Vulnerability-Affected Versions Identification: How Far Are We?

Xingchu Chen[1,2†‡], Chengwei Liu[3§], Jialun Cao[4], Yang Xiao[1,2*†‡],
Xinyue Cai[1,2†‡], Yeting Li[1,2 †‡], Jingyi Shi[1,2†‡], Tianqi Sun[1,2†‡], Haiming Chen[5], Wei Huo[1,2*†‡],
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]Nanyang Technological University, Singapore
[4]The Hong Kong University of Science and Technology, Hong Kong, China
[5]Institute of Software, UCAS, Beijing, China
{chenxingchu, xiaoyang, caixinyue, liyeting, shijingyi, suntianqi, huowei}@iie.ac.cn
chengwei.liu@ntu.edu.sg, jcaoap@cse.ust.hk, chm@ios.ac.cn

***Abstract*—Identifying which software versions are affected by a vulnerability is critical for patching, risk mitigation. Despite a growing body of tools, their real-world effectiveness remains unclear due to narrow evaluation scopes—often limited to early SZZ variants, outdated techniques, and small or coarse-grained datasets. In this paper, we present the first comprehensive empirical study of vulnerability-affected versions identification. We curate a high-quality benchmark of 1,128 real-world C/C++ vulnerabilities and systematically evaluate 12 representative tools from both tracing and matching paradigms across four dimensions: effectiveness at both vulnerability and version levels, root causes of false positives and negatives, sensitivity to patch characteristics, and ensemble potential. Our findings reveal fundamental limitations: no tool exceeds 45.0% accuracy, with key challenges stemming from heuristic dependence, limited semantic reasoning, and rigid matching logic. Patch structures such as add-only and cross-file changes further hinder performance. Although ensemble strategies can improve results by up to 10.1%, overall accuracy remains below 60.0%, highlighting the need for fundamentally new approaches. Moreover, our study offers actionable insights to guide tool development, combination strategies, and future research in this critical area. Finally, we release the replicated code and benchmark on our website to encourage future contributions.**

***Index Terms*—Vulnerability-affected versions identification, SZZ Algorithm, Vulnerability Detection, Combination Strategies.**

I. Introduction

Software vulnerabilities continue to pose serious threats to system security. For defenders and researchers, it is critical to determine *which software versions are affected by a known vulnerability*, referred to as *vulnerability-affected version identification.* Accurate identification underpins a wide range of downstream applications, including vulnerability detection [2]–[5], automated patching [6], [7], and exploitability or propagation analysis [8]–[10].

Despite its practical importance, reliably identifying affected versions remains challenging. Public vulnerability databases such as the NVD often contain incomplete or incorrect version metadata [11]–[13], and many real-world vulnerabilities, especially those patched silently, are usually not documented at all [14]–[16]. As a result, practitioners frequently turn to analysis tools to infer affected versions retrospectively.

A range of such tools has emerged, falling broadly into two categories. Tracing-based methods [17]–[22] extend the classic SZZ algorithm [23] to backtrack vulnerability-inducing commits and map them to released versions. Matching-based methods [13], [24]–[29] extract vulnerability-related code signatures and scan historical versions for semantic equivalence. However, despite promising individual results, the *real-world effectiveness of these tools remains unclear* due to the lack of standardized, large-scale evaluation. Previous studies fall short in several ways. First, existing evaluations focus narrowly on early tracing tools, assess only a few projects. [30]–[32] And previous studies does not provide a thorough investigation. Second, inconsistent metrics and evaluation setups make cross-tool comparisons unreliable. Third, newer and more sophisticated tools, particularly in the matching domain, remain largely unevaluated. As a result, developers lack systematic guidance on tool selection or integration, and researchers cannot build on reliable baselines. To address this gap, our study goes substantially further by conducting stage-wise root cause analysis, quantifying the severity of failures, and identifying previously unreported challenges.

Conducting a comprehensive evaluation presents two key challenges. First, building a high-quality benchmark requires curating ground-truth affected versions for real-world vulnerabilities, which is non-trivial given the inconsistent documentation, complex version histories, and the need for manual validation. Second, understanding why tools fail involves disentangling internal heuristics, code change semantics, and tool-specific assumptions, which are barely unveiled by prior studies.

[*] Corresponding author.
[†] Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences, Beijing, China
[‡] Beijing Key Laboratory of Network Security and Protection Technology, Beijing, China
[§] China-Singapore International Joint Research Institute (CSIJRI), Guangzhou, China

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00244

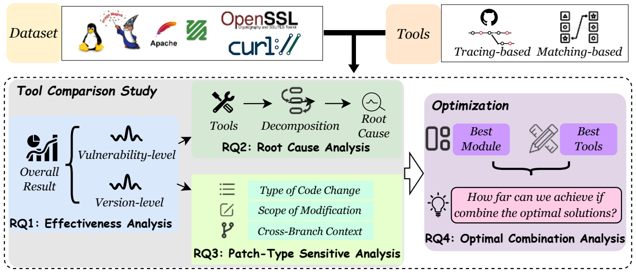


Figure 1. Overview of Our Study.

To fill these gaps, we present the first systematic empirical study on vulnerability-affected versions identification, as shown in Figure 1. First, we build a manually curated benchmark of 1,128 real-world vulnerabilities across 132 vulnerability types from diverse C/C++ projects. Created over 1.5 months with strict validation, this dataset provides fine-grained, reproducible ground truth for tool evaluation. Second, we conduct a comprehensive comparison of 12 representative tools from both tracing and matching paradigms. The evaluation spans four dimensions: (1) effectiveness at both the vulnerability and version level, (2) root cause analysis of false positives (FPs) and false negatives (FNs), (3) sensitivity to diverse patch patterns (e.g., add-only, cross-file), and (4) potential gains from hybrid or ensemble strategies. Third, to facilitate principled diagnosis, we introduce a stage-based decomposition framework that abstracts tracing and matching workflows into comparable steps. We then analyze the strategies of each tool at every stage, and conduct joint qualitative and quantitative assessments to pinpoint systematic errors.

Throughout the empirical study, we have discovered the following key findings: 1) no tool achieves more than 45.0% accuracy in identifying affected versions, raising serious concerns about their suitability for security-critical tasks; 2) The primary causes of FPs and FNs include heuristic over-reliance in tracing, insufficient semantic modeling and inflexible matching; 3) Add-only patches, cross-file changes, and multi-branch development significantly degrade performance; 4) Modular or ensemble combinations can improve accuracy by up to 10.1%, but systemic design flaws cap accuracy below 60.0%, signaling the need for fundamentally new approaches.

