VICBench: A Multi-Language Benchmark for Code Vulnerability Detection

Anonymous ACL submission

Abstract

Evaluating security vulnerability detection tools requires benchmark datasets with vulnerability-inducing commits (VICs)—the commits that first introduce vulnerabilities into codebases. VICs are essential for determining the full range of vulnerable software versions. Existing vulnerability datasets suffer from limited programming language coverage, restricted patch complexity, and narrow project scope. Through our dual annotation by human experts and an agentic workflow, we create a benchmark—*VICBench*—of 100 verified VICs for 100 CVEs across 88 projects in Python, Java, and C++, covering 48 CWE types. VICBench features complex real-world vulnerability fixes averaging 38.6 lines and corresponding VICs of 252.5 lines—significantly larger than prior work. Our evaluation shows that state-of-the-art algorithms V-SZZ and LLM4SZZ achieve only 33.3%–40.1% F1, confirming that using existing approaches still entails significant manual effort. VICBench enables robust evaluation of vulnerability detection approaches.

1 Introduction

Software vulnerability detection is critical for system security. Proactive detection—identifying vulnerabilities before they reach production—has become increasingly important as detection approaches evolve from static analyzers to deep learning models (Lin et al., 2020; Steenhoek et al., 2023; Rahman et al., 2024) and LLM-powered code review assistants (Li et al., 2025; Anthropic, 2025). Robust evaluation of these diverse approaches requires high-quality benchmark datasets with vulnerability-inducing commits (VICs). Following Bao et al. (2022), we define VICs as the commits that first introduce vulnerabilities into codebases. VICs mark the beginning of vulnerability lifecycles and represent the transition from non-vulnerable to exploitable code, enabling evaluation on original vulnerability patterns before refactoring obscures the initial flaw.

A comprehensive benchmark requires repository context, CVE descriptions, fixing commits, and critically, VICs. While the National Vulnerability Database (NIST, 2026) provides CVE descriptions and fixes, VICs are not documented. This allows assessing whether tools can detect vulnerabilities at their source and tracking detection capability throughout vulnerability evolution. Without VICs, benchmarks can only evaluate tools on evolved vulnerability forms before fixes, missing evaluation on the earliest and most critical vulnerability patterns.

Existing VIC datasets have significant limitations. V-SZZ (Bao et al., 2022) verified 172 CVEs but restricted annotations to 1-5 deleted lines, excluding complex multi-file patches. Jiang et al. (2024) contributed 1,000+ VICs but focused solely on Linux kernel (C/C++), limiting generalizability. Chen et al. (2025) annotated 1,128 C/C++ vulnerabilities at high cost (0.5 person-hours each) but did not release VIC annotations. Modern approaches require greater language diversity, patch complexity, and project coverage. Moreover, automated VIC detection faces challenges. Tracing-based methods such as SZZ (Śliwerski et al., 2005) and variants (Bao et al., 2022; Tang et al., 2025) use `git-blame` heuristics, which struggle with file refactoring, cross-file code movement, and complex vulnerability semantics. Unlike regular bugs, vulnerabilities persist longer due to fundamental design flaws (Nguyen et al., 2016) and often involve addition-only fixes without deleted lines (Chen et al., 2025), making traditional approaches ineffective, with studies showing SZZ variants achieve as low as **9% precision** and a best-case F-measure of only 61% even on regular bugs (Rosa et al., 2021). Our contributions include:

- A benchmark of **100 verified VICs** for 100

CVEs across 88 projects, 3 programming languages (Python, Java, and C++), and 48 CWE types. Our dataset features significantly larger patches than prior work: fix commits average 38.6 lines and VICs average 252.5 lines.

- A dual-annotation approach with human expert verification and agentic workflow assistance achieving κ=0.707 agreement, ensuring data quality and scalable dataset construction.
- An evaluation showing that existing VIC detection algorithms achieve only 33.3%–40.1% F1, indicating that existing automated approaches are not substitutes for manual annotation.

