Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

TRACE: Patch-Aware Retrieval-Augmented Reasoning for High-Recall Vulnerability Triage

Dihao Zhang
Nantong University
Xiaolin Ju
ju.xl@ntu.edu.cn
Nantong University
Zhan Gao
Nantong University
Xiang Chen
Nantong University
Lina Gong
Nanjing University of Aeronautics and Astronautics

Research Article

Keywords: Vulnerability detection, Large language models, Retrieval-augmented reasoning, Security patch analysis, Program dependence analysis

Posted Date: November 13th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11482209/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: No competing interests reported.

TRACE: Patch-Aware Retrieval-Augmented Reasoning for High-Recall Vulnerability Triage

Dihao Zhang[1], Xiaolin Ju[1*], Zhan Gao[1], Xiang Chen[1], Lina Gong[2*]

[1]School of Artificial Intelligence and Computer Science, Nantong University, Nantong, China.
[2]School of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing, China.

*Corresponding author(s). E-mail(s): ju.xl@ntu.edu.cn; gonglina@nuaa.edu.cn;
Contributing authors: zhangdihao74@gmail.com; gaozhan@ntu.edu.cn; xchencs@ntu.edu.cn;

Abstract

Security patches often preserve dangerous operations while changing the conditions under which they execute. LLM-based vulnerability detectors must therefore distinguish the presence of such operations from the persistence of their vulnerability causes. We present TRACE, a patch-aware retrieval-augmented framework for high-recall, function-level vulnerability triage. TRACE links structural evidence from historical vulnerable/fixed code pairs with explicit repair principles, retrieves relevant contrasts through dependency-aware re-ranking, and guides an LLM to check dangerous-path alignment, vulnerability-cause matching, and equivalent safeguards in the target function. Inference requires no target-side repair patch. On 586 PairVul repair pairs, TRACE achieves 93.52% recall and 75.48% F1, compared with 95.05% and 73.58% for VulInstruct. It correctly classifies both revisions in 40.27% of pairs, versus 34.98%, and produces 53 fewer false positives on repaired functions. On a fixed 301-function SecVulEval subset, TRACE achieves 92.50% recall and 70.48% F1, compared with 78.33% and 62.88% for VulInstruct. The F1 differences on both datasets and the PairVul pair-accuracy difference do not reach significance under Holm adjustment. Ablations show lower F1 without repair principles or fixed-side evidence, while error analysis identifies safeguard-equivalence failures as a recurring source of false positives in audited cases. These findings suggest that linked repair contrasts can support high-recall vulnerability triage, although reliably recognizing equivalent safeguards remains a limitation.

Keywords: Vulnerability detection, Large language models, Retrieval-augmented reasoning, Security patch analysis, Program dependence analysis

1 Introduction

A security patch can leave a function nearly unchanged while changing whether unsafe conditions can reach a dangerous operation. A memory access, pointer dereference, or resource release may remain in both revisions; the repair instead introduces a bound check, adds an early return, enforces a lifetime constraint, or changes the order of operations. Detecting the operation alone therefore cannot distinguish the vulnerable revision from its repaired counterpart. In developer-facing vulnerability triage, warnings that overlook these protective changes consume review attention without identifying an additional defect.

Large language models (LLMs) can interpret source code and describe plausible vulnerability causes, but a plausible explanation of a hazard does not establish that the relevant safety condition is violated in a given function. For example, a memory-copy operation does not imply an out-of-bounds write if all paths reaching it enforce the destination bound. Conversely, an apparent safeguard may be ineffective if it checks the wrong value or executes only after the dangerous operation. A well-supported vulnerability warning must therefore connect a reachable dangerous operation to its failure condition and determine whether an applicable safeguard blocks that condition. Such reasoning remains challenging when the available evidence is limited to a function and its local dependency context (Ullah et al. 2024; Ding et al. 2025; Guo et al. 2024b; Mao et al. 2025).

Earlier vulnerability detectors learn from code tokens, vulnerability-focused slices, or program structure (Li et al. 2018, 2021b; Zhou et al. 2019; Nguyen et al. 2022; Fu and Tantithamthavorn 2022); more recent LLM systems incorporate security specifications, static analysis, or external vulnerability knowledge (Li et al. 2025; Zhu et al. 2026; Du et al. 2026; Lewis et al. 2020). In particular, Vul-RAG uses vulnerability causes and fixing solutions as retrieved knowledge, while VulInstruct derives security specifications from vulnerability-related examples and patches. Thus, neither repair-aware retrieval nor security-condition reasoning is new in itself. The remaining challenge is preserving the relationship between a historical failure path and the protective condition its repair introduces when transferring evidence to an unseen target. TRACE addresses this challenge by linking vulnerable/fixed structural instances with their repair principles, selecting relevant contrasts through dependency-aware retrieval, and verifying whether the target reproduces the failure cause without an equivalent safeguard. Its contribution lies in this linked evidence organization and target-side verification protocol, rather than in the general use of patches or retrieval.

The useful unit of evidence is therefore a *repair contrast*, not a vulnerability example in isolation. Vulnerability-fix datasets and repair studies provide paired historical code changes, while framing repair as distinct from proving the safety of a different target function (Bhandari et al. 2021; Fan et al. 2020; Fu et al. 2022; Pearce et al. 2023). A patch does not constitute a general safety proof and may include unrelated changes.

Its value lies in combining both revisions, their dependence context, and an articulated security obligation that can be checked against a different function. Consequently, the retrieval task is to identify a relevant *mechanism*, and the inference task is to determine whether that mechanism still fails in the target.

We introduce TRACE for high-recall, function-level triage. TRACE first converts historical vulnerable/fixed revisions into linked instance-level structural evidence and principle-level repair knowledge. Next, it retrieves structurally aligned repair pairs through dense candidate recall and dependency-aware re-ranking. An LLM then checks dangerous-path alignment, vulnerability-cause matching, and the presence of equivalent safeguards to reach an evidence-grounded decision. The system is an inspection aid rather than a whole-program verifier: its conclusions are conditioned on the target context and repair evidence available at inference time.

A patch-aware detector should be evaluated on whether it recognizes repairs, not just on how many vulnerable functions it flags. Function-level recall alone can look favorable even if a method labels both sides of most repair pairs as vulnerable. We therefore complement function-level metrics with exact *pair accuracy*, which requires simultaneously identifying the vulnerable revision and dismissing its repaired counterpart. This distinction leads to a more discriminating reading of the main comparison: TRACE reaches 75.48% F1 and 93.52% recall on PairVul, compared with 73.58% F1 and 95.05% recall for VulInstruct; its F1 margin is small and not statistically significant in the reported comparison. Yet it reduces repaired-side false alarms by 53 and increases pair accuracy from 34.98% to 40.27%. On SecVulEval, the corresponding F1 scores are 70.48% and 62.88%. These outcomes warrant investigating both repair recognition and the remaining reasons for misclassification.

The contributions of this study are threefold:

1. A *repair-contrast evidence formulation* that couples vulnerable and fixed structural instances to the security condition established by their difference, making both the suspected failure path and a possible equivalent safeguard explicit.
2. A *patch-aware reasoning pipeline* in which dependency-guided evidence construction, two-stage structural retrieval, and linked repair-principle reasoning jointly constrain function-level triage without requiring a target-side patch at inference time.
3. An *evaluation of the resulting decision behavior* that separates vulnerable-function recall from repair-pair discrimination and investigates components, independent knowledge sources, LLM backbones, warning burden, and diagnostic errors.

