Improving code vulnerability detection using LLMs

Tudor-Andrei FĂRCĂȘANU, Traian Eugen REBEDEA
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
tudor.farcasanu@stud.acs.upb.ro, traian.rebedea@upb.ro

February 15, 2026

Contents

Abstract

Large Language Models (LLMs) show promise for automated vulnerability detection but struggle to distinguish between vulnerable code and its patched counterpart. This research advances LLM-based vulnerability detection through two approaches: dataset consolidation and reinforcement learning optimization. We merge three vulnerability datasets (PrimeVul, SVEN, SecVulEval) into approximately 6,600 unique pairs, then use a 32B teacher model to generate reasoning-augmented training data. Supervised fine-tuning of Qwen3-4B on this dataset achieves a 35% relative improvement in pair-wise classification accuracy (P-C from 19.4% to 26.2%) on the PrimeVul benchmark, continuing improvements from our prior LoRA-based approach. We also explore Group Relative Policy Optimization (GRPO) for further refinement, though infrastructure limitations restricted training to a small subset (64 samples). The results show that more diverse, reasoning-focused training data improves vulnerability discrimination, with the fine-tuned model showing reduced benign bias.

1 Introduction

This research aims to enhance the capabilities of Large Language Models (LLMs) in automated code vulnerability detection by combining dataset consolidation and reinforcement learning optimization. Building upon our previous research demonstrating that fine-tuned LLMs can improve vulnerability discrimination by 46% in pair-wise classification accuracy, this project pursues two key advancements to push the boundaries of LLM-based vulnerability detection.

The main objectives are to increase the diversity and scale of training data by merging multiple existing vulnerability datasets while minimizing redundancy across complementary data sources, and applying reinforcement learning techniques to further optimize model behavior through targeted reward signals that capture the nuances of vulnerability detection tasks. The expected outcome is a vulnerability detection system that achieves superior discrimination between vulnerable and patched code, scales effectively to larger and more diverse datasets, and generalizes better to novel vulnerability patterns and edge cases.

2 Related Work

2.1 The Challenge of Differentiating Vulnerable and Patched Code

Recent works highlight the difficulty LLMs face in distinguishing between vulnerable code and its minimally altered, patched version. The PrimeVul paper [1] highlighted this issue by introducing a benchmark composed of paired vulnerable and fixed code functions and proposed a set of pair-wise evaluation metrics (P-C, P-V, P-B, P-R) to specifically measure this discriminative capability. Their findings revealed that even powerful models such as GPT-4 struggled to correctly classify both samples in a vulnerable/benign pair, with P-C scores indicating significant room for improvement. This observation suggests that LLMs may be overfitting to superficial textual features rather than grasping the deep, underlying semantic causes of vulnerabilities.

2.2 Real-World Vulnerability Detection at Scale

The CyberGym benchmark [2] demonstrates the complexity of real-world vulnerability detection through a large-scale evaluation framework featuring 1,507 vulnerabilities across 188 open-source projects discovered by OSS-Fuzz. Rather than simple classification tasks, CyberGym requires AI agents to generate proof-of-concept tests that reproduce vulnerabilities given only vulnerability descriptions and source code repositories containing thousands of files and millions of lines of code. The benchmark reveals that even the top-performing AI agent combinations (OpenHands with Claude-3.7-Sonnet) achieve only 11.9% success rate on the primary task, underscoring the substantial gap between current capabilities and practical vulnerability analysis. Beyond benchmarking, CyberGym demonstrates tangible real-world impact: agents discovered 35 zero-day vulnerabilities and identified 17 historically incomplete patches in production software.

Building on the challenge of repository-scale analysis, Just-In-Time (JIT) vulnerability detection has emerged as another problem because vulnerabilities often manifest through multi-hop function calls across repositories rather than within isolated functions. The JitVul benchmark [3] addresses this limitation by linking each vulnerable function to its vulnerability-introducing and fixing commits across 879 CVEs spanning 91 vulnerability types. JitVul demonstrates that LLM-based agents with reasoning capabilities outperform isolated function-level analysis, with ReAct agents leveraging thought-action-observation patterns achieving superior discrimination between vulnerable and benign code compared to standard LLM prompting.