Availability: https://zenodo.org/records/18944736

2 Background and Related Work

Vulnerability-Inducing Commit Detection Tracing-based approaches for identifying VICs build on the Śliwerski-Zimmermann-Zeller (SZZ) algorithm (Śliwerski et al., 2005), originally designed for bug-inducing commits. VCCFinder (Perl et al., 2015) first adapted SZZ to vulnerabilities, followed by variants including AG-SZZ (Kim et al., 2006), MA-SZZ (da Costa et al., 2017), and RA-SZZ (Neto et al., 2018) evaluated in V-SZZ (Bao et al., 2022). Recent approaches like SEM-SZZ (Tang et al., 2024) use semantic analysis with data and control flow, while VERCATION (Cheng et al., 2025) and LLM4SZZ (Tang et al., 2025) incorporate LLMs for improved analysis. However, these methods struggle with file refactoring, cross-file code movement, and addition-only fixes (Chen et al., 2025). Concurrent work by Shi et al. (2026) addresses some of these limitations using graph-based reasoning for general bug-inducing commits, though without released datasets or security-specific focus. These challenges motivate the need for manually curated benchmarks that capture real-world complexity.

Vulnerability-Inducing Datasets Given the distinction between bug-inducing and vulnerability-inducing tasks (Camilo et al., 2015), several VIC datasets have been developed. VCCFinder (Perl et al., 2015) constructed 718 CVEs but was found unreproducible (Riom et al., 2021). V-SZZ (Bao et al., 2022) manually verified 172 CVEs across 46 projects but restricted annotations to fixes with 1-5 deleted lines, limiting coverage of complex patches. Jiang et al. (2024) contributed 1,000+ VICs but focused solely on Linux kernel (C/C++), limiting language diversity. Concurrent work by Chen et al. (2025) curated 1,128 C/C++ vulnerabilities from nine projects at high annotation cost (0.5 person-hours each) but did not release VIC annotations. Our dataset addresses these limitations by covering 100 CVEs across three programming languages (Python, Java, and C++) and 88 projects with significantly larger and more complex patches (Table 1), providing a robust publicly-available benchmark for evaluating vulnerability detection approaches.

3 Motivating Example

CVE-2021-44878, a pac4j vulnerability accepting unsigned ID tokens, demonstrates cross-file code movement defeats git-blame approaches (Figure 1).

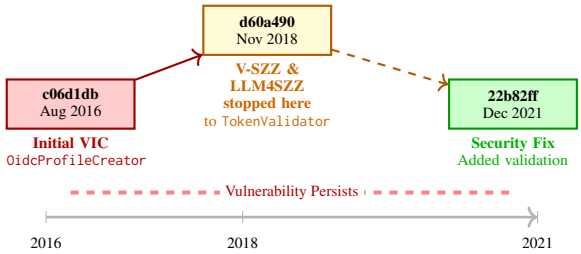

Figure 1: CVE-2021-44878 evolution timeline. V-SZZ and LLM4SZZ stopped at the 2018 refactoring commit (d60a490) where code moved to a new file, missing the true VIC (c06d1db) from 2016.

Step 1: Initial Introduction (c06d1db, Aug 2016) The vulnerability was introduced in `OidcProfileCreator.java`, accepting unsigned tokens without validation:

```
if ("none".equals(jwsAlgorithm.getName())) {
    jwsAlgorithm = null;  // UNSAFE: Accept unsigned tokens
} ...
if (jwsAlgorithm == null) {
    this.idTokenValidator = new IDTokenValidator(
        issuer, clientID);  // No signature verification!}
```

Step 2: Code Movement (d60a490, Nov 2018) Code extracted to `TokenValidator.java`, preserving the vulnerable pattern:

```
if ("none".equals(jwsAlgorithm.getName())) {
    jwsAlgorithm = null;  // STILL UNSAFE
} ...
if (jwsAlgorithm == null) {
    idTokenValidator = new IDTokenValidator(
        issuer, clientID);  // STILL no verification!}
```