2 Problem and Motivating Example

2.1 A concrete repair contrast in Linux KVM

The central task is to decide whether a target function reproduces the security-relevant behavior of a historical vulnerable implementation, taking its fix into account. CVE-2022-2153 (CWE-476) provides a concrete instance from Linux KVM. The National Vulnerability Database records CVE-2022-2153 as a Linux KVM null-pointer dereference (CWE-476) (National Institute of Standards and Technology 2022). In the affected function `kvm_irq_delivery_to_apic_fast`, the `APIC_DEST_SELF` branch

dereferences `src->vcpu` without first excluding a null `src`; the associated upstream commit documents the vulnerable path and its repair (Kuznetsov 2022a).

The repair patch shown in Listing 1 inserts an early guard before the call to `kvm_apic_set_irq`. When `src` is null, `KVM_BUG_ON` records the invalid condition, sets the delivered-interrupt count to zero, and returns before the dereference. Importantly, the potentially dangerous operation remains in the repaired function, but the added guard prevents execution from reaching it with a null source pointer along the self-destination branch. The security-relevant difference between the vulnerable and repaired versions therefore lies not in the presence of a dangerous operation, but in whether the required safety precondition is enforced before that operation.

```
  if (irq->shorthand == APIC_DEST_SELF) {
+     if (KVM_BUG_ON(!src, kvm)) {
+         *r = 0;
+         return true;
+     }
      *r = kvm_apic_set_irq(src->vcpu, irq, dest_map);
      return true;
  }
```

Listing 1 Security-relevant excerpt of the Linux KVM repair patch for CVE-2022-2153. Plus signs indicate added lines, while unchanged lines are shown without a diff prefix. The added guard prevents the local null-pointer dereference along the self-destination branch.

This function-local guard is one part of a broader upstream repair series that also strengthens upstream SynIC preconditions (Kuznetsov 2022b). We do not equate a single guard with a whole-system safety proof. Instead, it provides a concrete repair principle: *a non-null source APIC is required before the self-directed interrupt path dereferences the source pointer*. A different target function may enforce that requirement through another branch or helper, or it may fail to enforce it at all. This motivates the paired structural evidence and explicit repair-principle layer used by TRACE. The example demonstrates the source-level repair mechanism; no particular TRACE or baseline prediction for this sample is asserted without an aligned inference record.

2.2 Why high recall requires a pair-sensitive measure

An aggressive warning policy can detect nearly all vulnerable revisions while raising many alerts on repaired code. On a paired benchmark, the possible outputs for a vulnerable/fixed pair are $(1, 0)$, $(1, 1)$, $(0, 0)$, and $(0, 1)$, with $(1, 0)$ denoting the desired outcome. Ordinary F1 treats the two predictions as separate samples and does not reveal whether they are jointly correct. Pair accuracy directly measures the ability to recognize the security-relevant difference. It also exposes an important limitation of function-level evidence: a correctly retrieved repair does not guarantee that a model will recognize a semantically equivalent guard expressed elsewhere in the target.

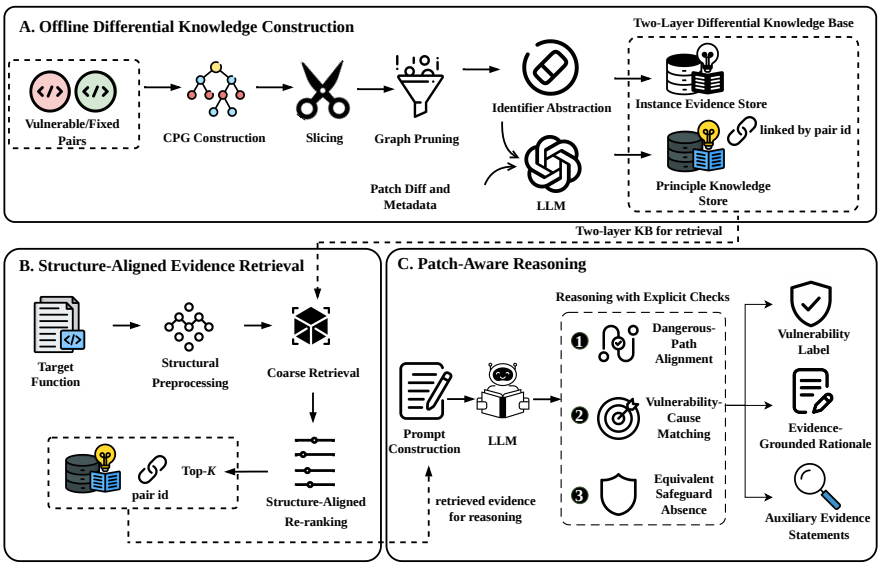


Fig. 1 The TRACE architecture. Offline repair-pair knowledge construction (M1) supplies linked instance and principle evidence to structure-aligned retrieval (M2; top-50 candidate retrieval, top-10 re-ranking). The retrieved evidence supports three explicit checks in patch-aware reasoning (M3).

3 TRACE: Patch-Aware Retrieval-Augmented Reasoning

3.1 Architecture and information flow

TRACE organizes vulnerability triage as a sequence of evidence construction, evidence selection, and evidence-constrained judgment (Fig. 1). The distinction between these operations is essential. Historical patches are used offline to explain what made a known revision vulnerable and what the fix changed; an unseen target function is inspected online without access to its own repair patch. Retrieval connects the two by selecting historical contrasts relevant to the target's structure.

3.2 M1: Offline differential knowledge construction

The historical evidence unit in TRACE is a vulnerable/fixed repair pair rather than an isolated vulnerable function. This choice enables M1 to examine both the original failure context and the security-relevant change introduced by the repair. Given the two revisions, their patch diff, and available CVE/CWE metadata, TRACE uses patch-modified statements as primary slicing seeds because they localize the code changes associated with the reported weakness. However, a security patch may introduce a guard or alter a state transition while leaving the dangerous operation unchanged. Therefore, security-sensitive operations, including memory accesses, allocation and release, synchronization, and error-handling operations, serve as auxiliary seeds. Combining these two sources of context helps preserve the dependency relationships between

the modified safety condition and the operation it protects. This evidence construction does not assume that every modified statement is security-critical or that the resulting slices capture all relevant program behavior.

TRACE constructs function-level code property graphs and applies backward and forward slicing over data and control dependencies (Weiser 1981; Ferrante et al. 1987; Horwitz et al. 1990; Yamaguchi et al. 2014). The former traces the values, guards, and states that reach security-sensitive operations; the latter follows how a changed value or resource state is used. The implementation keeps call sites, argument roles, return checks, and receiver information for external calls as limited call-site context, rather than claiming interprocedural whole-program verification. Graph pruning removes unrelated statements while preserving patch-related and dangerous-operation dependency paths.

To reduce project-specific lexical noise, we normalize identifiers into roles such as buffer, index, length, pointer, resource, lock, status, and object when their usage supports that interpretation. Security-relevant API and operator information is retained. The resulting *instance evidence* contains paired vulnerable/fixed slices, patch-related dependencies, abstracted identifiers, and repair-pair metadata. This structural context matters because a change in the fixed revision matters only when it preserves its relationship to the dangerous operation.