In summary, our main contributions are as follows:

- **Novelty** – We constructed the first large-scale, high-quality benchmark for vulnerability-affected version identification, covering 1,128 vulnerabilities across 132 vulnerability types from diverse C/C++ projects. This manually curated dataset, built over 1.5 months, can effectively support further reproducible and reliable evaluation of identification tools.
- **Rigorousness** – We performed a comprehensive empirical evaluation of 12 representative tools from two major categories, offering an in-depth assessment of their effectiveness across multiple dimensions.
- **Significance** – We identified key technical challenges and root causes behind tool limitations, and provided practical insights for tool improvement, ensemble design, and future research directions.
- **Usefulness** – We release the high-quality benchmark with 1,128 vulnerabilities across 132 vulnerability types from diverse C/C++ projects to ensure transparency and encourage future study. The artifact is released on the website [1], including replicated code and the benchmark.

II. Preliminaries and Related Work

A. Vulnerability-Affected Versions Identification

Given a reported vulnerability vul (e.g., a CVE) in a project P, public databases like NVD often list affected versions (e.g., via CPEs), but such metadata is frequently incomplete or inaccurate [11]–[13], limiting its reliability for downstream tasks [5], [6]. We define the task of **vulnerability-affected versions identification** as: given the set of released versions $V = \{v_1, v_2, ..., v_n\}$ of P, identify the subset $V_{aff} \subseteq V$ where each $v_i \in V_{aff}$ retains vulnerable logic of vul. Accurate identification of V_{aff} is critical for managing N-day threats and enabling a timely response.

B. Existing Identification Approaches

Generally, existing approaches for identifying vulnerability-affected versions fall into two main types. Tracing-based methods build on the SZZ algorithm [23] to locate vulnerability-inducing commits and infer affected versions. Matching-based methods directly identify vulnerable logic across versions through static techniques (e.g., syntactic or semantic pattern matching) or dynamic analysis (e.g., fuzzing).

*1) **Tracing-based Approaches**:* Tracing-based methods infer affected versions by first identifying the commits that introduced a vulnerability. A number of approaches build on the SZZ algorithm [23], originally designed for bug-inducing commit identification. `VCCFinder` [17] was the first to adapt SZZ to vulnerability scenarios, while `Lifetime` [18] improved commit identification with fine-grained heuristics. `V-SZZ` [19] introduced the idea of mapping vulnerability-inducing commits to version tags, enabling affected version inference.

Numerous SZZ variants (e.g., `AG-SZZ` [33], `MA-SZZ` [34], `RA-SZZ` [35], `PR-SZZ` [36]) have been proposed to improve the precision of bug-inducing commit identification. More recent efforts explore semantic and learning-based enhancements. `Neural-SZZ` [37] identifies vulnerable code using graph neural networks. `Vercation` [21] and `Sem-SZZ` [20] apply program slicing for semantic backtracking. `LLM4SZZ` [22] leverages large language models (LLMs) to refine both root cause localization and commit selection.

*2) **Matching-based Approaches**:* The matching-based tools are mainly designed for recurring vulnerability detection. While tools like `ReDeBug` [38] were originally developed for clone detection, others such as `VISION` [13] and `VerJava` [24] extend this idea specifically to identify affected versions which aligns with our evaluated scenario. These approaches evolve along multiple axes—ranging from syntactic matching to semantic analysis and structural representations. Early techniques, such as `ReDeBug` [38], Li et al. [39], `PatchGen`

[40] and VUDDY [29], relied on syntactic signatures or token-level features, offering fast yet coarse-grained matching.

To enhance accuracy and reduce false positives, later methods incorporated semantic context using program slicing, taint analysis, and version filtering. Representative tools include MVP [28], MOVERY [26], Tracer [41], V1SCAN [27], FIRE [42], and VULTURE [43]. To further generalize matching across codebases and abstractions, structural approaches emerged. These include AST-based models like VMUD [44], PATEN [45], and VISION [13], as well as graph-based representations such as HiddenCPG [46]. These techniques enable richer modeling of vulnerability patterns and greater resilience to code modifications.

Besides, dynamic methods [47], [48] use symbolic execution or proof-of-concept (PoC) exploits to validate vulnerability presence in each version. While highly precise, such methods face practical limitations due to the unavailability of PoCs and challenges in reliably building and executing historical versions. So they remain constrained in large-scale, automated settings.

C. Existing Evaluations

Several studies have examined the effectiveness of SZZ and its variants to understand their limitations and improve their design. Wen et al. [30] highlighted that SZZ often fails when there is no direct overlap between the fixing and inducing commits. Rezk et al. [49] investigated the impact of ghost commits on the SZZ algorithm and construct the evaluation dataset with developers' information. Rosa [31] introduced an NLP-based method for automatically constructing evaluation datasets, while Lyu et al. [32] investigated the influence of ghost commits and explored the theoretical upper bound of SZZ's accuracy. However, these evaluations primarily focus on early SZZ variants, use limited or unverified datasets, and cover only a small number of projects. As a result, they offer only a partial view of the problem and fall short of revealing broader tool effectiveness, robustness, or real-world applicability—gaps that our comprehensive empirical study aims to fill.

D. Motivation of This Paper

Considering the exponential growth in exploited vulnerabilities, the accurate management of vulnerability information, particularly the precise delineation of affected versions, has become more critical than ever for downstream tasks, such as patch deployment and risk evaluation. Unfortunately, current vulnerability datasets often exhibit low quality and inaccurate metadata, leaving practitioners without the detail required for effective remediation. Although existing works have been proposed in the fields of inducing commit tracing based approaches and vulnerability matching based detections, none has systematically investigated how well these two types of solutions support the automated identification of vulnerable versions in real-world settings. To address this gap, in this paper, we conduct the first comprehensive study that evaluates their practical capabilities and clarifies how far we have progressed toward fully automated, accurate vulnerable version identification, thereby illuminating directions for future research.

Table I. List of selected tools. # Baseline: Tool used as a baseline. # Citation: Number of citations.

Type	Tool	#Baseline	#Citation	Publication
Tracing-based	VCCFinder	7	330	CCS'15
	V-SZZ	6	50	ICSE'22
	Lifetime	0	33	SEC'22
	SEM-SZZ	0	0	TSE'24
	TC-SZZ	0	3	TSE'24
	LLM4SZZ	0	0	ISSTA'25
Matching-based	ReDeBug	24	314	S&P'12
	VUDDY	47	467	S&P'17
	MOVERY	3	39	SEC'22
	V1SCAN	3	16	SEC'23
	FIRE	0	2	SEC'24
	VULTURE	0	0	NDSS'25

III. Study Design

A. Tool Selection

We first conduct a systematic literature review (SLR) grounded in rigorous criteria to collect related papers on these two types of approaches. Specifically, we target at top-tier software engineering and security venues (e.g., ICSE, ASE, IEEE S&P, USENIX Security) to search related papers published between May 2020 and May 2025, using keywords such as "vulnerable version", "version range", "SZZ", and "recurring vulnerability." This yielded 22 papers. After that, we further apply backward and forward snowballing to identify other relevant papers that are missed by keyword searching, and another 19 papers are included, resulting in 41 relevant papers in total for tool selection.