Step 3: Security Fix (22b82ffd, Dec 2021) Added validation to reject unsigned tokens:

```
-if ("none".equals(jwsAlgorithm.getName())) {
-    jwsAlgorithm = null;
-}
```

```
final IDTokenValidator idTokenValidator;
-if (jwsAlgorithm == null) {
+if ("none".equals(jwsAlgorithm.getName())) {
+    if (!configuration.isAllowUnsignedIdTokens()) {
+        throw new TechnicalException(...)
+    logger.warn("Allowing unsigned ID tokens");
    idTokenValidator = new IDTokenValidator(...);
```

Git Blame Limitation Both V-SZZ and LLM4SZZ rely solely on `git blame`, which returns commit d60a490 with "New File" markers when tracing `TokenValidator.java`. Git blame cannot track cross-file code movement—only file renames—causing both algorithms to misidentify the refactoring commit as the vulnerability origin.

Pattern-Based Identification VIC-Agent combines `git blame` with LLM-guided pattern search using `git log -S` to cross file boundaries. By extracting the vulnerable pattern and searching across all files and history, it found two candidates: d60a490 (2018) in `TokenValidator.java` and c06d1db (2016) in `OidcProfileCreator.java`. It classified d60a490 as refactoring and verified c06d1db as the true VIC.

This example demonstrates that VIC detection requires reasoning about vulnerability semantics and code evolution across file boundaries, motivating the need for manually curated benchmarks. An additional case study (CVE-2019-15477) in Appendix A.1 illustrates complementary challenges.

4 Benchmark Creation

4.1 Annotation Approach

To ensure accurate labeling of VICs, we design a workflow (Figure 2) in which two independent procedures identify VICs for each vulnerability, and their results are then cross-validated to resolve discrepancies.

4.1.1 Procedure 1: Human Annotation

One author with nine years of programming experience annotated VICs for all 100 instances. Following Rodriguez-Perez et al. (2020) and V-SZZ (Bao et al., 2022), the annotator iteratively traced modified lines using CVE/CWE descriptions, commit messages, and git blame. While V-SZZ identifies the *earliest* commit containing vulnerable lines (Bao et al., 2022), our annotation focuses on when the vulnerability as a *security flaw* was semantically introduced—critical for cases involving refactoring or code movement. This required: (1) examining function names and program logic for addition-only fixes; (2) tracking only security-relevant lines.

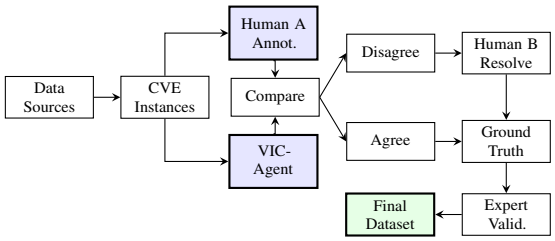

Figure 2: Annotation workflow: data collection, independent annotation procedures (human and agentic), disagreement resolution, and expert validation.

4.1.2 Procedure 2: VIC-Agent

To validate the robustness of our manual annotations and enable scalable dataset construction, we developed VIC-Agent, an automated workflow powered by large language models. Unlike traditional SZZ algorithms that rely on fixed heuristics, VIC-Agent brings critical flexibility through adaptive decision-making at each step, dynamically selecting between git tools (`git blame`, `git log -S`, `git show`) and employing LLM-based vulnerability analysis and commit verification.

The agent first analyzes the CVE description and fixing commit to understand the vulnerability's *security semantics*. When examining candidate commits, it uses LLM reasoning to verify whether a commit *introduced* new vulnerable logic or merely *refactored* existing vulnerable code. For refactored commits, it continues searching backwards through the pattern evolution chain.

This enables VIC-Agent to handle real-world complexities—file renames, cross-file movements, and refactoring patterns—that static algorithms cannot address. VIC-Agent achieved 84.4% recall and 94.9% precision in agreement with human annotations (see Section 5), reflecting its role as a construction tool and validating the dataset quality.

4.2 Quality Validation

Two annotations were considered in agreement if and only if both identified the exact same commit.