GPT-4o serves as the offline teacher LLM and distills a *principle record* from five inputs: the vulnerable slice, fixed slice, patch diff, dependency paths, and CVE/CWE metadata. The record describes the vulnerability cause, the missing or violated security condition, and the intended repair mechanism. Differential consistency checks require these descriptions to align with behaviors visible in the two revisions; a principle without support in the patch is rejected or regenerated. The principle is therefore not treated as a standalone rule: a stable pair identifier links it to its instance-level evidence.

3.3 M2: Structure-aligned evidence retrieval

For an unseen target, TRACE constructs its code representation and extracts vulnerability-focused context using security-sensitive operations and their dependencies as seeds. *Unlike offline knowledge construction, online query processing cannot use patch-modified lines because the target has no known fixing patch.* The same graph, pruning, and abstraction machinery is reused only where it is applicable to this unpatched input. This separation prevents offline repair annotations from becoming hidden online evidence.

Retrieval first uses UniXcoder embeddings of the structural slices to recall the top $N = 50$ historical instance records (Guo et al. 2022; Karpukhin et al. 2020). The initial stage favors broad coverage of potentially relevant dangerous paths. A subsequent ranker distinguishes candidates that share relevant dependency structure from those that only share common tokens or APIs. Its normalized score combines dense, lexical, AST, and dependency-path similarities:

$$s(x, i) = \alpha s_{\text{dense}}(x, i) + \beta s_{\text{lex}}(x, i) + \gamma s_{\text{ast}}(x, i) + \delta s_{\text{dep}}(x, i), \tag{1}$$

The fixed weights are $\alpha = 0.40$, $\beta = 0.10$, $\gamma = 0.15$, and $\delta = 0.35$. Dense similarity preserves broad semantic coverage, while dependency alignment weights source-to-sink and guard-to-operation relationships involved in a historical failure more heavily. Lexical and AST scores add complementary evidence rather than defining equivalence by themselves. After structural re-ranking, TRACE retains the top $K = 10$ *instance* records. Only then does it fetch their associated principle records through pair-ID linkage. The reasoning model receives selected pairs and their linked principles, not independent lists of generic vulnerability advice.

3.4 M3: Patch-aware reasoning and decision

The reasoning stage evaluates whether a selected historical repair contrast applies to the target. It uses the target slice and dependency paths together with each retrieved vulnerable slice, fixed slice, and principle record. Three checks govern the decision: *dangerous-path alignment* identifies a comparable security-sensitive operation and dependency relation; *vulnerability-cause matching* tests whether the same failure condition is plausible in the target; and *equivalent-safeguard verification* checks for protection that would invalidate the historical vulnerability analogy.

We treat a safeguard as equivalent only when the available evidence supports protection of the same relevant value, object, or resource state before the dangerous operation along the pertinent path, and against the same vulnerability cause. A similarly named check on another value, or a guard after the sink, does not meet this standard. In particular, *failure to recognize a guard is not by itself evidence of vulnerability*: path alignment and cause matching must also hold. These are LLM-assisted judgments over the available function context, not formal proofs of program behavior.

The decision protocol predicts vulnerable when the selected evidence supports both dangerous-path alignment and vulnerability-cause matching and no equivalent safeguard is identified for that matched case. The model produces structured check outcomes and an evidence-grounded rationale linked to a selected repair-pair identifier. The final label is constrained by those checks rather than obtained from unconstrained prose. When the evidence does not support the required path and cause, the corresponding candidate cannot establish a positive decision. The excerpt below summarizes the evidence boundaries imposed on the inference prompt.

Prompt excerpt

Scope: use only the target slice, target dependency paths, and retrieved evidence; do not use external context or the ground-truth label.
Checks: for each retrieved item, verify dangerous-path alignment, vulnerability-cause matching, and equivalent-safeguard presence.
Decision: predict `vulnerable` only if at least one item matches the dangerous path and vulnerability cause, and no equivalent safeguard is found. Weak or superficial matches are set to false.
Output: return JSON only, including prediction, selected pair ID, retrieval scores copied from evidence, concise rationale, and check records.

4 Evaluation Design

4.1 Research questions and study logic

The evaluation distinguishes four claims that are often conflated: whether the detector identifies individual vulnerable functions, whether it recognizes repaired counterparts, which components contribute to those outcomes, and how sensitive the findings are to evidence and model choices. We study them in the following order.

RQ1 (Detection effectiveness). How does TRACE compare with established vulnerability detectors, particularly VulInstruct, on the evaluated functions?

RQ2 (Repair-pair discrimination). To what extent does TRACE correctly distinguish both revisions of a repair pair, and how many false alarms remain on the fixed side?

RQ3 (Component contribution). Which parts of slicing, graph pruning, re-ranking, principle knowledge, fixed-side evidence, and identifier abstraction account for observed performance?

RQ4 (Sensitivity to evidence and inference models). How does TRACE respond to independently constructed repair-knowledge sources and different LLM backbones?

4.2 Datasets, splits, and evidence provenance

We use two complementary evaluation settings because function-level detection and repair-pair discrimination require different observations. PairVul provides 586 held-out vulnerable/fixed repair pairs, comprising 586 vulnerable and 586 repaired functions (Du et al. 2026). Its paired structure enables direct evaluation of whether a detector recognizes both the vulnerable revision and its fixed counterpart. The PairVul training partition supplies 2,317 repair pairs for its knowledge base ($\mathcal{K}_P$).

SecVulEval provides a complementary function-level evaluation on a differently balanced cohort. We use a fixed subset of 301 functions, including 120 vulnerable and 181 non-vulnerable functions, selected through `random_subset.json` (Ahmed et al. 2025). Its knowledge base ($\mathcal{K}_S$) is independently constructed from 2,317 repair pairs in the SecVulEval training data. The two knowledge bases have equal sizes but originate from different benchmark training partitions; they are not interchangeable copies of a shared corpus.

This design supports complementary observations of repair recognition and function-level detection under different dataset conditions. However, independent knowledge-base construction does not, by itself, rule out all cross-source near-duplicates. Pair accuracy is reported only for PairVul, where the vulnerable/fixed pairing is available. We therefore compare cohorts within each evaluation rather than interpreting differences in class balance as evidence of deployment-level performance.

For each test cohort, the knowledge-source experiment compares its matching training knowledge base, the independently constructed knowledge base from the other benchmark, and their union ($\mathcal{K}_P \cup \mathcal{K}_S$). The target functions and inference settings remain fixed, isolating the effect of the historical evidence source. The evaluation protocol excludes direct matches between retrieval evidence and tested code by CVE, fixing commit, and function identity. These exclusions address known identities; they

are not, by themselves, a guarantee against every cross-project or near-duplicate similarity.

4.3 Baselines and controlled settings

We compare TRACE with Code RAG and Vul-RAG (Du et al. 2026), SecVulEval-MA (Ahmed et al. 2025), LLMAO, adapted from test-free fault localization (Yang et al. 2024), VulTrial (Widyasari et al. 2026), and VulInstruct (Zhu et al. 2026). These baselines represent code retrieval, vulnerability-knowledge retrieval, LLM-based fault-localization reasoning, multi-agent assessment, and specification-guided reasoning. VulInstruct is the most relevant high-recall comparator because it also seeks to make security conditions explicit. In the reported main comparison, TRACE and VulInstruct use Qwen2.5-Coder-32B at temperature 0.1 with common test examples and metric definitions, while retaining their respective evidence-generation mechanisms. We do not equate a common backbone with equal retrieval cost, context length, or total compute. Results from the other baseline evaluations should be interpreted with the configuration provenance recorded for each run.