2.3 Explainability and Interpretability in Vulnerability Detection

Understanding why models make vulnerability predictions is increasingly important for both security practitioners and model developers. Recent work on Explainable Vulnerability Detection [4] using edge-aware graph attention networks demonstrates how to compute relevance scores for both nodes and edges in Code Property Graphs by leveraging attention weights and input gradients, then mapping these back to source-level constructs for human interpretation. This approach enables security teams to understand which specific code elements the model identified as problematic, bridging the gap between black-box model predictions and actionable insights.

2.4 Foundational Deep Learning and Graph-Based Methods

While LLM-based approaches represent the current frontier, foundational deep learning methods remain important references. Graph Neural Networks operating on Abstract Syntax Trees and Control Flow Graphs [5] have demonstrated effective vulnerability detection by capturing semantic program structure. Deep learning systems like VulDeePecker [6] and SySeVR [7] pioneered neural approaches to vulnerability identification. More recent dataflow-inspired approaches [8] show that combining traditional program analysis insights with deep learning continues to yield improvements. These foundational techniques inform our understanding of what semantic patterns are critical for vulnerability detection and suggest that enhancing LLMs with structured reasoning and rich training signals is a promising direction.

3 Data Preparation and Merging

The first phase merged multiple vulnerability datasets to maximize data diversity. I implemented a merging pipeline that combines three primary data sources for now:

PrimeVul serves as the primary foundation, providing paired vulnerable and patched code functions from CVEs. The dataset includes commit-level metadata linking each vulnerability to its fix, and vulnerability descriptions.

SVEN contributes 1,606 manually-verified functions (803 vulnerable, 803 non-vulnerable) covering 9 CWE types with claimed 94% label accuracy. SVEN's distinguishing feature is its detailed metadata including the specific lines that were changed between vulnerable and patched versions as well as character level changes.

SecVulEval adds additional CVE and CWE linked samples, with line level change metadata.

The merging process follows a priority-based deduplication strategy using `commit_id` as the primary key. When samples from multiple sources share the same commit, I prioritized SVEN data (due to its manual verification), then merged CVE descriptions from any available source. The merging script preserves all `line_changes` information, which is critical for training the model to identify specific vulnerable lines.

```
{
  "source": "primevul",
  "commit_id": "239c4f7...",
  "project": "wget",
  "cve": "CVE-2015-7665",
  "cve_desc": "Tails before 1.7 includes the wget program but
     does not prevent automatic fallback from passive FTP...",
  "vuln_func": "void function(...) { ... }",
  "patched_func": "void function(...) { ... }",
  "deleted_lines": [
    {"line_no": 645, "text": "err = ftp_do_port (csock, ...);"}
  ],
  "added_lines": []
}
```

Listing 1: Example of a merged dataset entry (structure)

After deduplication and filtering out samples where the vulnerable function is identical to the patched function, the consolidated dataset contains approximately 6,600 unique samples.

4 Teacher Model Annotation for Training Data Generation

To create training data for the reinforcement learning phase, I re-used the teacher model annotation pipeline using Qwen3-32B. I generate reasoning traces for each vulnerability sample, following a structured prompt format designed to elicit both classification and localization information.

```
You are an expert code security analyst. Analyze the provided
```

```
code and determine if it contains any security vulnerabilities.

Code:
...
774:            if (!opt.server_response)
775:              logputs (LOG_VERBOSE, "==> RETR ... ");
...
798:        err = ftp_retr (csock, u->file);
...
800:        switch (err)
801:          {
802:          case FTPRERR:
...
904:       }

Context:
- CVE: CVE-2015-7665
- Description: Tails before 1.7 includes the wget program but
  does not prevent automatic fallback from passive FTP to active
  FTP, which allows remote FTP servers to discover the Tor client
  IP address.

Hint: This code contains a security vulnerability.
The changed lines that might be related to the vulnerability are:
Lines: 11, 642, 645, 646, 648.

Task: After your reasoning, provide your analysis in JSON format:
{"classification": "VULNERABLE" or "NOT_VULNERABLE", ... }
```

Listing 2: Example system prompt (truncated) used for teacher model annotation

Each sample is formatted with line numbers to enable vulnerable line identification. The prompt includes contextual hints derived from the merged dataset: CVE identifiers, CWE classifications, CVE descriptions (where available), and the specific lines that were changed. For vulnerable samples, the deleted lines between vulnerable and patched versions serve as ground truth for the `vulnerable_lines` field.