Inter-Annotator Agreement: Human A and VIC-Agent achieved 71% observed agreement (71 of 100 cases). Computing Cohen's kappa with chance agreement correction (see Appendix A.3 for details), we obtained $\kappa = 0.707$, indicating *substantial agreement*. The observed agreement (71%) far exceeds chance expectation (1.15%).

Disagreement Resolution: For the 29 disagreement cases, a second human annotator (Human B) independently reviewed each instance, examining commit context, vulnerability semantics, and code evolution history. Human B incorporated insights from Human A's annotations and VIC-Agent's reasoning to establish the ground truth annotations.

Expert Validation: To further validate dataset quality, a principal security engineer with 16 years of security experience independently reviewed a random sample of 11 instances from the final dataset (11%). This validation confirmed the correctness of the vulnerability-inducing commit identifications and provided additional confidence in the dataset's reliability.

4.3 Data Sources

We randomly sampled 100 instances from 88 projects across Python, Java, and C/C++ from three datasets: (1) **ReposVul** (Wang et al., 2024), filtered for entries with CVE-relevant files, (2) **CWE-Bench-Java** (Li et al., 2025), and (3) **VJBench** (Wu et al., 2023). Our data set cover diverse CWE and patch types.

5 Dataset Analysis

5.1 Dataset Composition and Characteristics

Vulnerability Type Coverage: Our dataset encompasses 48 unique CWE types across 100 CVEs, with Cross-site Scripting (CWE-79, 12%) and Path Traversal (CWE-22, 10%) being most prevalent (see Appendix A.2). This diversity ensures comprehensive evaluation across varied security contexts.

Patch Complexity: Table 1 compares VICBench with existing benchmarks. Fix commits average 38.6 lines changed (median: 22), while VICs average 252.5 lines (median: 181). This 6.5X difference reflects real-world scenarios where vulnerabilities occur in larger feature implementations.

Dataset	N	Lang	Prj	Fix/VIC	Pub
V-SZZ (2022)	172	C/Ja	46	1-5 / -	Y
Jiang et al. (2024)	1000+	C	1	- / -	Y
Chen et al. (2025)	1128	C	9	Div / -	N
Ours	**100**	**Py/Ja/C**	**88**	**39 / 253**	**Y**

Table 1: Dataset comparison. Fix/VIC Lines show mean changed lines. Pub: Y/N indicates publicly available/unavailable. Our dataset has larger patches, languages (Python/Java/C++), and more projects than prior work.

5.2 VIC Detection Performance and Dataset Construction Validation

To demonstrate that our dataset could not be trivially acquired by running existing algorithms, we evaluated V-SZZ (Bao et al., 2022) (Java/C++ subset, 40 CVEs) and LLM4SZZ (Tang et al., 2025) (all 100 instances). Table 2 shows that V-SZZ achieves 33.3% F1 and LLM4SZZ achieves 40.1% F1, demonstrating that our dataset captures complexities beyond current automated approaches. Accurate VIC identification has practical implications for version range determination: Bao et al. (2022) found that 99 out of 172 CVEs had spurious version information in NVD and proposed an improved SZZ algorithm to address this. However, cases involving code refactoring and cross-file movement remain challenging—for example, in our motivating example of CVE-2021-44878 (Section 3), identifying the 2018 refactoring rather than the 2016 true VIC would incorrectly exclude two years of vulnerable versions. Our VIC-Agent achieves 89.3% F1 by focusing on semantic vulnerability introduction, extending prior work to handle such complex scenarios.

Note that VIC-Agent's performance should be interpreted as a construction-tool reference ceiling rather than an independent evaluation, as it participated in the annotation process (Cao et al., 2025).

Method	Recall	Precision	F1 Score
V-SZZ†	52.3%	24.5%	33.3%
LLM4SZZ	56.9%	31.0%	40.1%
VIC-Agent	84.4%	94.9%	89.3%

Table 2: VIC detection performance. †V-SZZ on Java/C++ only (40 CVEs), LLM4SZZ and VIC-Agent on all 100 instances.

