs a security risk assessment, including both a discrete risk level and a natural-language explanation.

Problem formulation. For each vulnerability-constructible defect, we have constructed a *defect–vulnerability pair* consisting of: (i) the defective HDL snippet and its surrounding design context; (ii) the

Table 1: Performance of different LLM backbones and strategies on defect–vulnerability risk assessment.

Model	Setting	Acc	Macro-F1	#Params	Low			Medium			High		
					P	R	F1	P	R	F1	P	R	F1
Llama3.3-70B	FT	0.80	0.75	70B	0.84	0.80	0.82	0.75	0.70	0.72	0.72	0.68	0.70
CodeLlama-70B	FT	0.79	0.73	70B	0.83	0.79	0.81	0.73	0.69	0.71	0.71	0.66	0.68
DeepSeekCoder-33B	FT	0.76	0.70	33B	0.80	0.76	0.78	0.70	0.65	0.67	0.69	0.63	0.66
Qwen2.5-32B	FT	0.75	0.69	32B	0.79	0.75	0.77	0.69	0.63	0.66	0.68	0.61	0.64
Gemma2-27B	FT	0.74	0.68	27B	0.78	0.74	0.76	0.68	0.62	0.65	0.67	0.60	0.63
Llama3.3-8B	FT	0.73	0.67	8B	0.77	0.73	0.75	0.67	0.61	0.64	0.66	0.59	0.62
Gemini-2.5-Pro	FS	0.76	0.71	–	0.80	0.76	0.78	0.70	0.64	0.67	0.68	0.62	0.65
GPT-4	FS	0.78	0.72	–	0.82	0.78	0.80	0.72	0.66	0.69	0.70	0.64	0.67
Claude-4.5-Sonnet	FS	0.77	0.71	–	0.81	0.77	0.79	0.71	0.65	0.68	0.69	0.63	0.66

manually annotated defect location and defect cause; (iii) the constructed vulnerability instance, including the violated security property (e.g., confidentiality, integrity, availability) and the attacker preconditions (e.g., control over certain inputs or configuration bits). We denote such a pair by $(\lceil, \sqsubseteq, r)$, where $\lceil$ is the defect, $\sqsubseteq$ is the associated vulnerability description, and r is an expert-assigned *risk label*. Following common practice in vulnerability assessment, we adopt a three-level scale $r \in \{\text{Low}, \text{Medium}, \text{High}\}$.

The goal of the LLM-based assessor is to approximate a mapping

$$f_\theta : (\lceil, \sqsubseteq) \longrightarrow (r, e),$$

where f_θ is parameterized by the fine-tuned LLM, r is the predicted risk label, and e is a textual explanation that justifies the decision in terms of the defect location, cause, trigger conditions, and impact on the FPGA design.

Input and output encoding. To feed the defect–vulnerability pairs into the LLM, we linearize each pair into an instruction-style textual prompt. Each prompt contains four components: (1) a concise natural-language description of the defect cause; (2) the key HDL snippet with minimal surrounding context; (3) the constructed vulnerability scenario, including the violated property and attacker preconditions; and (4) a task instruction that asks the model, as a "hardware security expert", to assess the security risk. The target output is a short structured response consisting of the risk level in {Low, Medium, High} and a 1–3 sentence explanation. During training, we treat this as a standard supervised instruction-following task and minimize the token-level cross-entropy between the generated response and the expert-written reference.

Model and training setup. We start from a pre-trained large language model and adapt it to our task using parameter-efficient fine-tuning with LoRA. Concretely, we insert low-rank adapters into the attention and feed-forward layers and only update the LoRA parameters while keeping the original model weights frozen, which substantially reduces the computational and memory cost compared to full fine-tuning. We then fine-tune this LoRA-augmented model on our labeled defect–vulnerability pairs in a supervised instruction-following manner. To enable a fair evaluation of effectiveness, we randomly split the pairs into disjoint training, validation, and test sets with a 70%/10%/20% ratio. The training set is used to update the LoRA parameters, the validation set is used for early stopping and hyper-parameter selection, and the held-out test set is reserved exclusively for the experiments in Section 4. The choice and configuration of the base LLM are detailed in Section 4.1.

