NDSS SYMPOSIUM 2026

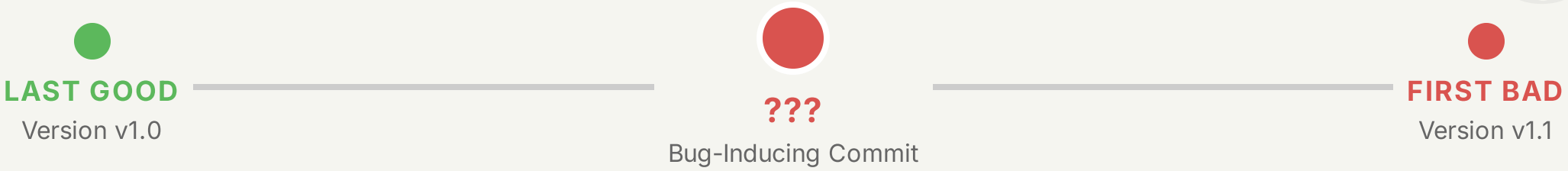
LLMBisect

Breaking Barriers in Bug Bisection with A Comparative Analysis Pipeline

Zheng Zhang[*], **Haonan Li**[*], **Xingyu Li**[*], **Hang Zhang**[†], **Zhiyun Qian**[*]

[*] University of California, Riverside [†] Indiana University Bloomington

Motivation: The Bug Bisection Problem



The Objective

Identify the specific commit that introduced a regression (Bug-Inducing Commit or BIC).

Crucial for understanding the root cause, assigning accountability, and crafting a correct fix.

The Barrier

Scale & Complexity: Large projects like the Linux Kernel have thousands of commits between versions.

> **Manual Bisection** is labor-intensive.
> **Automated Tests** are often flaky or unavailable for every historical commit.

Why Traditional Methods Fail

EXAMPLE: ADD-ONLY PATCH (CVE-2023-1073)

```
struct key *key = ...;
+ if (key->type != &key_type_user) {
+ return -EINVAL;
+ }
down_read(&key->sem);
```

The fix only **adds** a check. No lines are deleted.

SZZ (Blame-Based)

Relies on `git blame` of deleted lines. Fails on **add-only patches** where no lines are removed.

VoFinder (Clone-Based)

Compares function hashes. Fails if the BIC is in a **different function** or if unrelated edits change the hash (33% accuracy).

SymBisect (Symbolic)

Uses symbolic execution. Suffers from **path explosion** and scalability issues.

The LLM Opportunity & Challenges

Why LLMs?

Semantic Comprehension

Analyzes both code logic and natural language commit messages, extracting hints invisible to syntax-based tools.

Logical Reasoning

Moves beyond pattern matching to understand *why* a change might introduce a vulnerability.

Key Hurdles

False Positives

LLMs can be "eager to please," aggressively labeling innocent commits as buggy.

Scale & Cost

Processing thousands of commits naively is prohibitively expensive and slow.

Solution: A Multi-Stage Pipeline (Coarse-to-Fine Filtering)

LLMBisect Design: Multi-Stage Filtering Pipeline

> Design Philosophy: Exploiting LLM's comparative reasoning strength to progressively narrow the search space from coarse to fine.

COARSE-GRAINED

01 Candidate Generation

Extract candidate BICs at scale using lightweight heuristics.

- → Analyze code changes & keywords
- → Inexpensive & efficient
- → Substantially narrows search space

FINE-GRAINED

02 Comparative Selection

Apply LLM in a multi-round, comparative fashion.

- → Evaluate candidates side-by-side
- → Relative assessment
- → Markedly improves accuracy

CONSISTENCY

03 Majority Voting

Incorporate voting at selected key decision points.

- → Mitigate self-consistency issues
- → Avoid performance overhead
- → Ensure reliable results

Candidate Generation - Three Complementary Methods

Combine multiple strategies to maximize candidate coverage while minimizing noise

Critical-Line-Based Selection

Moves from function-level to line-level granularity. Uses LLM to identify the most vulnerability-relevant changed lines and tracks only commits touching them.

KEY BENEFIT

Reduces candidates by 81% on average; includes global

Function-Based Selection

The baseline method that includes all commits modifying patched functions. Generalizes state-of-the-art methods like VSZZ and V0Finder.

KEY BENEFIT

Ensures broad coverage and supports patches with only

Commit-Message-Based Selection

Extracts hints from commit messages using LLM to identify functions or variables mentioned in text but not present in code changes.

KEY BENEFIT

Identifies BICs in different functions/files; complementary

BIC Selection - Two-Round Comparative Analysis

The Challenge: Traditional sequential analysis stops at the first identified BIC, often missing the true culprit due to false positives (premature termination).

01

Comprehensive Identification

Instead of stopping early, the LLM inspects **ALL** candidates to identify every potential BIC.

- Eliminates premature termination
- Collects all possible suspects
- Maximizes recall

→

02

Comparative Selection

The LLM compares all identified potential BICs **side-by-side** to select the final one.

- Leverages comparative reasoning strength
- Relative assessment reduces errors
- Selects the most probable cause

Complete LLMBisect Pipeline Architecture

INPUT

Bug-Fix Commit

Code + Message

PHASE 1: CANDIDATE GENERATION

Critical-Line
Line-level granularity

Function-Based
Modified functions

Commit-Message
Textual hints

Union & Filter Duplicates

PHASE 2: ROUND 1

LLM Analysis

Inspect ALL candidates. Identify potential BICs. No early stop.