The annotation process prioritizes samples with rich metadata (CVE descriptions). The pipeline uses SGLang [9] for inference with Qwen3-32B, with concurrent requests.

5 Training

I used Qwen3-4B as the training target because it has "thinking mode" capabilities that produce detailed reasoning traces, and fits within the 3×80GB GPU memory constraint for full fine-tuning with DeepSpeed ZeRO-3 [10]. The larger Qwen3-32B serves as the teacher model for annotation. Training ran on UPB's cluster (fep) using 3×80GB GPUs, 64 CPU cores, and 300GB RAM.

5.1 Phase 1: Supervised Fine-Tuning (SFT)

In phase 1 we trained Qwen3-4B to mimic the teacher model's responses. This phase uses the generated dataset where each sample contains the formatted prompt and the teacher's response (including full reasoning trace in "thinking mode"). The model learns to produce

structured JSON outputs with classification, vulnerable line identification, and reasoning summaries.

SFT Hyperparameters:

- Learning rate: 2×10^{-5} with 3% warmup
- Batch size: 1 per device $\times$ 16 gradient accumulation steps
- Max sequence length: 32,768 tokens
- 1 epoch, gradient checkpointing enabled
- DeepSpeed ZeRO-3 [10] across 3 H100 GPUs

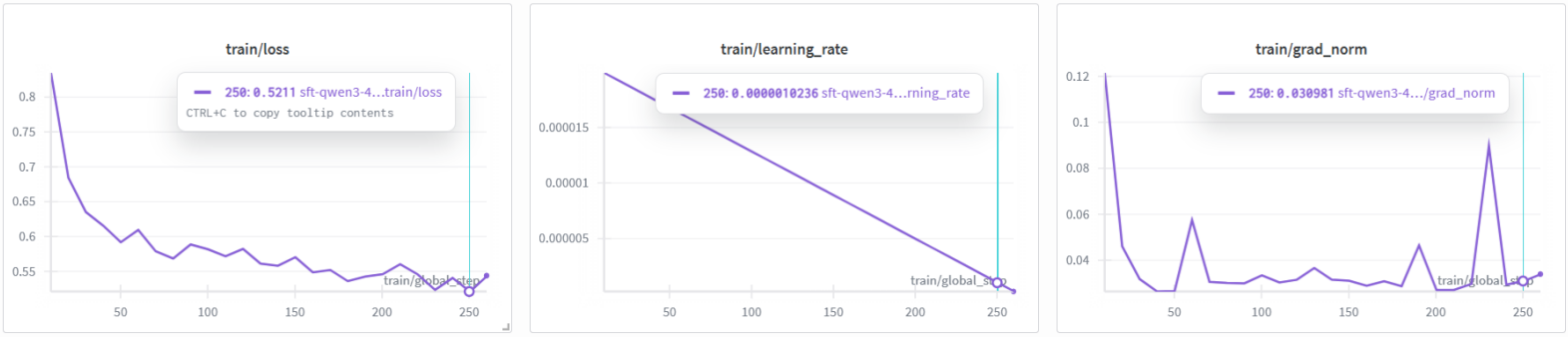

Figure 1: SFT Phase 1 training metrics: training loss, learning rate, and gradient norm.

5.2 Phase 2: Group Relative Policy Optimization (GRPO)

Phase 2 refines the SFT checkpoint using reinforcement learning with a programmatic reward function. GRPO, introduced in the DeepSeekMath paper [11], is a variant of Proximal Policy Optimization (PPO) specifically designed for language model training.