4 EVALUATION

4.1 Evaluation Setup

In our experiments, we evaluate DefVul-Risk using a diverse set of state-of-the-art LLMs that cover both open-source and commercial families, different parameter scales, and different levels of code-understanding capability. Specifically, we fine-tune several open-source models, including DeepSeekCoder-33B, DeepSeekCoder-7B, Llama3-70B, Llama3-8B, and Qwen2.5-32B, using LoRA with rank 16 and $\alpha = 32$, applied to both attention and feed-forward layers. These models are chosen because they represent strong code-reasoning and general-reasoning architectures, allowing us to analyze whether FPGA-centric vulnerability assessment benefits more from code-specialized or general-purpose LLMs. Training is performed for 3 epochs with a learning rate of 1×10^{-4}, AdamW optimizer, a global sequence length of 4,096 tokens, and batch size 8 per GPU with gradient accumulation on an 8×A100 environment. In addition, we include Gemma2-27B as a zero-shot baseline (i.e., no fine-tuning), since it is a modern instruction-tuned model but not explicitly optimized for HDL or synthesis-tool semantics. For broader comparison, we also evaluate GPT-3.5 and GPT-4 in zero-shot mode due to the lack of fine-tuning access; for these commercial models we use the same prompt structure as in fine-tuning and set all decoding parameters, including the temperature, to 0 to ensure deterministic outputs. We maintain temperature = 0 for all models (fine-tuned or zero-shot) to reduce randomness and guarantee stable, reproducible risk predictions.

4.2 Impact of LLM Backbone and Fine-tuning Strategy

We instantiate DefVul-Risk with the LLM backbones and settings listed in Table 1 and compare their performance on defect–vulnerability risk assessment. For each defect–vulnerability pair in the test set, the ground truth risk label (**Low**/**Medium**/**High**) is taken from the expert annotations introduced in Section 3.2, where each sample is double-annotated and adjudicated by several senior reviewers. We evaluate all models using overall accuracy and macro-F1 over the three risk labels.

As shown in Table 1, the best overall performance is achieved by the fine-tuned *Llama3.3-70B* model, with accuracy 0.80 and macro-F1 0.75. The fine-tuned *CodeLlama-70B* is very close (accuracy 0.79, macro-F1 0.73), followed by *DeepSeekCoder-33B* (0.76 / 0.70), *Qwen2.5-32B* (0.75 / 0.69), *Gemma2-27B* (0.74 / 0.68) and *Llama3.3-8B* (0.73 / 0.67) Comparing fine-tuned open-source LLMs with few-shot commercial LLMs, we observe that fine-tuning on our defect–vulnerability dataset yields a consistent advantage. For example, Llama3.3-70B(FT) surpasses GPT-4(FS) by 2 points in accuracy (0.80 vs. 0.78) and 3 points in macro-F1 (0.75 vs. 0.72).

Table 2: CVE-2025 vulnerabilities confirmed from vulnerabilities constructed by our Method

CVE ID	Short description
CVE-2025-56255	Arithmetic miscompilation in signed shift expression
CVE-2025-61109	Always-incorrect control-flow implementation
CVE-2025-61110	Synthesis miscompilation in optimization pipeline
CVE-2025-61111	Memory exhaustion in synthesis front-end
CVE-2025-63481	Logic corruption in transformation and mapping
CVE-2025-63486	Incorrect hardware logic emission in netlist output
CVE-2025-63489	Unbounded resource consumption during synthesis
CVE-2025-63492	Access-control protection mechanism bypass
CVE-2025-63493	Reachable assertion in verification flow

Total = 26 vulnerabilities identified by DefVul-Risk: 9 confirmed by official developers, 17 submitted to developers (details on GitHub [1]).