As most of these techniques target C/C++ projects, which is the most concerned ecosystem in vulnerability research, we excluded tools specific to other environments (e.g., VerJava [24], Neural-SZZ [37], AFV [25]) to maximize comparison targets. After that, we further filter out papers that are (1) not proposing new tools, (2) tools not being available(i.e.,VISION [13]), and (3) requiring additional information (i.e., OpenSZZ [50] requires additional information from Jira issues [51]) or compilation (i.e., Tracer [41]) that are infeasible for automated identification of vulnerable versions for general vulnerabilities. As a result, 12 tools in total, 6 matching-based tools and 6 tracing-based tools, covering diverse methodologies including heuristics, semantic reasoning, and LLM-powered analysis, are finally selected as the target tools of this study. Table I summarizes the selected tools, their classification, citation status, and publication venues. The detailed table of the SLR is presented on our website [1].

B. Dataset Construction

Existing datasets used for evaluating vulnerable version identification tools suffer from several key limitations: they either (1) target a single project [32], [52] (e.g., the Linux kernel), which include a limited number of vulnerabilities (e.g., only 100 in V-SZZ [19]), or (2) are with poor accuracy in their ground truths (Lifetime [18] directly takes the labeled

Table II. Overview of the Constructed Dataset. #Star: GitHub stars; #CVE: collected CVEs; #Patch: total patches; #Add-only / #Del-only / #Mixed: patch types by diff composition; #Version: total affected versions; Branch: development model.

Project	#Star	Domain	#CVE	#Patch	#Add-only	#Del-only	#Mixed	#Version	Branch
Linux kernel	190k	Operating Systems	717	928	222	17	689	17,324	Multiple
FFmpeg	49k	Multimedia Processing	71	238	35	0	203	9,428	Multiple
Curl	37k	Command-line Data Transfer	68	66	12	1	53	4,601	Single
OpenSSL	27k	Cryptography and Secure Communication	50	50	16	1	33	1,796	Multiple
ImageMagick	13k	Image Processing	72	71	7	0	64	12,377	Single
QEMU	11k	System Virtualization and Emulation	57	78	27	1	50	2,188	Multiple
Wireshark	8k	Network Traffic Analysis	50	53	4	0	49	8,849	Multiple
Apache HTTP Server	4k	Web Server	30	43	2	0	41	2,496	Single
OpenJPEG	1k	Image Processing	13	15	4	0	11	128	Single
Total	-	-	1,128	1,542	329	20	1,193	59,187	-

version in NVD, which is known as inaccurate). These issues hinder the generalizability and reliability of evaluation results. To address these limitations, we construct a new benchmark that is large-scale, diverse, and accurately annotated. The construction process consists of three main steps:

1) Project Selection. We select nine representative C/C++ projects based on the following criteria: (i) *popularity*, measured by GitHub stars, forks, and prior usage in vulnerability research; (ii) *vulnerability density*, ensuring the selected projects have a substantial number of reported CVEs; and (iii) *domain diversity*, covering a range of application areas.

2) Patch Collection. For each selected project, we manually collect fixing commit of vulnerability from Jan 2020 to Dec 2024. Patch identification is primarily based on CVE references from NVD, and is cross-referenced with official security advisories or project-maintained vulnerability logs.

3) Ground Truth Annotation. We determine affected version ranges following the annotation methodology of `V-SZZ` [19]. Then we further validated the affected versions as follows:

Two annotators independently identify vulnerable statements by consulting multiple sources, including CVE descriptions, vulnerability reports, GitHub issues and fixing commit. They trace the commit history of the vulnerable statements to identify the vulnerability-inducing commit. All versions between the inducing-commit and patch-commit are labeled as vulnerable. Disagreements are resolved by a third annotator through discussion and evidence inspection. Each annotator has 8 years experience in C/C++ language programming and vulnerability analysis. This multi-stage process ensures high labeling accuracy and mitigates individual bias. Among all vulnerabilities, 134 (11.9%) vulnerabilities had initial annotation inconsistencies, and the Cohen's Kappa for the labeling is 0.83.

On average, annotating each CVE requires approximately **0.5 person-hours**. To reduce complexity and preserve evaluation fidelity, we excluded release-candidate (RC) builds while labeling versions. The dataset can be found in our artifact. Table II summarizes key statistics of our constructed dataset, which includes affected version annotations for **1,128 CVEs**, covering a total of **59,983 vulnerable versions**. The dataset spans **132 distinct CWE types**, with broad coverage across vulnerability categories. To the best of our knowledge, this is the largest publicly available benchmark for vulnerability-affected version identification.

C. Experimental Setup

We evaluate tools under default settings. Tracing-based tools follow `V-SZZ` [19], labeling versions between introducing and fixing commits as vulnerable. For `LLM4SZZ`, we use llama3.1-70b [22]. For matching-based tools, we provide the fixing commit as input and apply each tool to the entire codebase of all historical versions. These tools extract features based on the patch and search for similar features to determine whether the version is vulnerable. A version is labeled as vulnerable when the tool identifies that the vulnerable code still exists. We strictly follow each tool's original setup to ensure correctness.

IV. Comparison and Evaluation

The evaluation answers four research questions (RQs):

- **RQ1: Effectiveness Analysis.** *How effective are current tools in identifying affected versions of vulnerabilities?*
- **RQ2: Root Cause Analysis.** *What are the primary causes of FPs and FNs produced by existing tools?*
- **RQ3: Patch-Type Sensitivity Analysis.** *How does identification performance vary across different patch types?*
- **RQ4: Tool Combination Analysis.** *Can combining existing tools improve the overall effectiveness?*

A. Effectiveness Analysis (RQ1)

*1) **Setup**:* The effectiveness is evaluated from two complementary perspectives:

- **Vulnerability-level Evaluation.** This evaluation assesses whether a tool accurately identifies the full set of affected versions for each vulnerability. A prediction is considered *correct* (true positive) only if it exactly matches the ground truth (no missing or extra versions). We also define a *no-miss* case as one that contains all ground-truth versions, despite the extra ones. This distinction reflects practical trade-offs: missing affected versions may introduce security risks, while over-reporting primarily increases maintenance overhead. We report two metrics: *Accuracy* (TP/Total), and *No-Miss Ratio* (NMR = No-Miss/Total), where *Total* is the number of evaluated vulnerabilities.