The key innovation of GRPO is eliminating the need for a separate value network (critic), reducing memory overhead by 40–60%. Instead of learning a value function, GRPO estimates the baseline for advantage calculation directly from the rewards of multiple outputs generated for the same prompt. For each prompt q, the model generates G candidate responses $\{o_1, ..., o_G\}$, each receiving a reward r_i. The advantage for each response is computed as:

$$A_i = \frac{r_i - \text{mean}(\{r_1, ..., r_G\})}{\text{std}(\{r_1, ..., r_G\})} \tag{1}$$

This group-relative advantage indicates whether a response is better ($A_i > 0$) or worse ($A_i < 0$) than the group average. The policy is then updated using the standard PPO clipped objective with these advantages [11, 12].

5.2.1 Reward Function

The reward function implements a five-level structure based on classification accuracy and vulnerable line localization:

- **Full reward (1.0):** Correct classification and $\geq$50% of vulnerable lines correctly identified

- **Partial reward (0.6):** Correct classification but insufficient line localization (<50%)
- **Localization-only reward (0.3):** Incorrect classification but some correct vulnerable lines identified
- **Format reward (0.05):** Valid JSON output but wrong classification and no correct lines
- **Zero reward (0.0):** Invalid response format (malformed JSON, incomplete output)

5.2.2 Framework Selection and Challenges

Implementing GRPO training for vulnerability detection unfortunately presented significant engineering challenges using UPB's cluster (fep). I experimented with multiple frameworks before managing to train the model:

TRL + DeepSpeed (Initial Attempt)

I first attempted training using the TRL library's [12] GRPOTrainer with DeepSpeed ZeRO-3 for offload capabilities. Despite extensive debugging, the model consistently produced empty completions during training (completion length = 1 token), even though the same model worked correctly at inference time. The training metrics showed `loss=0.0`, `grad_norm=0.0`, and `completions/mean_length=1.0`, indicating the generation phase was somehow failing silently within DeepSpeed's distributed context.

veRL with Container (Second Attempt)

I then tried veRL [13], using their pre-built container. However, the cluster's outdated Linux kernel and libraries caused CUDA compatibility issues with veRL's dependencies (particularly FlashAttention), resulting in errors including "invalid CUDA kernel" and segmentation faults after rollout, during the training process.

Unsloth (Third Attempt)

I also tried Unsloth [14], but two main issues prevented successful training: Unsloth ignored `gpu_memory_utilization` settings, causing OOM errors, and after working around memory issues, I encountered "RuntimeError: Triton Error [CUDA]: an illegal memory access was encountered" after only 3–5 training steps. This persisted across configurations including full fine-tuning and LoRA with ranks from 512 down to 64.

veRL Native (Final Approach)

After the cluster (very recently) upgraded its Linux kernel and drivers, I successfully installed veRL in a native conda environment with FlashAttention compiled from source. This configuration finally allowed me to finish a GRPO training run with a very small subset of the training data (64 samples, $\approx 0.51\%$ of the training data). However, training on the full dataset still wasn't possible, due to rollout slowing to a crawl after about 450 completions, and then after ≈ 580 the vLLM [15] rollout backend crashes with "CUDA error: Invalid access of peer GPU memory over nvlink," possibly due to driver or even hardware issues.

5.2.3 GRPO Hyperparameters (veRL)

- Learning rate: 1×10^{-6}
- Training batch size: 60 prompts per iteration
- Mini-batch size: 12, Micro-batch size: 1 per GPU
- Number of generations per prompt (G): 4, Backend: vLLM [15]
- Temperature: 0.6, Top-P: 0.95, Top-K: 20 (Qwen3 thinking mode defaults)
- **KL coefficient ($\beta = 0$):** Following recent research showing that KL penalties are not essential for GRPO-style training [16, 17], I disabled the KL divergence term. This eliminates the reference model, saving GPU memory and allowing the policy more freedom to explore during optimization, as well as speeding up the training process.
- Max context: 40,960 tokens (28681 tokens for prompt + 12279 tokens for response)
- FSDP2 with parameter offload using 3 H100 GPUs

5.2.4 Training Observations

Due to the extremely long generation times encountered for the full dataset (12,554 training samples with 40,960 token context windows), I validated the pipeline on a small subset (64 training samples and 12 validation samples), as the rollout phase became extremely slow after approximately 450 completions. During the limited training run, the average reward stabilized around 0.17–0.19, most likely due to more training data being needed for meaningful improvement.