6 Conclusion

We present a benchmark of 100 verified vulnerability-inducing commits across Python, Java, and C++, addressing critical gaps in existing VIC datasets. Through dual annotation achieving κ=0.707 agreement, our dataset captures real-world complexity with significantly larger patches than prior work. Evaluation shows state-of-the-art algorithms achieve only 33.3%–40.1% F1, confirming the necessity of manual curation. This benchmark enables robust evaluation of vulnerability detection approaches and supports research in proactive security tooling.

Limitations

While our dataset advances VIC benchmarking, several limitations should be acknowledged:

Dataset Scale and Coverage: Our dataset contains 100 manually verified CVEs, which is smaller than some automatically constructed datasets but prioritizes annotation quality over quantity. The dataset covers three programming languages (Python, Java, C++), which may not generalize to other languages such as JavaScript, Go, or Rust. Additionally, C++ is underrepresented at 8% (8 CVEs), reflecting its limited availability in our source datasets (ReposVul, CWE-Bench-Java, VJBench) rather than a deliberate design choice; this may limit the benchmark's utility for evaluating tools targeting C/C++ codebases specifically. Future work could expand coverage to additional languages and larger scale while maintaining annotation quality.

Data Source Selection: Our CVEs were sampled from existing curated datasets (ReposVul, CWE-Bench-Java, VJBench), which may introduce sampling bias toward certain project types, vulnerability patterns, or temporal distributions. The dataset focuses on open-source projects from public repositories, and findings may not fully generalize to proprietary or closed-source codebases.

Single VIC Assumption: Our annotation focused on identifying the primary vulnerability-inducing commit for each CVE. Some vulnerabilities may result from multiple contributing commits across different timeframes, which our current annotation does not capture exhaustively.

Ethical Considerations

This work presents a benchmark dataset for evaluating vulnerability-inducing commit (VIC) detection tools. All CVEs and source code used in this benchmark are publicly available through the National Vulnerability Database (NIST, 2026) and open-source repositories. No private or proprietary data was collected.

We acknowledge the dual-use nature of security benchmarks: while VICBench is intended to advance the evaluation of defensive security tools, detailed vulnerability information could theoretically inform offensive use. To mitigate this risk, all CVEs included in this dataset are already publicly disclosed and patched. The benchmark does not introduce new vulnerability information beyond what is already publicly available.

No human subjects were involved in dataset construction. The annotation process was conducted by the authors and a security expert reviewer.

References

Anthropic. 2025. Automate security reviews with claude code. Accessed: 2025.

Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-szz: Automatic identification of version ranges affected by cve vulnerabilities. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 2352–2364.

Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan. 2015. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 269–279.

Jialun Cao and 1 others. 2025. Rigor, reliability, and reproducibility matter: A decade-scale survey of 572 code benchmarks. *arXiv preprint*.

Xingchu Chen, Chengwei Liu, Jialun Cao, Yang Xiao, Xinyue Cai, Yeting Li, Jingyi Shi, Tianqi Sun, Haiming Chen, and Wei Huo. 2025. Vulnerability-affected versions identification: How far are we? *Preprint*, arXiv:2509.03876.

Yiran Cheng, Ting Zhang, Lwin Khin Shar, Shouguo Yang, Chaopeng Dong, David Lo, Shichao Lv, Zhiqiang Shi, and Limin Sun. 2025. Vercation: Precise vulnerable open-source software version identification based on static analysis and llm. *IEEE Transactions on Software Engineering*, pages 1–19.

Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657.

Muhui Jiang, Jinan Jiang, Tao Wu, Zuchao Ma, Xiapu Luo, and Yajin Zhou. 2024. Understanding vulnerability inducing commits of the linux kernel. *ACM Trans. Softw. Eng. Methodol.*, 33(7).

Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic identification of bug-introducing changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, pages 81–90.

Ziyang Li, Saikat Dutta, and Mayur Naik. 2025. IRIS: LLM-assisted static analysis for detecting security vulnerabilities. In *The Thirteenth International Conference on Learning Representations*.