Table III. Effectiveness of tools at the vulnerability and version levels. NM: No-Miss; NMR: No-Miss Ratio.

Type	Tool	Vulnerability-level				Version-level					
		TP	Accuracy	NM	NMR	FP	FN	TP	Precision	Recall	F1
Tracing-based	**VCCFinder**	**506**	**44.9%**	**776**	**68.8%**	12020	13879	45308	79.0%	76.6%	**77.8%**
	V-SZZ	468	41.5%	768	68.1%	16086	12215	46972	74.5%	**79.4%**	76.8%
	Lifetime	505	44.8%	773	68.5%	11974	13922	45265	79.1%	76.5%	**77.8%**
	SEM-SZZ	463	41.0%	621	55.1%	6257	21375	37812	85.8%	63.9%	73.2%
	TC-SZZ	195	17.3%	518	45.9%	24356	23719	35468	59.3%	59.9%	59.6%
	LLM4SZZ	459	40.7%	664	58.9%	9491	20281	38906	80.4%	65.7%	72.3%
	Sum	***2,596***	***38.4%***	***4120***	***60.9%***	***80,184***	***105,391***	***249,731***	***75.7%***	***70.3%***	***72.9%***
Matching-based	**ReDeBug**	417	37.0%	536	47.5%	3989	23289	35898	90.0%	60.7%	72.5%
	VUDDY	243	21.5%	296	26.2%	1227	39426	19761	**94.2%**	33.4%	49.3%
	MOVERY	374	33.2%	622	55.1%	11604	18642	40545	77.7%	68.5%	72.8%
	V1SCAN	326	28.9%	424	37.6%	2692	26719	32468	92.3%	54.9%	68.8%
	FIRE	406	36.0%	517	45.8%	4316	23236	35951	89.3%	60.7%	72.3%
	VULTURE	44	3.9%	622	55.1%	54889	28063	31124	36.2%	52.6%	42.9%
	Sum	***1,810***	***26.7%***	***3,017***	***44.6%***	***78,717***	***159,375***	***195,747***	***71.3%***	***55.1%***	***62.2%***

- **Version-level Evaluation.** Each version is evaluated independently to capture partial correctness and better reflect a tool's generalization ability. Tools may yield both FNs and FNs, reflecting different error tendencies. We adopt standard metrics at this level: *precision*, *recall*, and *F1-score*.

*2) **Results**:* Table III summarizes the effectiveness of each tool at both the vulnerability and version levels.

Overall Results. Among all tools, `VCCFinder` achieves the highest performance, with an accuracy of 44.9%, a No-Miss Ratio of 68.8%, and an F1-score of 77.8%. Only five tools achieve an accuracy of at least 40.0%, and only three tools reach a No-Miss Ratio above 60.0%. These results highlight a significant deficiency in the current methods' ability to accurately identify the complete set of affected versions. Even the accuracy of the best tool (i.e., `VCCFinder`) is less than half, which is far beyond a reliable usage in the practice. Moreover, all evaluated tools miss part of the affected versions for at least 30.0% of the vulnerabilities, which poses substantial security risks to downstream software leading to potentially unpatched vulnerabilities.

Finding to RQ1: Existing tools remain limited in accurately inferring affected versions. The best tool achieves under 50.0% accuracy at the vulnerability level, with over 30.0% of vulnerabilities missing at least one affected version.

Vulnerability-Level vs. Version-Level Discrepancy. We observe a consistent and significant gap between evaluation metrics at the vulnerability and version levels. Specifically, vulnerability-level accuracy is typically lower than the No-Miss Ratio, and both are often lower than version-level F1-scores. This discrepancy stems from fundamental differences in evaluation granularity and tolerance to partial correctness. Vulnerability-level accuracy demands an exact match with the full set of affected versions, making it highly sensitive to both false positives and false negatives. The No-Miss Ratio relaxes this by tolerating over-estimation but still penalizes any missed affected version. In contrast, version-level metrics evaluate correctness at the granularity of individual versions, offering a more nuanced and forgiving view that better aligns with practical usage scenarios. An exception is observed with `VULTURE`, where the NMR (55.1%) exceeds the F1-score (42.9%) due to its extremely high recall and low precision. This suggests that some tools prioritize broad coverage of potentially affected versions, achieving high recall and NMR at the cost of precision, thereby limiting their utility for precise vulnerability localization.

Finding to RQ1: Existing tools cannot precisely identify the full affected versions of vulnerabilities (with no version omissions or false positives). However, they achieve an higher F1-scores in approximating the affected versions.

Methodology-Level Comparison. Tracing-based tools tend to outperform matching-based ones across both evaluation levels, yielding higher summary accuracy (38.4% vs. 26.7%), No-Miss Ratio (60.9% vs. 44.6%), and F1-score (72.9% vs. 62.2%). Most tracing-based tools (except `TC-SZZ`) exceed 40.0% in accuracy and 55.0% in No-Miss Ratio—thresholds that are typically not reached by matching-based tools, which often prioritize precision over completeness. Version-level metrics reveal important exceptions. Matching-based tools such as `MOVERY` (72.8%), `ReDeBug` (72.5%), and `FIRE` (72.3%) achieve F1-scores comparable to those of tracing-based tools like `LLM4SZZ` (72.3%). In terms of precision, matching-based tools often achieve superior results—`VUDDY` and `V1SCAN` reach 94.2% and 92.3%, though typically at the cost of recall (e.g., `VUDDY`: 33.4%). By contrast, tools like `VULTURE` prioritize recall (52.6%) over precision (36.2%), reflecting a design bias toward completeness rather than correctness.

These disparities reflect fundamental methodological differences. Matching-based tools compare code at the block/function level using syntactic or hash-based similarity, achieving high precision but remaining fragile to refactorings and minor edits. Tracing-based tools, by contrast, exploit fine-grained historical signals (e.g., `git blame`) to capture broader temporal context, offering better recall and resilience to superficial changes, yet potentially missing semantic dependencies beyond modified lines. We analyze these trade-offs further in RQ3.

> **Finding to RQ1:** While tracing-based tools outperform matching-based ones on average, several matching-based tools achieve comparable F1 scores at the version level and often outperform in precision.

B. Root Cause Analysis (RQ2)

*1) **Setup**:* To understand the root causes of FPs and FNs in vulnerability-affected version identification, we adopt a mixed-method approach combining qualitative analysis and quantitative validation. We first systematically examine the key technical strategies used in tracing-based and matching-based methods, analyzing potential limitations at each stage. To validate these observations, we randomly sample 100 vulnerabilities from our dataset and evaluate identification results under representative strategies.