6 Evaluation

Evaluation was conducted on a combined test set consisting of:

- **PrimeVul test set:** 407 vulnerability pairs (814 samples)
- **PrimeVul validation set:** 354 vulnerability pairs
- **SVEN validation set:** 83 samples (unused during training)

Two prompt formats were tested:

1. **Training format:** The same structured prompt used during training (with line numbers, requesting JSON output), but *without* any hints about vulnerability status or changed lines.
2. **std_cls format:** Simple classification prompt from Semester 2, shown below.

```
Please analyze the following code:
```
{code_snippet}
```
Please indicate your analysis result with one of the options:
(1) YES: A security vulnerability detected.
(2) NO: No security vulnerability.

```

```
Only reply with one of the options above. Do not include any further
    information.
```

Listing 3: std_cls prompt format from Semester 2

7 Results

Pair-wise Metrics

The pair-wise metrics measure the model's ability to correctly classify both samples in a vulnerability pair (vulnerable function + patched function):

- **P-C (Pair-Correct)** ↑: Both samples correctly classified
- **P-V (Pair-Vulnerable)** ↓: Both incorrectly labeled as vulnerable (false positive bias)
- **P-B (Pair-Benign)** ↓: Both incorrectly labeled as benign (false negative bias)
- **P-R (Pair-Reversed)** ↓: Labels swapped (vulnerable↔patched)

7.1 PrimeVul Test Set Results

Table 1 shows results using the training prompt format:

Table 1: PrimeVul Test Results – Training Prompt Format (407 pairs)

Model	Acc	F1	P-C (%) ↑	P-V (%) ↓	P-B (%) ↓	P-R (%) ↓
Qwen3-4B (Base)	0.493	0.476	19.4	26.8	33.0	20.8
Qwen3-4B (SFT)	**0.560**	**0.589**	**26.2**	36.1	23.0	14.7
Qwen3-4B (GRPO)†	0.515	0.536	22.1	33.9	25.5	18.5

Table 2 shows results using the std_cls prompt format:

Table 2: PrimeVul Test Results – std_cls Prompt Format (407 pairs)

Model	Acc	F1	P-C (%) ↑	P-V (%) ↓	P-B (%) ↓	P-R (%) ↓
Qwen3-4B (Base)	0.503	0.189	6.2	5.4	83.0	5.4
Qwen3-4B (SFT)	**0.554**	**0.533**	**26.3**	24.6	33.7	15.5
Qwen3-4B (GRPO)†	0.528	0.506	24.3	24.1	32.9	18.7

7.2 Combined Dataset Results

Tables 3 and 4 show results on the full combined dataset (PrimeVul test + valid + SVEN val, 938 pairs total):

Table 3: Combined Dataset Results – Training Prompt Format (938 pairs)

Model	Acc	F1	P-C (%) ↑	P-V (%) ↓	P-B (%) ↓	P-R (%) ↓
Qwen3-4B (Base)	0.522	0.522	20.4	31.6	31.1	16.8
Qwen3-4B (SFT)	**0.549**	**0.582**	**25.5**	37.5	20.6	16.4
Qwen3-4B (GRPO)†	0.533	0.571	22.0	40.0	22.4	15.6

Table 4: Combined Dataset Results – std_cls Prompt Format (938 pairs)

Model	Acc	F1	P-C (%) ↑	P-V (%) ↓	P-B (%) ↓	P-R (%) ↓
Qwen3-4B (Base)	0.523	0.266	9.5	7.1	77.7	5.6
Qwen3-4B (SFT)	**0.550**	**0.539**	**26.5**	26.0	30.8	16.7
Qwen3-4B (GRPO)†	0.548	0.537	24.1	28.1	32.7	15.1

†GRPO trained on small subset only (64 training, 12 validation samples) due to training issues.

7.3 Analysis

The SFT model achieves consistent improvements across all configurations: 6–7 % gains in accuracy and F1, and a 35% relative improvement in P-C (from 19.4% to 26.2% on PrimeVul test). P-B (benign bias) drops from 33% to 23%, indicating the model is less likely to dismiss vulnerabilities as benign.