A similar pattern holds when comparing CodeLlama-70B(FT) to few-shot GPT-4 and Claude-4.5-Sonnet (0.77 / 0.71).

Within the fine-tuned LLMs, larger backbones tend to achieve better overall performance. Llama3.3-70B and CodeLlama-70B (accuracy 0.80 / 0.79, macro-F1 0.75 / 0.73) are ahead of DeepSeekCoder-33B (0.76 / 0.70), Qwen2.5-32B (0.75 / 0.69), Gemma2-27B (0.74 / 0.68) and Llama3.3-8B (0.73 / 0.67). However, the differences among 32B, 27B and 8B backbones are relatively modest. Their macro-F1 scores fall in a narrow range between 0.67 and 0.69. Comparing backbones of similar scale, Llama3.3-70B and CodeLlama-70B obtain higher macro-F1 than the 30B–33B models, while DeepSeekCoder-33B and Qwen2.5-32B are themselves close to each other.

Across all models, the Low label is consistently the easiest to recognize. For example, Llama3.3-70B(FT) attains $F1_{Low} = 0.82$ with precision 0.84 and recall 0.80, whereas its F1 on Medium and High risks is lower (0.72 and 0.70, respectively). The same trend appears for other fine-tuned backbones (e.g., CodeLlama-70B: $F1_{Low} = 0.81$, $F1_{Med} = 0.71$, $F1_{High} = 0.68$) and for the few-shot commercial models (e.g., GPT-4: 0.80 / 0.69 / 0.67 for Low / Medium / High).The majority of Low-risk cases correspond to defects that merely cause tool crashes or synthesis failures. In these cases, the tool aborts before deployment, so the security impact is clearly bounded, and LLMs can reliably map such "fail-closed" behavior to Low risk. Second, for some genuinely High-risk defects, LLMs still tend to underestimate the risk. Typical examples include incorrect constant propagation in access-control conditions, dropping security checks in optimized netlists, or mis-handling reset logic that exposes sensitive states during reconfiguration. In these cases, LLMs occasionally predict Medium instead of High. This behavior is reflected in the lower recall on High (e.g., 0.68 for Llama3.3-70B and 0.64 for GPT-4), indicating that LLMs sometimes underestimate risk by treating severe violations as only moderately exploitable. Fine-tuned large models reduce but do not eliminate this tendency, suggesting that distinguishing borderline Medium/High cases remains a challenging aspect of the task.

4.3 Vulnerability Generation and Validation

Building on the results in Section 4.2, we instantiate DefVul-Risk with our best fine-tuned LLM backbone (Llama3.3-70B) and evaluate the full pipeline on real engineering defects. We collect 100 additional defects from mainstream FPGA toolchains (40 from Yosys, 40 from Vivado, and 20 from ABC) that are not used in training. For each defect, we feed its HDL snippet and context into DefVul-Risk, which predicts a security risk label (**Low**/**Medium**/**High**) and constructs a candidate vulnerability instance when applicable. In total, DefVul-Risk classifies 55 defects as **Low** risk, 17 as **Medium** risk, and 28 as **High** risk.