*2) **Analysis of Tracing-based Tools**:* Table IV summarizes the strategies adopted by representative tracing-based tools across four critical stages: *Statement Selection*, *Commit Tracing*, *Vulnerability-Inducing Commit Identification*, and *Affected Version Inference*.

a) Statement Selection: Most tracing-based tools (e.g., `V-SZZ`, `VCCFinder`, `Lifetime`, `TC-SZZ`) rely on simple patch-based heuristics to select tracing targets, typically focusing on deleted lines or those adjacent to additions. `V-SZZ` and `TC-SZZ` are even limited to patches with deletions. While these heuristics are straightforward and easy to implement, they often fail to capture the semantic core of the vulnerability. Our manual analysis of 100 representative vulnerabilities reveals two primary sources of error in this selection process. First, when patches span multiple functions or files, most existing tools treat all code hunks uniformly, without distinguishing between semantically relevant and irrelevant edits. The lack of granularity results in the inclusion of noisy, non-critical changes as tracing candidates, which weakens the signal used for version identification. Among the analyzed cases, 49 patches involve modifications across multiple functions or files, of which 16 contain such irrelevant hunks. Although `LLM4SZZ` try to address this issue using LLM for contextual filtering, it correctly excluded the noise in only 9 of these 16 cases—indicating limited success even with advanced semantic modeling.

Second, even in patches that modify only a single function, heuristic-based methods frequently fail to select the correct vulnerable statements. It is mainly due to their reliance on surface-level syntactic proximity rather than any principled understanding of the vulnerability's root cause. Consequently, statement selection remains brittle in the absence of deeper semantic analysis. Across the full set of 100 cases, these limitations led to incorrect statement selection in 49 instances. More sophisticated techniques such as semantic dependency analysis (`SEM-SZZ`) and LLM-based inference (`LLM4SZZ`) have been proposed to mitigate this issue. Nonetheless, these methods still incorrectly identified relevant code in 39 and 28 cases, respectively, underscoring persistent challenges in semantic reasoning and contextual disambiguation.

b) Commit Tracing: Most tools (e.g., `VCCFinder`, `Lifetime`, `SEM-SZZ`, `LLM4SZZ`) apply one-step backward tracing, which often misses earlier commits that introduced the vulnerability. Among the 100 cases, only 70 were traceable via a single step, whereas 30 required multi-step tracing. The strategy of `TC-SZZ` tracing back to the initial commit, however, led to more false positives, with version-level precision being only 59.3%, which significantly lower than `V-SZZ`'s 74.5%. Iterative tracing methods used in `V-SZZ` attempt to address this via similarity-based heuristics but still suffer from inaccuracies: 16 were over-traced and 12 were under-traced.

c) Vulnerability-Inducing Commit Identification: Once candidate commits are obtained, tools adopt varying heuristics to identify the final vulnerability-inducing commit, such as selecting the earliest or most frequently blamed commit. These heuristics, however, do not verify whether the selected commit actually introduced the vulnerability. `LLM4SZZ` try semantic verification by prompting an LLM to check vulnerability existence prior to a given commit. Nevertheless, our evaluation on 100 cases showed that `LLM4SZZ` still produced **12** FPs and **29** FNs, suggesting the difficulty of reliable semantic reasoning at the commit level.

d) Affected Version Inference: In multi-branch development environments, patches may be inconsistently propagated. Tools that only analyze the `main` branch (e.g., `V-SZZ`) risk missing relevant versions. Though cross-branch matching is supported, `V-SZZ`'s simple criteria for finding duplicated patches still failed to detect relevant patches in 13 out of 100 cases.

> **Finding to RQ2:** Tracing-based approaches are sensitive to the accuracy of identified vulnerability statements. While semantic analysis and LLMs enhance performance, key limitations persist. Moreover, their effectiveness is further constrained by the inaccuracy in selecting vulnerability-inducing commits and the strategy of commit tracing.

*3) **Analysis of Matching-based Tools**:* Table V summarizes the taxonomy of matching-based tools, which typically follow two phases: *vulnerability signature construction* and *signature matching*.

a) Vulnerability Signature Construction: Tools in this category extract syntactic or semantic features from known vulnerability patches to match similar instances in other versions. However, the granularity and flexibility of the extracted signatures significantly affect their effectiveness. Early tools such as `ReDebug` and `VUDDY` employ coarse-grained strategies: `ReDebug` builds sliding-window token sequences over patch hunks, while `VUDDY` uses the entire pre-patch function as a matching signature. These coarse-grained strategies are sensitive to unrelated code edits, resulting in

Table IV. Overview of the key stages and strategy variants adopted by tracing-based methods.

Stage	Strategy	VCCFinder	V-SZZ	Lifetime	SEM-SZZ	TC-SZZ	LLM4SZZ
S1: Statements Selection	Deleted or context lines(heuristic)	✓	✓	✓		✓	
	Data/control flow-based selection				✓		
	LLM-based selection						✓
S2: Commit Tracing	Single-step blame tracing	✓		✓	✓		✓
	Iterative tracing with similarity filter		✓				
	Iterative tracing until the initial commit					✓	
S3: Vulnerability-inducing Commit Selection	Earliest candidate commit		✓			✓	
	Most-blamed commit	✓		✓			
	Commit covering all target lines				✓		
	LLM-selected commit						✓
S4: Affected Versions Inference	With cross-branch patch reuse		✓				
	Without cross-branch patch reuse	✓		✓	✓	✓	✓

Table V. Overview of the key stages and strategy variants adopted by matching-based methods.

Stage	Strategy	ReDeBug	VUDDY	MOVERY	V1SCAN	FIRE	VULTURE
S1: Vulnerability Signature Construction	Pre-patch Context Signature	✓					
	Modified statement signature			✓	✓	✓	✓
	Entire pre-patch function signature		✓	✓	✓	✓	✓
	Post-patch function signature				✓	✓	✓
	Semantically related statements			✓		✓	
S2: Matching	Exact Matching	✓	✓	✓	✓	✓	✓
	Approximate Matching via LSH				✓		✓
	Approximate Matching via similarity metrics					✓	

frequent FNs. To improve specificity, `Movery` and `FIRE` extract vulnerability-relevant statements via semantic analysis. However, both tools apply uniform extraction rules regardless of patch semantics or vulnerability type. For example, `Movery` incorrectly extracted features in 47 cases revealing limitations in generalized semantic modeling.