The base Qwen3-4B shows drastically different behavior depending on prompt format:

- With std_cls prompt: (P-B=83%, P-C=6.2%) conservative, rarely predicts vulnerabilities
- With training prompt: (P-B=33%, P-C=19.4%) balanced, actively attempts classification

This suggests the base model's usefulness for this task is highly prompt-dependent, and that JSON structured prompts with line numbers elicit more engaged analysis.

After SFT, the model maintains consistent behavior across both prompt formats (P-C of 26.2% vs 26.3%), showing that fine-tuning successfully taught the model to engage with the vulnerability detection task regardless of prompt style.

While SFT improves overall detection, P-V (both samples labeled vulnerable) increases from 26.8% to 36.1%. The trade-off of catching more true vulnerabilities at the cost of more false positives may be acceptable in security contexts where missing a vulnerability is more costly than investigating a false alarm.

The GRPO model trained on only 64 samples shows slightly lower performance than SFT. This is unsurprising given the limited training data. The average reward during GRPO training (0.17–0.19) indicates most samples received low rewards, suggesting either insufficient training data or limitations in the discrete reward signal.

7.3.1 Comparison with Semester 2 (Qwen3-8B + LoRA)

In the second semester research report, we fine-tuned Qwen3-8B using LoRA on a smaller dataset (816 pairs from PrimeVul only). That approach achieved P-C of 18.67% (up from 12.78% baseline), a 46% relative improvement. The current Semester 3 full SFT on Qwen3-4B with the expanded dataset (6,600 samples) achieves P-C of 26.2%, representing a further 40% relative improvement over the Semester 2 results. This progression demonstrates (unsurprisingly) that larger, more diverse training data improves performance, and that full fine-tuning can be more effective than LoRA, even on a smaller model.

Table 5: Comparison: Semester 2 (Qwen3-8B + LoRA) vs. Semester 3 (Qwen3-4B + SFT) – std_cls prompt

Model	Acc	F1	P-C (%) ↑	P-V (%) ↓	P-B (%) ↓	P-R (%) ↓
Semester 2 (816 pairs training data, LoRA)						
Qwen3-8B (Base)	0.528	0.325	12.78	9.09	70.02	8.11
Qwen3-8B (LoRA)	0.528	0.427	18.67	14.74	52.58	14.00
Semester 3 (≈ 6600 pairs training data, full SFT)						
Qwen3-4B (Base)	0.503	0.189	6.2	5.4	83.0	5.4
Qwen3-4B (SFT)	**0.554**	**0.533**	**26.3**	24.6	33.7	15.5

7.4 Examples

To illustrate the practical impact of fine-tuning, we present examples where the SFT model correctly detected vulnerabilities that the base model missed:

Example 1: TensorFlow – Missing Parameter Validation (CWE-20)

```
Status SetUnknownShape(const NodeDef* node, int output_port) {
    shape_inference::ShapeHandle shape =
        GetUnknownOutputShape(node, output_port);

    InferenceContext* ctx = GetContext(node);
    if (ctx == nullptr) {
        return errors::InvalidArgument("Missing context");
    }

    ctx->set_output(output_port, shape);  // output_port NOT validated!
    return Status::OK();
}
```

Listing 4: TensorFlow SetUnknownShape function with missing validation

The vulnerability occurs because `output_port` is validated in `GetUnknownOutputShape` but not before being passed to `ctx->set_output()`, which could lead to out-of-bounds access. The base model predicted this as **NOT VULNERABLE**, while the SFT model correctly identified it as **VULNERABLE**. This is reflected in their reasoning:

- Base: "...No user input, internal API, ctx null check exists..."
- SFT: "...output_port could be out-of-bounds, no validation before set_output..."