Table 1 Controls for the main TRACE–VulInstruct comparison.

Factor	VulInstruct	TRACE
Evidence mechanism	Security specifications	Retrieved repair pairs and linked principles
Backbone	Qwen2.5-Coder-32B	Qwen2.5-Coder-32B
Temperature	0.1	0.1
Test examples	Same fixed set	Same fixed set
Method preserved	Specification-guided inference	Patch-aware retrieval and checks
Evaluation	Shared labels and scripts	Shared labels and scripts

4.4 Metrics and statistical interpretation

We report accuracy, precision, recall, and F1 with vulnerable functions as positives, together with TP, FN, FP, and TN counts. To measure joint repair recognition on PairVul, exact pair accuracy is defined by

$$\text{PairAcc} = \frac{1}{N}\sum_{j=1}^{N}\mathbb{I}\left[\hat{y}_j^v = 1 \wedge \hat{y}_j^f = 0\right]. \tag{2}$$

Each pair contributes exactly one vulnerable function v and one fixed function f; $(1,0)$ is the fully correct prediction, while $(1,1)$ represents a false alarm on the fixed counterpart. We additionally report the fixed-side false-positive rate and warnings per true positive, $(TP+FP)/TP$, as a count-based triage proxy, not as an observation of developer time or computational latency.

For the five comparisons in Table 3, we align predictions by test-instance identifiers. On PairVul, the vulnerable and repaired functions of each pair form one resampling block and always remain together; SecVulEval is resampled at the function level.

We compute pointwise 95% percentile confidence intervals for the metric differences from 10,000 paired bootstrap replicates, applying identical sampled indices to the two methods. For F1, we use 100,000 paired random permutations that exchange the two methods' predictions within each block, evaluate two-sided absolute metric differences, and apply a plus-one correction to the permutation p-value. For PairAcc, we apply a two-sided exact McNemar test to pair-level joint-correctness indicators. We set the bootstrap and permutation random seeds to 20261019 and 20261020, respectively.

Both unadjusted p-values and Holm-adjusted p-values across the five pre-specified Table 3 contrasts are reported. We use a family-wise significance threshold of 0.05 for adjusted inference. Because the intervals are pointwise rather than simultaneous or multiplicity-adjusted, exclusion of zero in an individual interval should not be conflated with family-wise significance. The comparisons are re-expressible from aligned prediction records, which are needed for independent replication; aggregate confusion matrices alone cannot recover their dependence structure.

4.5 Implementation configuration

TRACE uses Joern to construct code property graphs, following the representation introduced by Yamaguchi et al., and UniXcoder to obtain dense code representations (Yamaguchi et al. 2014; Guo et al. 2022). Candidate retrieval retains $N = 50$ items, structure-aware re-ranking selects $K = 10$, and the four ranking weights are 0.40, 0.10, 0.15, and 0.35 for dense, lexical, AST, and dependency evidence. GPT-4o is used for offline repair-principle distillation, whereas the main online inference backbone is Qwen2.5-Coder-32B with temperature 0.1 (Hurst et al. 2024; Hui et al. 2024). The offline teacher is not invoked for each test function. The six ablations remove one component at a time from the same three-module design. This study does not implement a new safeguard-path prover, interprocedural analyzer, or multi-agent system.

5 Experimental Results

5.1 RQ1: Vulnerability detection effectiveness

Main finding. TRACE retains high recall while improving precision and F1 over VulInstruct on the evaluated cohorts; the PairVul F1 margin is modest and is not significant in the reported nominal test. Table 2 compares the methods on both benchmarks. TRACE attains 93.52% recall and 75.48% F1 on PairVul, versus VulInstruct's 95.05% recall and 73.58% F1. The nine additional vulnerable functions detected by VulInstruct come with 53 more false alarms on fixed functions (371 versus 318). TRACE therefore exchanges a small amount of vulnerable-side recall for fewer fixed-side warnings and higher precision (63.28% versus 60.02%).

On SecVulEval, the pattern differs: TRACE identifies 111 of 120 vulnerable functions, compared with 94 for VulInstruct, while false positives remain similar (84 versus 85). The resulting recall and F1 are 92.50% and 70.48% for TRACE, compared with 78.33% and 62.88% for VulInstruct. This is evidence of improved positive coverage in that particular 301-function cohort rather than proof of broader superiority across all SecVulEval samples.

Table 2 Function-level vulnerability detection results (%) on the fixed evaluation cohorts. Baseline protocols retain their native evidence mechanisms.

Method	PairVul				SecVulEval			
	Acc.	Prec.	Rec.	F1	Acc.	Prec.	Rec.	F1
Vul-RAG	59.47	64.57	41.98	50.88	60.80	51.02	41.67	45.87
SecVulEval-MA	49.40	49.23	38.40	43.14	48.50	35.29	35.00	35.15
Code RAG	53.67	**65.47**	15.53	25.10	60.80	52.78	15.83	24.36
LLMAO	51.79	57.55	13.65	22.07	61.13	55.17	13.33	21.48
VulTrial	60.84	59.03	70.82	64.39	61.79	51.57	68.33	58.78
VulInstruct	65.87	60.02	**95.05**	73.58	63.12	52.51	78.33	62.88
TRACE	**69.62**	**63.28**	93.52	**75.48**	**69.10**	**56.92**	**92.50**	**70.48**

The confusion counts in Table 4 make the precision–recall trade-off inspectable. PairVul uses a balanced vulnerable/fixed cohort; SecVulEval uses 120 positives and 181 negatives. We therefore interpret the scores within each benchmark and avoid inferring real-world alert rates from these class proportions.

Table 3 provides uncertainty estimates for the observed performance differences. Relative to Vul-RAG on PairVul and SecVulEval-MA on SecVulEval, TRACE achieves F1 improvements of 24.60 and 35.33 percentage points, respectively; both remain significant after Holm adjustment (adjusted $p < 0.005$).

The comparison with VulInstruct requires a more qualified interpretation. On PairVul, TRACE improves F1 by 1.90 percentage points, but the pointwise 95% confidence interval $[-0.6, 4.4]$ includes zero and the raw p-value is 0.14. This result does not establish a statistically significant advantage. On SecVulEval, the observed F1 improvement is 7.60 percentage points, with a pointwise 95% confidence interval of $[1.0, 14.1]$ and raw $p = 0.04$. However, the Holm-adjusted p-value is 0.09, exceeding the family-wise significance threshold of 0.05.

Consequently, the results support clear improvements over the weaker reference comparators, while the observed advantages over VulInstruct do not establish superiority under the prespecified family-wise testing procedure. The lack of statistical significance should not be interpreted as evidence that the methods perform equivalently.

Table 3 Paired statistical comparisons of TRACE against reference detectors. Differences and pointwise percentile-bootstrap intervals are in percentage points. Raw p-values are from paired F1 permutation tests or exact McNemar tests (PairAcc); Holm adjustment controls the family-wise error rate across the five comparisons. For raw values reported as < 0.001, the displayed adjusted bounds are conservative.