Moreover, tools including `Movery`, `V1SCAN`, `FIRE`, and `VULTURE` further assume that deleted lines in the patch must be present in the target code for a match. This rigid assumption fails to accommodate variants due to refactoring or non-deletion-based repairs. In our dataset, this assumption led to 55.8% of deletable-line vulnerabilities being overlooked. Notably, these tools share a structural weakness with tracing-based methods: they treat all patch modifications as equally relevant. For patches spanning multiple functions or files, this strategy often leads to semantically unrelated edits being included in the extracted signatures, thereby increasing FPs.

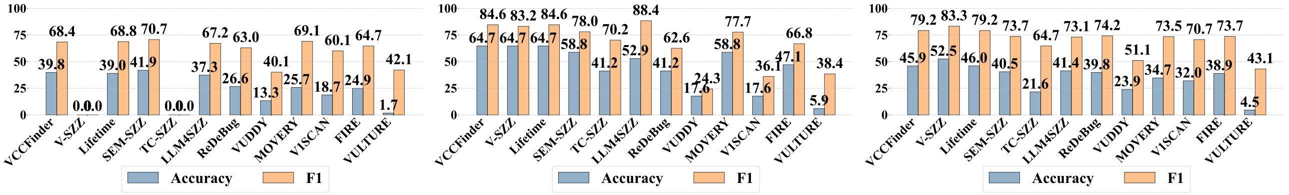
b) Signature Matching: Most tools perform *exact matching* between extracted features and target code, which is brittle and contributes to high FN rates. While tools (e.g., `V1SCAN`, `FIRE`, `VULTURE`) relax matching using structural or semantic similarity, none verify whether the matched code actually contains the vulnerability. Consequently, these tools either fail to identify semantically equivalent vulnerabilities or misidentify safe code as vulnerable. FPs also stem from flawed signature granularity. `VUDDY`'s function-level signatures, for instance, produce incorrect matches in two scenarios: (1) lack of structural normalization—if the patch modifies a function's structure (e.g., through reordering or renaming), matching may erroneously flag non-vulnerable code; (2) structural duplicates—older versions may contain functions with similar structure but semantically unrelated functions, which are wrongly matched due to lack of inter-procedural reasoning.

Finding to RQ2: The use of coarse-grained features, inflexible matching strategies, and insufficient semantic modeling limits the ability of matching-based methods to accurately identify vulnerability-affected versions.

C. Patch-Type Sensitivity Analysis (RQ3)

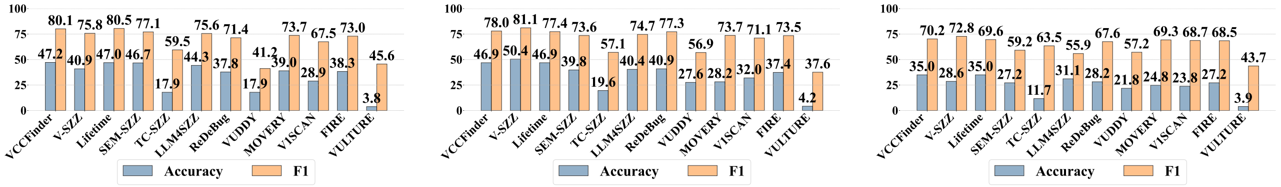
While RQ1 and RQ2 assess tool effectiveness from overall and stage-specific perspectives, they leave open the question of how structural properties of vulnerability patches influence performance. Thus, we analyze tool robustness across three patch-level dimensions, based on their direct alignment with the internal assumptions commonly made by existing tracing- and matching-based methods. The insights also support the design and evaluation of ensemble configurations in RQ4.

- **Type of Code Changes.** As observed in RQ2, most tools rely heavily on deleted lines for tracing. We therefore classify patches into *Add-only*, *Del-only*, and *Mixed* types, to evaluate sensitivity of this structural dependency.
- **Scope of Modifications.** Tools typically treat all modifications uniformly, regardless of how code changes are localized. However, real-world patches may cantain a single function,



(a) Added-only (b) Del-only (c) Mixed Patches

Figure 2. Impact of Patch Modification Types on Tool Performance (Add-only, Del-only, Mixed).



(a) Single-Function (b) Multi-Function Single-File (c) Multi-Function Multi-File

Figure 3. Tool Performance under Varying Patch Modification Scopes.

span multiple functions within a file, or modify code across files. This dimension helps assess each tool's resilience to dispersed or concentrated changes.

- **Cross-Branch Context.** In multi-branch development workflows, the same vulnerability may be patched differently across branches. Although V-SZZ incorporates cross-branch information when identifying affected versions, the general impact of single-branch versus multi-branch patches on tool performance has not been fully explored/determined. While V-SZZ incorporates cross-branch information when identifying affected versions, the general impact of single-branch versus multi-branch patches on tool performance remains underexplored.

*1) **Setup**:* Based on the above design, we evaluate tools along with these three patch dimensions to understand their structural robustness and uncover potential blind spots.

*2) **Results**:* **Impact of Type of Code Changes: Add-only, Del-only, Mixed.** Figure 2 shows tool performance across three patch types. Tracing-based tools perform well on *Del-only* and *Mixed* patches, but their performance drops sharply on *Add-only* patches, with version-level F1 decreasing by over 10 percentage points. This is because these tools rely on deleted lines to initiate blame tracing. Without deletions, tools like V-SZZ and TC-SZZ cannot operate. SEM-SZZ is more robust, as its dependency analysis allows itself to extract signals beyond simple line deletions. This observation reflects the semantics of patch content: deleted lines often contain the root cause, while added lines typically represent mitigation and offer limited cues for identifying the original vulnerability. Without deletions, tracing tools lack meaningful anchors to begin backward analysis.

Matching-based tools follow a different trend. They perform best on *Mixed* patches, where both added and deleted lines provide richer context for constructing semantic signatures. Performance declines on *Add-only* and *Del-only* patches due to reduced context. In particular, *Add-only* patches are more challenging, as they contain mitigation logic rather than faulty code, making semantic matching less effective.

Performance also varies by tool: SEM-SZZ performs best on *Add-only* patches, LLM4SZZ on *Del-only*, and V-SZZ on *Mixed* patches. This diversity highlights the complementarity of different approaches and motivates the ensemble configurations explored in RQ4.

Del-only patches are rare (1.3%), while *Add-only* and *Mixed* patches account for 21.3% and 77.4% of the dataset, respectively (Table II). Improving robustness on these dominant types—especially *Add-only*—is therefore essential for real-world applicability.