Example 2: deark – Division by Zero from File Input (CWE-369)

```
void fmtutil_macbitmap_read_pixmap_only_fields(deark *c,
        dbuf *f, struct fmtutil_macbitmap_info *bi, i64 pos) {

    bi->pixelsize = dbuf_getu16be(f, pos+18);  // Read from file

    // No validation of pixelsize!
    bi->pdwidth = (bi->rowbytes * 8) / bi->pixelsize;  // Division by
   zero!

    if (bi->pdwidth < bi->npwidth) {
        bi->pdwidth = bi->npwidth;
    }
}
```

Listing 5: deark bitmap parsing with division by zero vulnerability

Here, `pixelsize` is read directly from user-controlled file input without validation. If the file contains 0 for this field, it causes a division by zero crash. The base model missed this vulnerability, assuming valid data and predicting **NOT VULNERABLE**, while the SFT model treats file input as untrusted, and correctly detected it as **VULNERABLE**. Again, this is reflected in the reasoning:

- Base: "...Processing known file format, data expected to be valid..."
- SFT: "...pixelsize read from untrusted input, division by zero if zero..."

Example 3: TensorFlow – Unchecked Iterator Access (CWE-824)

```
Status GetInitOp(...) {
  const auto& init_op_sig_it =
      meta_graph_def.signature_def().find(kSavedModelInitOpSignatureKey)
   ;
  if (init_op_sig_it != sig_def_map.end()) {
    *init_op_name = init_op_sig_it->second.outputs()
                        .find(kSavedModelInitOpSignatureKey)
                        ->second.name();  // Potential crash!
  }
  return Status::OK();
}
```

Listing 6: TensorFlow GetInitOp with unchecked iterator dereference

The vulnerability occurs when accessing `init_op_sig_it->...->second.name()`. If the inner `find()` returns `end()`, dereferencing it causes undefined behavior (crash or memory corruption). The SFT model's reasoning:

```
Another angle: Are there any unchecked assumptions? When accessing
   init_op_sig_it->second.outputs().find(...)}, if the outputs don't
   have the key, find() would return an iterator to end, and then
   accessing .name() would be undefined behavior... This is a possible
   use-after-free or crash, which could be exploited.
```

Example 4: GPAC – Out-of-Bounds Array Access (CWE-125)

```
static s32 svc_parse_slice(GF_BitStream *bs, AVCState *avc,
                           AVCSliceInfo *si) {
    s32 pps_id = gf_bs_read_ue_log(bs, "pps_id");
```

```
    if (pps_id > 255) return -1;

    si->pps = &avc->pps[pps_id];            // Checked OK
    si->pps->id = pps_id;

    si->sps = &avc->sps[si->pps->sps_id + GF_SVC_SSPS_ID_SHIFT];  // NOT
    CHECKED!
    if (!si->sps->log2_max_frame_num) return -2;
    // ...
}
```

Listing 7: GPAC SVC slice parsing with missing bounds check

The code validates `pps_id` before using it as an array index, but fails to validate `sps_id + GF_SVC_SSPS_ID_SHIFT` before accessing the `sps` array. If `sps_id` is large, this causes out-of-bounds memory access. The base model missed this because it saw the `pps_id` validation and assumed all indices were checked. The SFT model correctly identified the missing validation on the second array access.

Example 5: TensorFlow – Memory Exhaustion (CWE-400) – False Negative

This example shows a case where *both* models failed:

```
Status ConstantFolding::IsSimplifiableReshape(const NodeDef& node,
                              const GraphProperties& properties) const {
  const NodeDef* new_shape = node_map_->GetNode(node.input(1));
  if (!IsReallyConstant(*new_shape)) {
    return errors::InvalidArgument("Shape must be constant");
  }

  TensorVector outputs;
  Status s = EvaluateNode(*new_shape, TensorVector(), &outputs);

  if (outputs[0]->dtype() == DT_INT32) {
    std::vector<int32> shp;
    for (int i = 0; i < outputs[0]->NumElements(); ++i) {  // No bounds
   check!
      int32_t dim = outputs[0]->flat<int32>()(i);
      shp.push_back(dim);  // Memory exhaustion if NumElements() is huge
    }
  }
}
```

Listing 8: TensorFlow constant folding with unbounded allocation

The vulnerability: `NumElements()` comes from user-controlled input (the tensor shape). An attacker can craft a tensor claiming to have 2^{31} elements, causing the `push_back` operations to exhaust memory (Denial of Service). Both models were distracted by the error handling and missed the missing bounds check on the loop iteration count.