Guanjun Lin, Jun Zhang, Wei Luo, Lei Pan, Yang Xiang, Olivier De Vel, and Paul Montague. 2020. Software vulnerability detection using deep neural networks: A survey. *Proceedings of the IEEE*, 108(10):1825–1848.

Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the szz algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 380–390.

Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empirical Softw. Engg.*, 21(6):2268–2297.

NIST. 2026. National vulnerability database. Accessed: 2026.

Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 426–437, New York, NY, USA. Association for Computing Machinery.

Md Mahbubur Rahman, Ira Ceka, Chengzhi Mao, Saikat Chakraborty, Baishakhi Ray, and Wei Le. 2024. Towards causal deep learning for vulnerability detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Timothé Riom, Arthur Sawadogo, Kevin Allix, Tegawendé F. Bissyandé, Naouel Moha, and Jacques Klein. 2021. Revisiting the vccfinder approach for the identification of vulnerability-contributing commits. *Empirical Softw. Engg.*, 26(3).

Guillermo Rodriguez-Perez, Gregorio Robles, Alexander Serebrenik, Arie Zaidman, Jose Miguel German, and Jesus M Gonzalez-Barahona. 2020. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25(6):1294–1340.

Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating szz implementations through a developer-informed oracle. In *Proceedings of the 43rd International Conference on Software Engineering*, ICSE '21, page 436–447. IEEE Press.

Yu Shi, Hao Li, Bram Adams, and Ahmed E. Hassan. 2026. Beyond blame: Rethinking szz with knowledge graph search. *arXiv preprint arXiv:2602.02934*.

Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An empirical study of deep learning models for vulnerability detection. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, ICSE '23, page 2133–2145. IEEE Press.

Lingxiao Tang, Jiakun Liu, Zhongxin Liu, Xiaohu Yang, and Lingfeng Bao. 2025. Llm4szz: Enhancing szz algorithm with context-enhanced assessment on large language models. *Proc. ACM Softw. Eng.*, 2(ISSTA).

Lingxiao Tang, Chao Ni, Qiao Huang, and Lingfeng Bao. 2024. Enhancing bug-inducing commit identification: A fine-grained semantic analysis approach. *IEEE Transactions on Software Engineering*, 50(11):3037–3052.

Xinchen Wang, Ruida Hu, Cuiyun Gao, Xin-Cheng Wen, Yujia Chen, and Qing Liao. 2024. Reposvul: A repository-level high-quality vulnerability dataset. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*, ICSE-Companion '24, page 472–483, New York, NY, USA. Association for Computing Machinery.

Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How effective are neural networks for fixing security vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, page 1282–1294, New York, NY, USA. Association for Computing Machinery.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? In *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, page 1–5, New York, NY, USA. Association for Computing Machinery.

A Appendix

A.1 Additional Case Study: CVE-2019-15477

CVE-2019-15477 is an XSS vulnerability in Jooby's default error handler that allowed unescaped error messages to be rendered in HTML responses. This case demonstrates why automated VIC detection fails when vulnerabilities persist through file deletions and architectural refactorings. Figure 3 shows the 5-year evolution of this vulnerability across major code transformations.

Commit Evolution History The vulnerability followed this evolution through the codebase:

1. **Initial Introduction (d548655, Jun 2014):** Created `RouteHandler.java` with unescaped error model:

```
private Map<String,Object> errorModel(
    Request request, Exception ex, HttpStatus status)
      {
    Map<String,Object> error = new LinkedHashMap<>();
    String message = ex.getMessage();
    message = message == null ? status.reason() :
      message;
    error.put("message", message);       // UNESCAPED
    error.put("stackTrace", dump(ex));
    error.put("status", status.value());
    error.put("reason", status.reason());  //
      UNESCAPED
    error.put("referer", request.header("Referer")...)
      ;
    return error;
}
```

2. **File Deletion (2071f2e, Feb 2015):** `RouteHandler.java` deleted; error logic moved to `Err.java` interface default method. The unescaped message handling persisted.