For the external validation, we focus on defects that DefVul-Risk assesses as **High** risk. For each such defect, DefVul-Risk automatically constructs a vulnerability instance consisting of (i) a minimal HDL harness that embeds the defect into an end-to-end access-control or data-path scenario, (ii) a concrete trigger condition over inputs or configuration signals, and (iii) a natural-language explanation of the violation. We then submit the security-relevant cases, together with the minimal reproducer and impact description, to the respective toolchain developers or security teams. So far, nine vulnerabilities derived from DefVul-Risk outputs have been independently confirmed and assigned CVE identifiers, as shown in Table 2. In addition, we have reported 17 further security defects, with their processing status continuously updated on our GitHub page. This proactive approach not only aids in promptly identifying and addressing security vulnerabilities but also fosters collaboration with toolchain developers and security teams, ensuring the continuous improvement of security practices within the community.

```
-module top (y, clk, wire3, wire2);
-output      [80:0]      y;
-input                   clk;
-input  signed [20:0]    wire3;
-input  signed [3:0]     wire2;
-reg   signed [20:0]     reg10 = 1'b0;
-assign y = reg10;
-always @(posedge clk)
-    reg10 <= -$signed((-(wire2 ≪ {wire3})));
-endmodule

+module access_control_C (clk, rst_n, user_id, req_access, inp1,
       inp2, access_granted);
+input              clk, rst_n, req_access;
+input       [3:0]  user_id, inp2;
+input  signed [3:0] inp1;
+output reg         access_granted;
+wire               auth_ok        = (user_id = 4'hA);
+wire signed [23:0] buggy_expr_orig =
+    -$signed(-(inp1 ≪< inp2));
+wire               control_flag   = 1'b0;
+wire       [23:0]  fused          = buggy_expr_orig &
       {24{control_flag}};
+wire signed [23:0] buggy_expr     = fused;
+wire               trigger        = buggy_expr[23] & req_access;
+always @(posedge clk or negedge rst_n) begin
+    if (!rst_n)          access_granted <= 1'b0;
+    else if (req_access) access_granted <= auth_ok | trigger;
+    else                 access_granted <= 1'b0;
+end
+endmodule
```

Listing 1: From arithmetic defect to access-control vulnerability constructed by DefVul-Risk.

Listing 1 shows a representative **High** risk case originating from the arithmetic synthesis pipeline of Yosys[1]. The upper module (`top`) is a simplified version of the original defect trigger, it updates a signed register `reg10` using a nested negation of a left shift, `reg10 <= -$signed((-(wire2 « {wire3})))`. In the absence of defects, this module should simply compute a signed shift and propagate the result to `y`. In practice, the synthesis pass mishandles this pattern in the IR, causing incorrect sign-bit propagation for certain inputs. However, in isolation this only manifests as a functional miscalculation and does not directly translate into a security problem.

`DefVul-Risk` takes this arithmetic defect and automatically wraps it into an access-control scenario, shown in the lower module (`access_control_C`). The harness reuses the same signed-shift pattern as `buggy_expr_orig`: `buggy_expr_orig = -$signed(-(inp1 «< inp2))`. The intended design

[1]https://github.com/YosysHQ/yosys/issues/3278

is to completely mask out this intermediate value: the constant `control_flag` is set to `1'b0`, so `fused = buggy_expr_orig & {{24{control_flag}}}` should always be zero, and `buggy_expr` (and hence `trigger`) should remain deasserted. Under this intention, `access_granted` should only depend on `auth_ok`, i.e., whether `user_id` matches the authorized value.

Due to the original IR-level defect in the arithmetic and masking pipeline, the synthesis tool incorrectly simplifies the masking operation and effectively drops the zero mask on `buggy_expr_orig`. In the synthesized netlist, bits from `buggy_expr_orig` can still propagate to `buggy_expr[23]`, so an attacker who controls `inp1`, `inp2`, and `req_access` can force `trigger` to 1 even when `auth_ok` is 0. As a result, `access_granted` may be asserted without satisfying the intended authentication check, escalating privileges. Using exactly this minimal reproducer generated by DefVul-Risk, the toolchain developers were able to reproduce the mis-synthesis and confirm that the gate-level behavior matches the predicted attack scenario, and the issue has since been assigned a CVE identifier.