Dataset	Metric	Comparator	Δ (pp)	95% CI (pp)	Raw p	Holm p
PairVul	F1	Vul-RAG	+24.60	[20.0, 29.2]	< 0.001	< 0.005
PairVul	F1	VulInstruct	+1.90	[−0.6, 4.4]	0.14	0.14
SecVulEval	F1	SecVulEval-MA	+35.33	[27.2, 43.6]	< 0.001	< 0.005
SecVulEval	F1	VulInstruct	+7.60	[1.0, 14.1]	0.04	0.09
PairVul	PairAcc	VulInstruct	+5.29	[0.7, 9.9]	0.03	0.09

Table 4 Function-level confusion counts for the main comparison and the additional PairVul repair-pair baseline. PairVul totals agree with the joint prediction states in Table 5.

Dataset	Method	TP	FN	FP	TN
PairVul	Vul-RAG	246	340	135	451
PairVul	VulInstruct	557	29	371	215
PairVul	TRACE	548	38	318	268
SecVulEval	VulInstruct	94	26	85	96
SecVulEval	TRACE	111	9	84	97

5.2 RQ2: Discriminating vulnerable and repaired revisions

Main finding. TRACE correctly distinguishes both revisions in 236 of 586 PairVul pairs (40.27%), compared with 205 pairs (34.98%) for VulInstruct and 170 pairs (29.01%) for Vul-RAG (Table 5). These differences are +5.29 and +11.26 percentage points, respectively. Against VulInstruct, the PairAcc difference has a pointwise 95% CI of $[0.7, 9.9]$ pp and raw $p = 0.03$; the Holm-adjusted $p = 0.09$ does not meet the family-wise 0.05 threshold. We do not report a PairAcc significance test against Vul-RAG.

The joint outcomes expose distinct error patterns. TRACE predicts both versions vulnerable in 312 pairs, versus 352 for VulInstruct and 76 for Vul-RAG. Vul-RAG instead labels both versions non-vulnerable in 281 pairs, consistent with its much lower vulnerable-side coverage (246 true positives versus TRACE's 548). Although Vul-RAG raises fewer fixed-side false alarms than TRACE (135 versus 318), it misses far more vulnerable functions (340 versus 38), lowering its joint pair accuracy. Compared with VulInstruct, TRACE reduces repaired-side false alarms by 53 (318 versus 371), yet still labels 53.24% of repair pairs $(1, 1)$. These differences show why function-level F1 alone cannot infer correct repair-pair discrimination; the RQ3 ablations examine the role of fixed-side evidence.

Table 5 Joint prediction states and pair accuracy on PairVul (586 repair pairs).

Method	Joint prediction counts				PairAcc (%)
	$(1, 0)$ Correct pair	$(1, 1)$ Both flagged	$(0, 0)$ Neither flagged	$(0, 1)$ Reversed	
Vul-RAG	170	76	281	59	29.01
VulInstruct	205	352	10	19	34.98
TRACE	**236**	312	32	6	**40.27**

Notes. States are (vulnerable-version prediction, fixed-version prediction); 1 = vulnerable, 0 = non-vulnerable. Bold indicates the highest correct-pair count and PairAcc. Vul-RAG state counts are reconstructed using its confirmed 170 correct pairs and function-level totals consistent with Table 2.

5.3 RQ3: What the components contribute

Main finding. The principle layer and fixed-side evidence make the largest difference to the observed F1, while structural processing supports the relevance of retrieved examples. Table 6 removes each of the six original components. Without principle-level knowledge, PairVul F1 declines from 75.48% to 69.80% (-5.68 pp) and SecVulEval F1 from 70.48% to 63.98% (-6.50 pp). A plain collection of structurally related code, without an explicit description of the violated condition and repair mechanism, therefore provides weaker evidence for the measured inference workflow.

Table 6 Ablation results of TRACE on PairVul and SecVulEval (%).

Variant	PairVul			SecVulEval		
	Recall	F1	ΔF1	Recall	F1	ΔF1
TRACE (full)	93.52	**75.48**	–	92.50	70.48	–
w/o vulnerability-focused slicing	89.08	71.60	-3.88	87.50	66.04	-4.44
w/o graph pruning	91.81	73.65	-1.83	90.83	68.55	-1.93
w/o structure-aligned re-ranking	90.78	72.68	-2.80	90.00	67.50	-2.98
w/o principle-level knowledge	88.57	69.80	-5.68	85.83	63.98	-6.50
w/o fixed-side evidence	**94.20**	71.32	-4.16	**93.33**	65.50	-4.98
w/o identifier abstraction	92.49	74.71	-0.77	91.67	**70.51**	+0.03

Note. w/o denotes without. ΔF1 is the F1 difference from full TRACE, in percentage points. Bold indicates the highest Recall and F1 in each dataset.

Removing fixed-side evidence produces a diagnostic trade-off. PairVul recall increases from 93.52% to 94.20%, but F1 decreases to 71.32%; SecVulEval recall similarly rises to 93.33% while F1 falls to 65.50%. This is consistent with repaired-side evidence primarily constraining over-positive predictions rather than maximizing the number of warnings. It aligns with RQ2, where the greatest practical improvement is fewer false alarms on fixed revisions.

The structural stages contribute in more moderate, consistent ways. Removing vulnerability-focused slicing reduces PairVul F1 by 3.88 pp; removing structural re-ranking reduces it by 2.80 pp; and removing graph pruning reduces it by 1.83 pp. These losses support preserving security-relevant dependency context before the LLM evaluates the applicability of a patch. Identifier abstraction is less robust: omitting it lowers PairVul F1 by 0.77 pp but raises SecVulEval F1 by 0.03 pp (70.51% versus 70.48%). The latter difference is too small to support a general claim of improvement in either direction. Pair accuracy is not reported for the ablations because their pair-level outcomes are unavailable.

5.4 RQ4: Sensitivity to knowledge sources and LLM backbones

Main finding. Repair evidence from either training source remains useful, but matching the test distribution yields stronger results; the relative performance of LLM backbones also varies by benchmark. Table 7 compares $\mathcal{K}_P$, built from PairVul training repair pairs, $\mathcal{K}_S$, independently built from SecVulEval training data, and their

merged collection. Each test cohort is unchanged across the three configurations. On PairVul, replacing $\mathcal{K}_P$ with $\mathcal{K}_S$ reduces F1 from 75.48% to 72.31%; on SecVulEval, replacing $\mathcal{K}_S$ with $\mathcal{K}_P$ reduces F1 from 70.48% to 66.88%. Combining both sources increases F1 modestly to 76.07% and 71.34%, respectively. Thus, repair principles learned from the other training source remain usable, yet training-source alignment matters. The modest gains from merging also indicate that adding repair evidence does not automatically make the retrieved cases more relevant.

Table 7 Knowledge-source transfer between two independently constructed 2,317-pair training knowledge bases (%). The merged setting combines PairVul and SecVulEval evidence.