PHASE 3: ROUND 2

Comparative Selection

Compare potential BICs side-by-side. Majority Voting.

Identified BIC

Evaluation Setup and Dataset

TARGET SYSTEM

Linux Kernel CVEs

200

Total Cases

- 100 cases from 2023
- 100 cases from 2024

(2024 is post-LLM cutoff)

GROUND TRUTH GENERATION

We utilized "Fixes:" tags provided by kernel developers, followed by manual verification of all 200 cases.

Critical Control: **Fix tags were removed** from the input during experiments to prevent data leakage and ensure fair evaluation.

COMPARISON TARGETS

SymBisect

POC-BASED

State-of-the-art symbolic execution tool.

VSZZ

SZZ-STYLE

Advanced SZZ algorithm for vulnerabilities.

VoFinder

CLONE DETECTION

Vulnerable code clone detection tool.

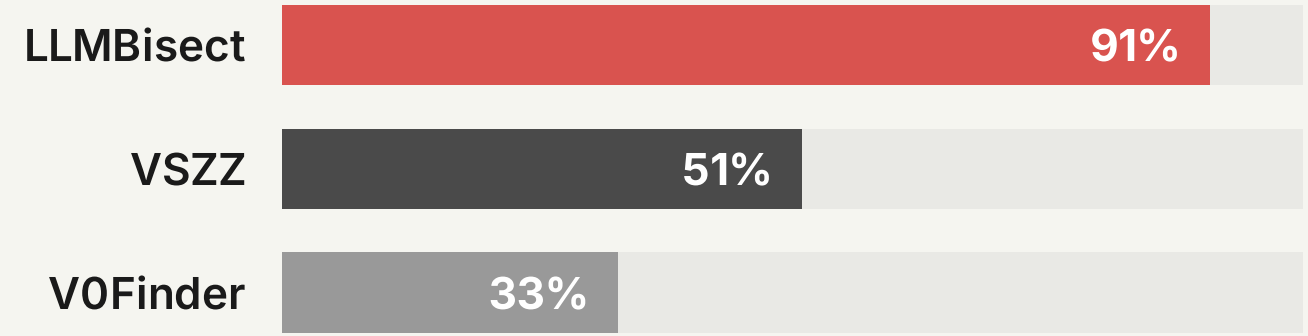
LLMBisect Achieves 91% Accuracy

91%

ACCURACY

Significantly outperforms state-of-the-art methods by more than **38%**.

Accuracy Comparison (200 CVEs)



CONSISTENT PERFORMANCE

2023: 92% Accuracy
2024: 90% Accuracy

VS. SYMBISECT

On 32 syzbot bugs:
LLMBisect: 90.6% vs
SymBisect: 75%

Ablation Study: Component Impact

Design Component	Accuracy	Improvement
Baseline (Patch-Function Only)	30.5%	-
+ Coarse Filtering	58.0%	**+27.5%**
+ Critical-Line Candidates (C2)	81.5%	**+23.5%**
+ Commit-Message Candidates (C3)	84.0%	**+2.5%**
+ All Candidates (C1+C2+C3)	87.0%	**+3.0%**
+ Majority Voting (Final)	**91.0%**	**+4.0%**

Filtering is Critical

Applying the coarse filter nearly **doubles** accuracy (30% → 58%) by removing irrelevant noise early.

Multi-Source Synergy

Combining syntactic (C1, C2) and semantic (C3) candidates pushes performance to **91%**, proving that no single view is sufficient.

Cost Analysis

~$4.9

AVERAGE COST PER CVE

(Using OpenAI o1 model)

330k

AVG. TOKENS

Input + Output

→ **High ROI:** The cost is negligible compared to the hours of expert manual analysis required for bisection.

→ **Majority Voting:** While running the final step 7 times increases token usage, it is essential for achieving 91% accuracy.

→ **Optimization:** Future work with open-source models (e.g., Llama 3) can further reduce this cost.

Impact & Future Directions

91%

SOTA Performance

Outperforms existing tools (SZZ, SymBisect) by over 38%, setting a new benchmark.

1st

LLM Approach

Pioneered semantic reasoning for bisection, moving beyond syntactic heuristics.

✓

Practical Utility

Proven effectiveness in downstream tasks like vulnerable version detection.

FUTURE ROADMAP

Model Evolution

Our results use **OpenAI-o1 (2024)**. Newer models released in the past year offer significant potential for even higher accuracy.

Multi-Language

Expand support to Java, Python, and Rust ecosystems.

Open Source

Adapt for Llama 3 / DeepSeek to enable low-cost, on-premise deployment.

Summary

The Barrier

Traditional methods struggle to balance precision and recall, limited by rigid heuristics and scalability issues.

The Innovation

LLMBisect introduces a **comparative analysis pipeline** that leverages LLMs for semantic understanding and logical reasoning.

The Result

Achieves **91% accuracy** (vs. 51% SOTA), offering a practical, scalable solution for automated bug bisection.

Conclusion

Thank You

Questions?

 github.com/seclab-ucr/LLMBisect