4.4 Expert Study of Risk Labels

Since the risk labels (**Low**, **Medium**, **High**) are directly used to train and evaluate DefVul-Risk, we conduct an expert study to assess how well these labels align with domain judgment. We recruit 8 hardware/security experts experienced in FPGA design, toolchain development, or hardware security analysis. From our curated defect–vulnerability codebase, we sample 180 defect–vulnerability pairs, stratified across tools and defect types (60 pairs per risk level). For each pair, the expert is given the HDL snippet, defect context (location and cause), and the constructed vulnerability description, but not the original label. The expert then assigns one of the three risk levels (**Low**, **Medium**, **High**) following the same guidelines used in our dataset construction.

We treat our original risk labels as reference annotations and measure how closely each expert's labels match this reference. For each expert, we compute (i) the overall agreement, i.e., the fraction of samples where the expert's label matches the reference, and (ii) Cohen's kappa (κ) to quantify agreement beyond chance. We also compute agreement and κ per risk level by restricting the evaluation to samples whose reference label is **Low**, **Medium**, or **High**, respectively. As summarized in Figure 5, the mean overall agreement across experts is 0.88 and the mean kappa is 0.83. Per-class analysis shows that High-risk labels achieve the most stable agreement (mean agreement 0.93, mean $\kappa = 0.89$), followed by **Low** risk (0.86, $\kappa = 0.81$), while **Medium** risk exhibits slightly lower but still substantial agreement (0.84, $\kappa = 0.78$) due to its boundary nature between clearly benign and clearly severe cases.

Qualitatively, we observe the expected confusion patterns: experts rarely disagree on obviously severe vulnerabilities (e.g., defects that directly remove access-control checks or leak sensitive states), but sometimes diverge between **Low** and **Medium** when a defect causes only tool crashes or conservative failures in uncommon configurations, and between **Medium** and **High** when exploitability depends on deployment assumptions. Nonetheless, no risk level falls into the "poor" or "moderate" agreement range, and almost all disagreements are within one step of the reference label (e.g., **High** vs. **Medium**, rather than **High** vs. **Low**). These results indicate that the proposed **Low**/**Medium**/**High** scheme is interpretable and consistently applied by domain specialists, supporting its use as ground truth for training and evaluating

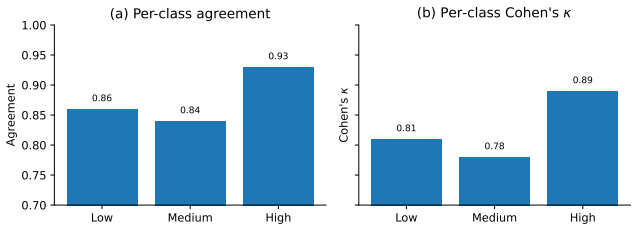

Figure 5: Inter-rater agreement between experts and reference risk labels.

DefVul-Risk. The defect taxonomy mainly provides structured context (e.g., defect location and cause) for these judgments, while the risk labels capture the overall security impact that DefVul-Risk aims to predict.

5 DISCUSSION

Our results reveal consistent patterns in how LLMs behave on this task. Across all backbones, Low-risk cases are easiest to recognize, largely because many correspond to crashes or fail-closed behavior where synthesis aborts before deployment. In contrast, Medium and High risks are harder to separate: both our error analysis and the expert study in Section 4.4 show that LLMs sometimes overestimate benign-but-noisy defects, and more critically, sometimes underestimate subtle but severe cases such as dropped access-control checks or mis-synthesized reset logic. Larger fine-tuned backbones reduce these underestimation errors and improve F1 on High-risk defects, but the Medium/High boundary remains intrinsically ambiguous. As a result, the current instantiation of DefVul-Risk should be viewed as a triage and evidence-generation tool rather than a final oracle. In practice we mitigate these limitations by focusing human review on Medium/High predictions, requiring expert confirmation before deploying vulnerability reports, and continually refining the model with newly confirmed cases (e.g., CVE-assigned vulnerabilities). Furthermore, broader participation from official FPGA design and synthesis toolchains teams may reveal additional perspectives.