Test set	Knowledge-base source	Precision	Recall	F1
PairVul	PairVul training pairs	63.28	93.52	75.48
PairVul	SecVulEval-derived pairs	60.23	90.44	72.31
PairVul	Merged knowledge base	63.95	93.86	**76.07**
SecVulEval	SecVulEval-derived pairs	56.92	92.50	70.48
SecVulEval	PairVul training pairs	53.81	88.33	66.88
SecVulEval	Merged knowledge base	57.73	93.33	**71.34**

Backbone variation in Table 8 shows a related dependency. The evaluated models are GPT-4o, Claude 3.5 Sonnet, Qwen2.5-Coder-32B, and DeepSeek-Coder-33B (Hurst et al. 2024; Anthropic 2024; Hui et al. 2024; Guo et al. 2024a). Qwen2.5-Coder-32B achieves the highest PairVul F1 (75.48%) and recall (93.52%) among the tested models. On SecVulEval, however, Claude 3.5 Sonnet reaches the highest F1 (72.10%) and recall (95.83%). We keep the Qwen backbone as the common model for the principal baseline comparison; the backbone study shows that the repair-aware pipeline is not tied to a single model's performance profile.

Table 8 Sensitivity of TRACE to the LLM inference backbone (%).

	PairVul		SecVulEval	
Backbone	Recall	F1	Recall	F1
GPT-4o	83.62	73.57	95.00	71.47
Claude 3.5 Sonnet	87.03	74.67	**95.83**	**72.10**
Qwen2.5-Coder-32B	**93.52**	**75.48**	92.50	70.48
DeepSeek-Coder-33B	82.76	72.07	90.00	68.35

5.5 Operational and failure analysis

Beyond the four research questions, we examine the observable warning burden and the failure types that delimit the method's use in developer triage. These analyses serve a different purpose from RQ4: they describe what an analyst may receive and

where evidence-grounded judgments break down, rather than testing invariance to knowledge source or backbone.

Warning burden

Warning counts provide a limited operational view of these results. Table 9 reports $(TP + FP)/TP$: TRACE needs 1.58 warnings per true positive on PairVul compared with 1.67 for VulInstruct. On SecVulEval, the ratios are 1.76 and 1.90, respectively; however, TRACE issues *more* warnings in absolute terms on that cohort (195 versus 179), because it detects substantially more positives. These quantities describe alert volume on the benchmark, not measured review time, latency, or inference cost.

Table 9 Warning counts per correctly identified vulnerable function in the main comparison.

Dataset	Method	TP	Warnings	Warnings/TP
PairVul	VulInstruct	557	928	1.67
PairVul	TRACE	548	866	1.58
SecVulEval	VulInstruct	94	179	1.90
SecVulEval	TRACE	111	195	1.76

Failure mechanisms

Table 10 separates missed vulnerabilities from spurious warnings. The false-negative categories cover all 38 PairVul and nine SecVulEval FN cases, whereas false-positive categories refer to audited subsets of 50/318 and 30/84 FPs, respectively. Among PairVul misses, missing repair evidence (14), incomplete slices (13), and retrieval mismatches (11) identify three different points at which useful conditions may be unavailable to M3. Within the audited FP subsets, safeguard-equivalence failures are the most frequent assigned category (22/50 and 12/30), followed by over-generalized repair analogies and ambiguous semantics. These cases help locate potential failure mechanisms, but we should not project the audited FP fractions onto the entire benchmark without a representative sampling design, and category labels alone do not identify a unique cause for each error.

6 Discussion

6.1 The mechanism-level contribution of repair evidence

TRACE provides a mechanism to turn a patch-linked example into a checkable account of a target function, rather than using the patch as an isolated vulnerability exemplar (Du et al. 2026; Zhu et al. 2026). A structurally similar vulnerable function suggests a possible danger; the associated fixed revision specifies which condition must change to avoid that danger. The two-layer knowledge base keeps these roles connected, and structure-aware retrieval selects the contrasts on which the final dangerous-path,

Table 10 Error categories in the current TRACE evaluation. All false negatives (38 on PairVul and 9 on SecVulEval) are categorized; false-positive categories refer to audited subsets of 50 and 30 cases, respectively.

Direction	Primary cause	PairVul	SecVulEval
FN	Missing repair evidence	14	4
FN	Incomplete structural slice	13	3
FN	Retrieval mismatch	11	2
FP	Safeguard equivalence failure	22	12
FP	Over-generalized repair analogy	18	11
FP	Ambiguous vulnerability semantics	10	7

cause, and equivalent-safeguard checks operate. Compared with retrieval of code similarity alone or the presentation of general security obligations, this design makes a concrete, path-related repair condition part of the evidence supplied for each decision. It remains an LLM judgment over represented context, not a formal proof of safety.

The ablations are consistent with the intended roles of the evidence: removing principle-level knowledge reduces F1 on both cohorts, while removing fixed-side evidence raises recall slightly but lowers F1 substantially. The vulnerability and repair sides therefore supply complementary information. The benefit is incomplete, however. TRACE still predicts both revisions vulnerable for 312/586 PairVul pairs, showing that equivalent-safeguard recognition remains a bottleneck even when the repair contrast is available. The Linux KVM repair illustrated in Listing 1 exemplifies how a safety guard changes the conditions under which a dangerous operation can execute. This example illustrates the source-level repair mechanism rather than an independently observed TRACE inference trace.

6.2 Interpreting gains against strong baselines

The comparison with VulInstruct supports a qualified rather than universal advantage. On PairVul, TRACE reduces fixed-side false alarms by 53 and raises exact pair accuracy from 34.98% to 40.27%, but VulInstruct retains slightly higher vulnerable-side recall. The +1.90 pp F1 difference has a raw p-value of 0.14. On the 301-function SecVulEval subset, the +7.60 pp F1 difference has a raw p-value of 0.04; the PairVul PairAcc contrast has a raw p-value of 0.03. The latter two results do not remain significant after Holm correction (adjusted $p = 0.09$ for both). Thus, the strongest supported interpretation is a promising repair-aware decision behavior in these evaluated cohorts, not conclusive superiority to VulInstruct across contexts.

The independently built 2,317-pair training knowledge bases provide a second qualification. Using the other benchmark's evidence source reduces F1 on each evaluation cohort, while merging the sources yields only moderate gains. Knowledge coverage and match to the target distribution both matter; the result does not establish temporal or project-level out-of-distribution generalization. Similarly, backbone sensitivity cautions against treating a result from one model as an invariant property of the pipeline.

6.3 Implications for developer-facing triage

For developers, a useful warning should be inspectable as well as sensitive, reflecting longstanding concerns in security-oriented static analysis and deployed analysis tools (Chess and McGraw 2004; Sadowski et al. 2018). TRACE can associate an alert with a retrieved repair pair, the relevant dependency slice, and a statement of the condition believed to remain unsatisfied. This makes the analyst's immediate task more focused: inspect whether the target truly follows the alleged path and whether an alternative guard blocks it. It also highlights the cost of mistakes. An unrecognized equivalent safeguard leads to review of repaired code, whereas an incomplete slice or missing historical mechanism may conceal a genuine vulnerability.

The measured warning ratios quantify alert volume, not analyst labor. TRACE produces 1.58 warnings per true positive on PairVul versus 1.67 for VulInstruct; on SecVulEval the corresponding figures are 1.76 versus 1.90, although TRACE produces more total warnings there because it identifies more true positives. We measured no elapsed inspection time, end-to-end latency, or token expenditure. Consequently, the operational claim is limited to evidence organization, fixed-side behavior, and observable warning counts; reductions in human effort or computational cost remain open questions.