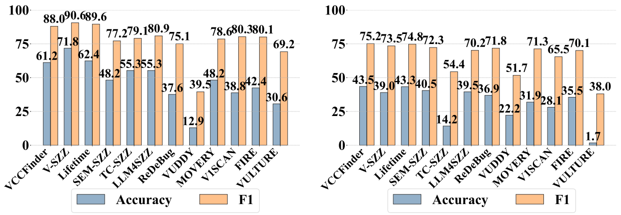
> **Finding to RQ3:** Tracing-based tools fail on *Add-only* patches due to their reliance on deletions. Matching-based tools perform better on *Mixed* patches, where both additions and deletions provide richer context.

Impact of Scope of Modification: Function-, File-, and Multi-File Changes. Figure 3 compares tool performance across different patch scopes. Both tracing-based and matching-based methods perform similarly on patches confined to a single function or multiple functions within the same file. However, their effectiveness drops significantly on *multi-file* patches. A closer inspection reveals that multi-file patches introduce substantially more non-vulnerability-related changes. In a sample of 50 patches from each category, multi-file patches (20) included significantly more non-vulnerability-related changes than single-file ones (7), adding noise and hindering accurate detection.

Among all tools, recall tends to increase while precision declines as the patch scope expands. This is due to coarse-grained analysis: tracing-based tools operate at the hunk level, while matching-based tools assess at the function level. A single matched function may cause an entire patch to be flagged, increasing recall but reducing precision.

In terms of performance, Lifetime achieves the highest F1 on *single-function* patches, while V-SZZ performs the best on both *single-file* and *multi-file* patches. The dataset includes 585 single-function (51.9%), 337 single-file (29.9%), and 206 multi-file patches (18.2%). Though multi-file patches are less frequent, they still constitute a substantial portion of real-world cases and remain a key challenge.

Finding to RQ3: Multi-file patches introduce more irrelevant changes, leading to notable performance drops across tools. Although less common, they pose a significant challenge and warrant focused attention.



(a) Single-Branch

(b) Multi-Branch

Figure 4. Performance of Different Tools under Single-Branch and Multi-Branch Projects.

Impact of Cross-Branch: Single-branch vs Multi-branch. Figure 4 summarizes tool performance under single-branch and multi-branch settings. In multi-branch scenarios most tools suffer marked performance declines, for instance, VULTURE's F1-score plunges by 31.2% and V-SZZ's accuracy drops 32.7%, highlighting the added complexity of identifying affected versions across diverging codebases. Among tracing-based tools, V-SZZ is the only one that incorporates cross-branch information during version inference. This strategy helps mitigate recall degradation in multi-branch settings. However, its multi-step tracing introduces noise from unrelated commits, which reduces precision and partially offsets the benefit. Other tracing tools rely solely on patches from the main branch and thus miss cross-branch vulnerabilities, leading to larger performance declines. Matching-based tools are more severely affected by multi-branch development due to increased structural variability across branches. Prior work [6] shows that only 9.37% of patches are syntactically identical across branches, limiting the effectiveness of exact or near-exact matching strategies used by tools such as MOVERY and FIRE. Overall, identifying affected versions in multi-branch projects remains a challenging task.

Table VI. Performance of Representative Tool Combinations under Three Strategies.

Strategy	Tool Combination	Vuln. Acc.	Version F1
Inclusion	LLM4SZZ+, ReDeBug	52.1%	84.0%
	LLM4SZZ+, V1SCAN, ReDeBug	51.7%	84.5%
Voting	V-SZZ, Lifetime, SEM-SZZ, Movery, LLM4SZZ+	**55.0%**	**84.8%**
Best-in-Dimension	SEM-SZZ, LLM4SZZ, V-SZZ	50.3%	81.7%
Reference: Best Individual Tools			
VCCFinder (standalone)		44.9%	77.8%
LLM4SZZ+ (modular hybrid)		50.7%	81.8%

Finding to RQ3: Multi-branch development significantly degrades the effectiveness of both tracing-based and matching-based tools, due to limited cross-branch patch utilization and substantial code divergence across branches.

D. Tool Combination Analysis (RQ4)

Based on earlier findings, this RQ investigates whether combining tool components or outputs can yield performance improvements over standalone tools. We explore this from two angles: modular recomposition of tracing-based tools and ensemble strategies spanning tracing- and matching-based tools.

*1) **Setup**:* Our evaluation is organized into two phases:

Phase-1: Modular Recomposition of Tracing-based Tools. We focus on tracing-based tools because their modular workflows are amenable to recombination, whereas the core stages of matching-based tools are often tightly coupled and less separable. Based on the stage-wise decomposition in RQ2, we identify four stages: (S1) statement selection, (S2) impact range inference, (S3) commit tracing, and (S4) cross-branch patch reuse. Prior results show that LLM-based methods in S1 and patch propagation in S4 consistently outperform alternatives. Fixing these two stages, we systematically explore combinations of 2 alternatives in S2 and 4 in S3, yielding 7 hybrid configurations ($2 \times 4 - 1 = 7$). The best variant (LLM4SZZ+) serves as a representative for Phase-2.

Phase-2: Cross-Tool Combination. This phase investigates whether outputs from diverse tools—spanning both tracing- and matching-based paradigms—can be effectively integrated. We select ten high-performing tools (F1-score $\geq$ 70%): VCCFinder, V-SZZ, Lifetime, SEM-SZZ, LLM4SZZ, ReDebug, Movery, FIRE, V1SCAN, and the newly derived LLM4SZZ+. We evaluate three ensemble strategies as follows:

- **Inclusion Strategy.** The cumulative effect of integrating tools is evaluated by testing sizes 2 to 10 union sets.
- **Voting Strategy.** To assess consensus-based robustness, we evaluate all combinations of 3, 5, 7, and 9 tools, marking a version as affected if the majority agrees.
- **Best-in-Dimension Strategy.** We also select the best-performing tool in each of four key dimensions identified in RQ3 (e.g., patch modeling, commit tracing) and aggregate their outputs, leveraging their complementary strengths.

*2) **Results:*** Table VI summarizes the top-performing configurations under each combination strategy.

Modular Recomposition. The best configuration, `LLM4SZZ+`, integrates LLM-based statement selection (S1), single-step blame tracing (S2), full-line coverage for commit tracing (S3), and cross-branch patch reuse (S4). This variant achieves 50.7% accuracy at the vulnerability level and 81.8% F1-score at the version level—outperforming the strongest tracing-based baseline (`VCCFinder`) by 5.8% and 4.0%, and the earlier hybrid version (`LLM4SZZ`) by 10.0% and 9.5%. Notably, replacing S3 component with LLM-selected commits significantly reduces performance (-7.3% accuracy, -5.7% F1), suggesting that though LLMs enhance early-stage selection, heuristic-based commit tracing remains more effective.