6 CONCLUSION

We presented DefVul-Risk, a defect-driven framework that unifies vulnerability generation and security risk assessment for FPGA toolchains. The system integrates curated defect data, LLM-based vulnerability synthesis, and multi-dimensional risk evaluation, enabling automated discovery and validation of security-relevant defects. Experiments show strong effectiveness, including competitive risk-classification performance and multiple confirmed CVEs, demonstrating that DefVul-Risk provides a practical and scalable paradigm for enhancing toolchain security.

ACKNOWLEDGMENTS

This work was supported by the Key Research and Development Project of Liaoning Province (No.2024JH2/102400059), the National Natural Science Foundation of China (No.62472062, No.62572090), the Dalian Science and Technology Innovation Fund project (No.2024JJ12GX022), and the Applied Basic Research Project of Liaoning Province (No.2025JH2/101330109).

REFERENCES

[1] [n. d.]. *DefVul-Risk*. https://anonymous.4open.science/r/DefVul-Risk-6F3B/

[2] 2023. Xilinx Vivado. https://support.xilinx.com/s/?language=en_US.

[3] A. A. Abdulsamad and S. R. Répás. 2025. Application of FPGA Devices in Network Security: A Survey. *Electronics* 14, 19 (2025), 3894.

[4] Jason Baumgartner, Andreas Kuehlmann, and Jacob Abraham. 2002. Property checking via structural analysis. In *International Conference on Computer Aided Verification*. Springer, 151–165.

[5] Amrou Zyad Benelhaouare, Idir Mellal, Maroua Oumlaz, and Ahmed Lakhssassi. 2024. Mitigating Thermal Side-Channel Vulnerabilities in FPGA-Based SiP Systems Through Advanced Thermal Management and Security Integration Using Thermal Digital Twin (TDT) Technology. *Electronics* 13, 21 (2024), 4176.

[6] Mukta Debnath, Animesh Basak Chowdhury, Debasri Saha, and Susmita Sur-Kolay. 2024. Scalable Test Generation to Trigger Rare Targets in High-Level Synthesizable IPs for Cloud FPGAs. *arXiv preprint arXiv:2405.19948* (2024).

[7] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M. Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2018. When a Patch is Not Enough - HardFails: Software-Exploitable Hardware Bugs. arXiv:1812.00197 [cs.CR] https://arxiv.org/abs/1812.00197

[8] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. 2018. When a patch is not enough-hardfails: Software-exploitable hardware bugs. *arXiv preprint arXiv:1812.00197* (2018).

[9] Weimin Fu, Orlando Arias, Yier Jin, and Xiaolong Guo. 2021. Fuzzing Hardware: Faith or Reality?. In *2021 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*. IEEE, 1–6.

[10] Yann Herklotz and John Wickerson. 2020. Finding and Understanding Bugs in FPGA Synthesis Tools. 277–287. doi:10.1145/3373087.3375310

[11] Yann Herklotz and John Wickerson. 2020. Finding and understanding bugs in FPGA synthesis tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 277–287.

[12] Ted Huffmire, Brett Brotherton, Timothy Sherwood, Ryan Kastner, Timothy Levin, Thuy D Nguyen, and Cynthia Irvine. 2008. Managing security in FPGA-based embedded systems. *IEEE Design & Test of Computers* 25, 6 (2008), 590–598.

[13] Andreas Kuehlmann, Viresh Paruthi, Florian Krohm, and Malay K Ganai. 2002. Robust Boolean reasoning for equivalence checking and functional property verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21, 12 (2002), 1377–1394.

[14] Akshita Reddy Mavurapu, Haoqi Shan, Xiaolong Guo, Orlando Arias, and Dean Sullivan. 2023. HeisenTrojans: They Are Not There Until They Are Triggered. In *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*. IEEE, 1–7.