7 Threats to Validity

Internal validity. The extracted function-local graph may omit a relevant caller condition or helper-level safeguard. Patch diffs can mix security changes with refactoring, and teacher-generated principles can overgeneralize the intended repair. Similar code may also remain across nominally separated datasets, a recognized concern in vulnerability-dataset evaluation (Chakraborty et al. 2024; Li et al. 2026). The stated CVE, commit, and function-identity exclusions reduce direct leakage, but without project- and near-duplicate-level audits they cannot establish complete independence. Variations between baseline implementations and common-backbone adaptations can influence the fairness of workflow comparisons.

Construct validity. A labeled vulnerable function is not necessarily demonstrably exploitable, and a labeled fixed function need not be free of every other weakness. Pair accuracy measures correct joint labels for a known fix pair; it is not a theorem that the repair is semantically equivalent in arbitrary contexts. Likewise, warnings per true positive is an alert-count proxy, not an observed developer-effort or computational-cost metric. The error audit uses all categorized FNs and only sampled FPs, so its category frequencies have different denominators.

External validity. The evaluated code is C/C++, the PairVul cohort is balanced by construction, and the SecVulEval evaluation contains 301 selected functions (Du et al. 2026; Ahmed et al. 2025). Unknown CWE families, unseen project idioms, missing call graphs, other programming languages, and deployment prevalence may change precision and recall; recent evaluations likewise document sensitivity to data sources and distribution shifts (Chakraborty et al. 2024; Li et al. 2026). The independently sourced PairVul and SecVulEval training knowledge bases support a comparison of

evidence-source sensitivity, but this experiment does not establish full temporal or cross-project robustness.

Statistical and reproducibility validity. We account for dependent PairVul observations by resampling/permuting intact pairs, and for five planned contrasts by reporting both raw and Holm-adjusted p-values. The 95% bootstrap intervals are pointwise, not multiplicity-adjusted; the two VulInstruct gains with raw $p < 0.05$ do not remain significant after Holm correction. PairAcc inference depends on the cross-method joint-correctness indicators; neither confusion matrices nor separate pair-state marginals suffice. Although the resampling plan and seeds are specified, independent verification still requires the aligned sample-level predictions, scripts, and audit trails in the artifact release.

8 Related Work

8.1 Learning-based vulnerability detection

Learning-based detectors represent vulnerabilities as token sequences, vulnerability-focused slices, data-flow or dependence graphs, and transformer encoders. VulDeePecker and SySeVR extract program slices; Devign and ReGVD use graph-based program representations; LineVul applies a transformer to line-level prediction (Li et al. 2018, 2021b; Zhou et al. 2019; Nguyen et al. 2022; Fu and Tantithamthavorn 2022). Fine-grained interpretation work further examines how a detector can localize code associated with a prediction (Li et al. 2021a). These approaches establish useful source-level representations, but a high-risk operation or learned pattern may remain in a repaired function even when a guard changes the condition under which that operation is reached. TRACE therefore treats structural similarity as a retrieval signal, not by itself as evidence that a target remains vulnerable.

Evaluation realism is consequential: performance on familiar datasets need not transfer to independent samples or broader CWE coverage (Chakraborty et al. 2024; Li et al. 2026). This motivates interpreting results within the stated datasets and splits rather than treating function-level scores as evidence of generalization.

8.2 LLM-based vulnerability analysis

LLM-based vulnerability analysis has been studied through direct code prompting, benchmark evaluation, static-analysis integration, explanation-oriented workflows, and analyses of model reliance on code metrics (Ullah et al. 2024; Ding et al. 2025; Li et al. 2025; Cao et al. 2024; Mao et al. 2025; Weissberg et al. 2026). These studies show why a plausible natural-language explanation should not be equated with a verified vulnerability condition: performance depends on the evidence and evaluation setting, and out-of-distribution behavior remains an active concern (Guo et al. 2024b; Li et al. 2026). TRACE is likewise LLM-assisted, but constrains its decision with retrieved vulnerable/fixed evidence and an explicit test for an equivalent safeguard; this design does not make the resulting judgment a formal proof.

8.3 Retrieval-augmented and specification-guided detection

Retrieval-augmented generation provides a general framework for supplying external evidence to language models (Lewis et al. 2020). Vul-RAG retrieves multidimensional knowledge from vulnerable code and fixes and uses both vulnerability causes and fixing solutions in its reasoning; VulInstruct derives general and domain-specific security specifications from vulnerability-related code and patches (Du et al. 2026; Zhu et al. 2026). TRACE's distinction is narrower: it preserves individual vulnerable/fixed structural instances alongside linked repair principles and uses dependency-aware alignment to retrieve those instances before target-side reasoning. This characterizes TRACE's evidence organization without implying that prior methods lack repair-aware reasoning or checks for relevant safety conditions.

8.4 Patch analysis and dependency-aware evidence

Patch-linked vulnerability datasets make fixing changes available as evidence, and automated vulnerability-repair studies examine how models can use such examples to generate fixes (Bhandari et al. 2021; Fan et al. 2020; Fu et al. 2022; Pearce et al. 2023). Program slicing, dependence graphs, and code property graphs provide established ways to connect changed statements to the values and paths they constrain (Weiser 1981; Ferrante et al. 1987; Horwitz et al. 1990; Yamaguchi et al. 2014). These lines of work supply ingredients for repair-grounded analysis, but do not by themselves define TRACE's linked instance/principle retrieval and equivalent-safeguard decision. TRACE uses these techniques to compare a target against both revisions of historical repair evidence; its pair-level evaluation tests whether the known vulnerable and fixed revisions are distinguished, not whether a patch proves safety in arbitrary contexts.

9 Conclusion

TRACE treats a historical vulnerability and its repair as a linked explanation of when a dangerous path is, or is not, vulnerable. Its two-layer evidence base, structural retrieval, and explicit safeguard-aware reasoning operationalize that contrast for function-level triage. The evaluated results show high recall, fewer repaired-side false alarms than VulInstruct on PairVul, and sensitivity to the quality of both repair knowledge and inference context; they do not establish uniform statistically significant gains over the strongest baseline. The main remaining challenge is reliably recognizing equivalent protections when the retrieved repair and the target express the same safety condition differently.

10 Data and Artifact Availability

The project artifact is listed at https://anonymous.4open.science/r/TRACE-B9E6/. PairVul and SecVulEval are drawn from their publicly released benchmark resources. The aggregate results reported here cover main classification, repair-pair states, component ablation, knowledge transfer, backbone sensitivity, and diagnostic error categories. Reproducing Table 3 requires the aligned prediction records and the specified paired-bootstrap, permutation, and exact McNemar procedures. Table 10 additionally requires

the categorized false-negative cases and sampled false-positive audit records. The public CVE-2022-2153 illustration follows the upstream KVM patch and is not reported as a TRACE inference trace.