These examples demonstrate that fine-tuning helps the model recognize vulnerability patterns including: parameter validation requirements across function calls, security implications of untrusted input in arithmetic operations, unchecked iterator dereferences, and missing bounds checks on array indices. However, other vulnerabilities like resource exhaustion through unbounded loops remain challenging.

8 Conclusion and Future Work

Supervised fine-tuning using our consolidated dataset significantly improves LLM-based vulnerability detection. The SFT model achieved a 35% relative improvement in pair-wise classification accuracy (P-C) over the base Qwen3-4B model on the PrimeVul test set, with improvements of 6–7 percentage points in accuracy and F1 scores across evaluation configurations. The consolidated dataset pipeline successfully merged PrimeVul, SVEN, and SecVulEval sources, creating approximately 6,600 unique samples with rich metadata including CVE descriptions and line-level change information.

However, GRPO reinforcement learning training proved challenging due to infrastructure limitations. Despite experimenting with multiple frameworks (TRL+DeepSpeed, veRL container, Unsloth, veRL native), only training on a small subset (64 samples) succeeded. The limited GRPO training did not yield improvements over SFT, with evaluation showing slightly lower performance, due to insufficient training data.

Future work could include:

- Investigating the rollout slowdowns and various errors to enable training on the full dataset.
- Experimenting with recent advances such as DAPO [16], which should offer improved training stability and efficiency.
- Choosing newer, maybe code-specialized models (e.g. Qwen3-Coder-Next, DeepSeek-V3.2, Kimi K2.5) that may have better inherent understanding of code security patterns.
- The current five-tier discrete reward may not prove that relevant, perhaps SAST tools could be used to provide more fine-grained feedback.

Resources

Models and datasets from this research are publicly available:

- SFT Model
- GRPO Model (veRL, 64 training samples)
- GRPO Model (Unsloth, 5 steps)
- Merged Dataset
- Fine-tuning Dataset

References

[1] 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?, 2024.

[2] Zhun Wang, Tianneng Shi, Jingxuan He, Matthew Cai, Jialin Zhang, and Dawn Song. Cybergym: Evaluating ai agents' real-world cybersecurity capabilities at scale, 2025.

[3] Alperen Yildiz, Sin G. Teo, Yiling Lou, Yebo Feng, Chong Wang, and Dinil M. Divakaran. Benchmarking llms and llm-based agents in practical vulnerability detection for code repositories, 2025.

[4] Radowanul Haque, Aftab Ali, Sally McClean, and Naveed Khan. Explainable vulnerability detection in c/c++ using edge-aware graph attention networks, 2025.

[5] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks, 2019.

[6] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings 2018 Network and Distributed System Security Symposium*, NDSS 2018. Internet Society, 2018.

[7] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Yawei Zhu, and Zhaoxuan Chen. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, July 2022.

[8] Benjamin Steenhoek, Hongyang Gao, and Wei Le. Dataflow analysis-inspired deep learning for efficient vulnerability detection, 2023.

[9] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Jeff Huang, Chuyue Sun, Cody Hao Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E. Gonzalez, Clark Barrett, and Ying Sheng. Sglang: Efficient execution of structured language model programs, 2024.

[10] Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020.

[11] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. Deepseekmath: Pushing the limits of mathematical reasoning in open language models, 2024.

[12] Leandro von Werra, Younes Belkada, Lewis Tunstall, Edward Beeching, Kashif Sharma, Shengyi Huang, Rishabh Mishra, Nathan Manber, et al. Trl: Transformer reinforcement learning, 2020.

[13] Guangming Sheng, Chi Zhang, Zilingfeng Ye, Xibin Wu, Wang Zhang, Ru Zhang, Yanghua Peng, Haibin Lin, and Chuan Wu. Hybridflow: A flexible and efficient rlhf framework, 2024.

[14] Daniel Han, Michael Han, and Unsloth team. Unsloth, 2023.

[15] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP)*, 2023.

[16] Qiying Yu, Zheng Zhang, Ruofei Wan, Xin Xing, and Baichun Wu. Dapo: An open-source llm reinforcement learning system at scale, 2025.

[17] Yiping Liu, Zijian Li, Michael Hall, Percy Liang, and Tatsunori Hashimoto. Understanding r1-zero-like training: A critical perspective, 2025.