[15] Adib Nahiyan, Mehdi Sadi, Rahul Vittal, Gustavo Contreras, Domenic Forte, and Mark Tehranipoor. 2017. Hardware trojan detection through information flow security verification. In *2017 IEEE International Test Conference (ITC)*. IEEE, 1–10.

[16] Raphael Proulx, Faisal Alsaadi, and Yier Jin Gao. 2023. A Survey on FPGA Cybersecurity Design Strategies. *Comput. Surveys* (2023).

[17] Eduardo Rhod, Behnam Ghavami, Zhenman Fang, and Lesley Shannon. 2023. A cycle-accurate soft error vulnerability analysis framework for FPGA-based designs. *arXiv preprint arXiv:2303.12269* (2023).

[18] Arun Saravanan, Dhafer Almakhles, and Mohammed Alam. 2025. SynFuzz: Leveraging Fuzzing of Netlists to Detect Synthesis Bugs in FPGA Toolchains. *arXiv preprint arXiv:2504.18812* (2025). https://arxiv.org/abs/2504.18812

[19] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. 2024. The emergence of hardware fuzzing: A critical review of its significance. *arXiv preprint arXiv:2403.12812* (2024).

[20] Raghul Saravanan and Sai Manoj Pudukotai Dinakarrao. 2024. The fuzz odyssey: A survey on hardware fuzzing frameworks for hardware design verification. In *Proceedings of the Great Lakes Symposium on VLSI 2024*. 192–197.

[21] Sergei Skorobogatov and Christopher Woods. 2012. Breakthrough silicon scanning discovers backdoor in military chip. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 23–40.

[22] Flavien Solt and Kaveh Razavi. 2025. Lost in Translation: Enabling Confused Deputy Attacks on EDA Software with TransFuzz. *USENIX Security. Paper= https://comsec. ethz. ch/wp-content/files/mirtl_sec25. pdfURL= https://comsec. ethz. ch/mirtl* (2025).

[23] Berkeley Logic Synthesis and Verification Group. 2020. *ABC: A System for Sequential Synthesis and Verification*. Technical Report. University of California, Berkeley. https://github.com/berkeley-abc/abc

[24] US Department of Defense. 2022. Field Programmable Gate Array Best Practices – Threat Catalog. https://media.defense.gov/2022/Dec/08/2003127935/-1/-1/0/CTR_DOD_MICROELECTRONICS-FPGA_BEST_PRACTICES_THREAT_CATALOG.PDF Accessed: 2025-11-12.

[25] Zun Wang, He Jiang, Xiaochen Li, Shikai Guo, Xu Zhao, and Yi Zhang. 2025. Insights From Bugs in FPGA High-Level Synthesis Tools: An Empirical Study of Bambu Bugs. *IEEE Transactions on Reliability* (2025).

[26] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*. 97.

[27] Zhihao Xu, Jiacheng Liu, Yunlong Wang, and Tao Chen. 2024. A Novel HDL Code Generator for Effectively Testing FPGA Logic Synthesis Compilers. *arXiv preprint arXiv:2407.12037* (2024). https://arxiv.org/abs/2407.12037

[28] Salam R Zantout. 2018. *Hardware Trojan Detection in FPGA through Side-Channel Power Analysis and Machine Learning*. Master's thesis. University of California, Irvine.

[29] Jiliang Zhang and Gang Qu. 2019. Recent attacks and defenses on FPGA-based systems. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 12, 3 (2019), 1–24.

[30] Yi Zhang, Liang Chen, and Tianyu Zhao. 2025. A Novel Mutation-Based Method for Detecting FPGA Logic Synthesis Tool Bugs. *arXiv preprint arXiv:2508.15536* (2025).

[31] Yi Zhang, He Jiang, Shikai Guo, Xiaochen Li, Hui Liu, and Chongyang Shi. 2025. Toward Understanding FPGA Synthesis Tool Bugs. *ACM Transactions on Software Engineering and Methodology* 34, 7 (2025), 1–37.