3. **LLM4SZZ Stopped (49b524c, May 2015):** Refactored to use `toMap()` method in `DefHandler` class. The vulnerability remained:

```
Results.when(MediaType.html, () ->
    Results.html(VIEW).put("err", ex.toMap()))
```

4. **V-SZZ Stopped (8a3308c, Sep 2017):** Added stacktrace parameter `toMap(stackstrace)`. Still unescaped:

```
err.put("message", message);  // Still unescaped!
err.put("reason", status.reason());  // Still
      unescaped!
```

5. **Fix Applied (27b4af2, Aug 2019):** Merge commit integrating PR #1368 that added XSS filtering after 5 years:

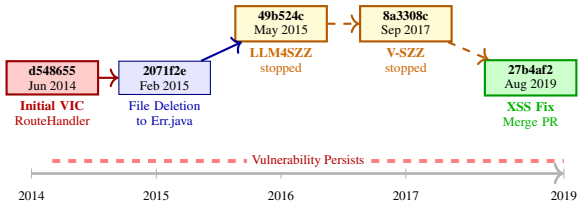

Figure 3: CVE-2019-15477 evolution timeline showing vulnerability persistence through file deletion and refactorings from Jun 2014 to Aug 2019.

```
Function<Object,String> xssFilter = env.xss("html")
      ...;
details.compute("message", escaper);  // FIXED
details.compute("reason", escaper);   // FIXED
```

Algorithm Performance Analysis The **V-SZZ** stopped at commit 8a3308c (Sep 2017), which only added a boolean parameter for stacktrace visibility. Git blame cannot trace beyond the Feb 2015 file deletion, treating `Err.java` as unrelated to the deleted `RouteHandler.java`. **LLM4SZZ** stopped at commit 49b524c (May 2015), misidentifying architectural refactoring as vulnerability introduction and hallucinating a false root cause about "Supplier patterns." **VIC-Agent** identified d548655 (Jun 2014) through iterative backward search using `git log -S` to trace the vulnerability pattern (unescaped error messages) through 6 commits: 8a3308c (parameter addition), 49b524c (method refactoring), 2071f2e (package move), 3f19db103 (file extraction), and finally d548655 (origin). The approach tracked semantic continuity across 3 method names (`errorModel()` → `err()` → `toMap()`) and multiple files despite transformations that broke line-based and blame-based tracing.

A.2 CWE Type Distribution

Table 3 shows the distribution of CWE types in our dataset.

CWE	Count	Description
CWE-79	12	Cross-site Scripting (XSS)
CWE-22	10	Path Traversal
CWE-20	8	Improper Input Validation
CWE-611	4	XML External Entity (XXE)
CWE-601	4	URL Redirection to Untrusted Site
Other (43)	**63**	**Diverse types**

Table 3: Top 5 CWE types in our dataset. Total of 48 unique CWE types across 100 instances.

A.3 Cohen's Kappa Calculation Details

Cohen's kappa is defined as:

$$\kappa = \frac{P_o - P_e}{1 - P_e} \tag{1}$$

where P_o is the observed agreement proportion and P_e is the expected agreement by chance.

Agreement Criteria: Two annotations were considered in agreement if and only if both identified the exact same commit. This strict definition avoids circular reasoning about ground truth.

For VIC annotation, the probability of random agreement depends on the commit pool size from which annotators could select. For each CVE, we identified the earliest VIC between the two annotations and counted commits from this earliest VIC (inclusive) to the fix commit (exclusive) using git history. Pool sizes ranged from 2 to 212,671 commits with a median of 1,690 commits. Expected agreement was computed as:

$$P_e = \frac{1}{N} \sum_{i=1}^{N} \frac{1}{n_i} \tag{2}$$

where $N = 100$ is the number of CVEs and n_i is the commit pool size for CVE i.

In our case, $P_o = 0.71$ (71 agreements out of 100 cases) and $P_e = 0.0115$ (1.15%), yielding $\kappa = 0.707$.