Statements and Declarations

Competing interests. *The authors declare that they have no competing interests.*
Funding. *This work was partially supported by the National Natural Science Foundation of China (Grant No. 61872263) and the Postgraduate Research & Practice Innovation Program of Jiangsu Province (Grant No. SJCX25_2000).*
Author contributions. ***Dihao Zhang:*** *Data curation, Software, Validation, Conceptualization, Methodology, Writing – review & editing.* ***Xiaolin Ju:*** *Conceptualization, Methodology, Writing – review & editing, Supervision.* ***Zhan Gao:*** *Supervision, Writing – review & editing.* ***Xiang Chen:*** *Conceptualization, Methodology, Supervision.* ***Lina Gong:*** *Conceptualization, Methodology, Writing – review & editing.*

References

Ahmed MBU, Harzevili NS, Shin J, et al (2025) SecVulEval: Benchmarking llms for real-world c/c++ vulnerability detection. arXiv preprint arXiv:250519828

Anthropic (2024) Claude 3.5 sonnet model card addendum. https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf

Bhandari G, Naseer A, Moonen L (2021) Cvefixes: automated collection of vulnerabilities and their fixes from open-source software. In: Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering, pp 30–39

Cao D, Liao Y, Shang X (2024) Realvul: Can we detect vulnerabilities in web applications with llm? In: Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pp 8268–8282

Chakraborty P, Arumugam KK, Alfadel M, et al (2024) Revisiting the performance of deep learning-based vulnerability detection on realistic datasets. IEEE Transactions on Software Engineering 50(8):2163–2177

Chess B, McGraw G (2004) Static analysis for security. IEEE security & privacy 2(6):76–79

Ding Y, Fu Y, Ibrahim O, et al (2025) Vulnerability detection with code language models: How far are we? In: 2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE), IEEE, pp 1729–1741

Du X, Zheng G, Wang K, et al (2026) Vul-RAG: Enhancing llm-based vulnerability detection via knowledge-level rag. ACM Transactions on Software Engineering and

Methodology 35(10):1–25. https://doi.org/10.1145/3797277

Fan J, Li Y, Wang S, et al (2020) Ac/c++ code vulnerability dataset with code changes and cve summaries. In: Proceedings of the 17th international conference on mining software repositories, pp 508–512

Ferrante J, Ottenstein KJ, Warren JD (1987) The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS) 9(3):319–349

Fu M, Tantithamthavorn C (2022) Linevul: A transformer-based line-level vulnerability prediction. In: Proceedings of the 19th international conference on mining software repositories, pp 608–620

Fu M, Tantithamthavorn C, Le T, et al (2022) Vulrepair: a t5-based automated software vulnerability repair. In: Proceedings of the 30th ACM joint european software engineering conference and symposium on the foundations of software engineering, pp 935–947

Guo D, Lu S, Duan N, et al (2022) Unixcoder: Unified cross-modal pre-training for code representation. In: Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pp 7212–7225

Guo D, Zhu Q, Yang D, et al (2024a) DeepSeek-Coder: when the large language model meets programming–the rise of code intelligence. arXiv preprint arXiv:240114196

Guo Y, Patsakis C, Hu Q, et al (2024b) Outside the comfort zone: Analysing llm capabilities in software vulnerability detection. In: European symposium on research in computer security, Springer, pp 271–289

Horwitz S, Reps T, Binkley D (1990) Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems (TOPLAS) 12(1):26–60

Hui B, Yang J, Cui Z, et al (2024) Qwen2.5-coder technical report. arXiv preprint arXiv:240912186

Hurst A, Lerer A, Goucher AP, et al (2024) Gpt-4o system card. arXiv preprint arXiv:241021276

Karpukhin V, Oguz B, Min S, et al (2020) Dense passage retrieval for open-domain question answering. In: Proceedings of the 2020 conference on empirical methods in natural language processing (EMNLP), pp 6769–6781

Kuznetsov V (2022a) KVM: x86: Avoid theoretical NULL pointer dereference in kvm_irq_delivery_to_apic_fast(). Linux kernel commit 00b5f37189d24ac3ed46cb7f11742094778c46ce, URL https://github.com/torvalds/linux/commit/00b5f37189d24ac3ed46cb7f11742094778c46ce

Kuznetsov V (2022b) [PATCH 0/3] KVM: x86: Prevent NULL pointer dereference in kvm_irq_delivery_to_apic_fast(). Linux Kernel Mailing List, URL https://lkml.iu.edu/2203.3/01450.html

Lewis P, Perez E, Piktus A, et al (2020) Retrieval-augmented generation for knowledge-intensive nlp tasks. Advances in neural information processing systems 33:9459–9474

Li Y, Wang S, Nguyen TN (2021a) Vulnerability detection with fine-grained interpretations. In: Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering, pp 292–303

Li Y, Bui NT, Zhang T, et al (2026) Out of distribution, out of luck: How well can llms trained on vulnerability datasets detect top 25 cwe weaknesses? In: Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering, pp 2096–2108

Li Z, Zou D, Xu S, et al (2018) VulDeePecker: A deep learning-based system for vulnerability detection. In: Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS). Internet Society, https://doi.org/10.14722/ndss.2018.23158, URL https://doi.org/10.14722/ndss.2018.23158

Li Z, Zou D, Xu S, et al (2021b) Sysevr: A framework for using deep learning to detect software vulnerabilities. IEEE Transactions on Dependable and Secure Computing 19(4):2244–2258

Li Z, Dutta S, Naik M (2025) Iris: Llm-assisted static analysis for detecting security vulnerabilities. In: International conference on learning representations, pp 35735–35758

Mao Q, Li Z, Hu X, et al (2025) Towards explainable vulnerability detection with large language models. IEEE Transactions on Software Engineering

National Institute of Standards and Technology (2022) CVE-2022-2153 Detail. National Vulnerability Database, URL https://nvd.nist.gov/vuln/detail/CVE-2022-2153

Nguyen VA, Nguyen DQ, Nguyen V, et al (2022) Regvd: Revisiting graph neural networks for vulnerability detection. In: Proceedings of the ACM/IEEE 44th international conference on software engineering: Companion proceedings, pp 178–182

Pearce H, Tan B, Ahmad B, et al (2023) Examining zero-shot vulnerability repair with large language models. In: 2023 IEEE symposium on security and privacy (SP), IEEE, pp 2339–2356

Sadowski C, Aftandilian E, Eagle A, et al (2018) Lessons from building static analysis tools at google. Communications of the ACM 61(4):58–66

Ullah S, Han M, Pujar S, et al (2024) Llms cannot reliably identify and reason about security vulnerabilities (yet?): A comprehensive evaluation, framework, and benchmarks. In: 2024 IEEE symposium on security and privacy (SP), IEEE, pp 862–880

Weiser M (1981) Program slicing. In: Proceedings of the 5th International Conference on Software Engineering, pp 439–449

Weissberg F, Pirch L, Imgrund E, et al (2026) Llm-based vulnerability discovery through the lens of code metrics. In: Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering, pp 2019–2031

Widyasari R, Weyssow M, Irsan IC, et al (2026) Let the trial begin: A mock-court approach to vulnerability detection using llm-based agents. In: Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering, pp 1030–1042

Yamaguchi F, Golde N, Arp D, et al (2014) Modeling and discovering vulnerabilities with code property graphs. In: 2014 IEEE symposium on security and privacy, IEEE, pp 590–604

Yang AZ, Le Goues C, Martins R, et al (2024) Large language models for test-free fault localization. In: Proceedings of the 46th IEEE/ACM international conference on software engineering, pp 1–12

Zhou Y, Liu S, Siow J, et al (2019) Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. Advances in neural information processing systems 32

Zhu H, Li J, Gao C, et al (2026) VulInstruct: Teaching llms root-cause reasoning for vulnerability detection via security specifications. Proceedings of the ACM on Software Engineering 3(FSE):2813–2834. https://doi.org/10.1145/3808132