Finding to RQ4: Modular recomposition yields great improvement. LLMs are best at statement selection, whereas commit identification benefits more from heuristic methods.

Inclusion Strategy. The combination of `ReDebug` and `LLM4SZZ+` achieves the highest accuracy at the vulnerability level (52.1%), an improvement of 7.2% over `VCCFinder` and 1.4% over `LLM4SZZ+`. At the version level, it reaches an F1-score of 84.0%. Another configuration—`ReDebug`, `V1SCAN`, and `LLM4SZZ+`—achieves the highest version-level F1-score (84.5%), outperforming `VCCFinder` and `LLM4SZZ+` by 6.7% and 2.7%, with a vulnerability-level accuracy of 51.7%.

Voting Strategy. The optimal combination includes `V-SZZ`, `Lifetime`, `SEM-SZZ`, `Movery`, and `LLM4SZZ+`. This ensemble achieves 55.0% accuracy at the vulnerability level and 84.8% F1-score at the version level—improvements of 10.1% and 7.0% over `VCCFinder`, and 4.3% and 3.0% over `LLM4SZZ+`. The results highlight the effectiveness of majority voting in mitigating individual tool weaknesses.

Best-in-Dimension Strategy. Focusing on patch modification types (Add-only, Del-only, Mixed), the combination of `SEM-SZZ`, `LLM4SZZ`, and `V-SZZ` achieves 50.3% accuracy at the vulnerability level and 81.7% F1-score at the version level—surpassing `VCCFinder` by 5.4% and 3.9%. While marginally below `LLM4SZZ+`, it shows the value of aligning tool selection with orthogonal performance dimensions.

In summary, the experimental results across all three strategies consistently demonstrate the benefits of tool combination. First, ensemble methods provide substantial improvements over the best standalone and hybrid tools, confirming the overall effectiveness of tool integration. Second, the top-performing combinations in the Inclusion and Voting strategies consistently integrate both tracing-based and matching-based approaches, validating their complementary strengths. Third, `LLM4SZZ+` is included in all leading combinations, underscoring its robustness as a foundation for ensemble configurations. However, the best vulnerability-level accuracy remains below 60%, reflecting fundamental limits in existing tool architectures—such as reliance on heuristics, insufficient semantic modeling, and coarse-grained decision logic.

Finding to RQ4: Ensemble strategies significantly improve performance over individual tools, yet the bottleneck highlights common architectural limitations that remain unresolved.

V. Discussion

A. Challenges and Implications

Our study reveals that existing tools for vulnerability-affected version identification suffer from substantial limitations across three dimensions: ❶ noisy and coarse-grained patches, ❷ semantic mismatch between fix locations and root causes, and ❸ shallow presence verification. These issues span diff-based heuristics (e.g., `V-SZZ`), semantic analysis (e.g., `MOVERY`), and LLM-based approaches (e.g., `LLM4SZZ`).

Noisy patches. Most tools assume that any changes in a vulnerability patch are relevant. However, patches frequently contain unrelated edits, such as refactoring or multi-issue fixes, which impair downstream analysis. Our manual inspection indicates that 19% of patches contain unrelated changes. Despite this, few tools perform effective preprocessing to filter noise. Although `LLM4SZZ` attempts to isolate relevant edits using large language models, its failure rate remains 35%, highlighting this task's difficulty.

Semantic mismatch. A more fundamental limitation arises from the disconnect between patch locations and vulnerability root causes. Certain vulnerability types (e.g., *double free*, *use-after-free*) involve complex inter-procedural interactions, with contributing code located in different functions or files. Even within a single function, control-flow-preserving changes (e.g., replacing `return` with `goto` for cleanup) can lead to incorrect identification when tools lack flow sensitivity. In our study, `SEM-SZZ` and `MOVERY` yielded 39 and 47 false positives, respectively, when identifying vulnerability-related statements across 100 samples.

Simplistic presence verification. Existing tools often rely on tracing heuristics or feature-matching strategies to determine whether a version is vulnerable. Tracing-based methods are prone to misidentifying vulnerability-introducing commits, while feature-matching approaches focus on syntactic similarity without verifying whether the vulnerability condition persists. The absence of root-cause-aware reasoning leads to both high false-positive and false-negative rates.

B. Practical Workarounds via Tool Combination

Given that no individual tool achieves more than 44.9% accuracy or 77.8% F1-score, we explore ensemble strategies as practical alternatives. A majority-voting approach across five tools improves accuracy and F1-score by 10.1% and 7.0%, respectively, over the best standalone method. The finding demonstrates that tools offer complementary strengths, and that combination helps offset single tool's limitations. Though not a substitute for deeper technical advances, such ensembles provide a viable interim solution in practice.

C. Future Directions

To address the above limitations more fundamentally, we highlight three promising research directions:

1) Patch preprocessing. Future approaches should move beyond treating patches as atomic units. Combining static analysis with LLMs could enable finer-grained filtering of unrelated edits. Patch modularization—partitioning edits by functional or semantic unit—may further improve the mapping between fixes and individual vulnerabilities.

2) Root cause localization. While root cause analysis remains difficult when limited to patches, progress can be made by designing analysis strategies tailored to specific vulnerability types and remediation patterns. Additionally, prompting LLMs with structured expert knowledge—e.g., through chain-of-thought reasoning or multi-role dialogue—may help elicit more accurate vulnerability semantics.

3) Presence verification. As code evolves, the boundary between fixed and vulnerable states becomes blurred. We advocate for verification strategies grounded in vulnerability semantics, ideally aligned with identified root causes. Few-shot prompting, multi-agent validation, and root-cause-aware reasoning are promising techniques to improve robustness and reduce both false positives and false negatives.

D. Threats to Validity

1) ***External Validity****:* Our study focuses on C/C++ open-source projects, which may limit the generalizability to other languages and ecosystems. Nonetheless, C/C++ is central to memory-related vulnerabilities, the main focus of existing tools. To ensure coverage, we selected nine actively maintained projects from diverse domains (e.g., OSs, databases, web servers). While our dataset includes 1,128 vulnerabilities, it may underrepresent rare or poorly documented types. We mitigated this by validating coverage against the CWE Top 25, though long-tail cases may still be missing.

2) ***Internal Validity****:* Manual labeling of ground truth may introduce error, but it is necessary due to the absence of standard benchmarks. We mitigated this via double annotation by experienced reviewers, with conflicts resolved by a third expert. While tool selection may miss obscure methods, we combined comprehensive keyword searches with multi-round snowballing, covering all major technique categories and emphasizing tools with public implementations. Tools were run with default configurations, which may affect fairness; however, these typically reflect optimal settings as reported. Lastly, manual FP/FN inspection may involve bias; to reduce this, we used standardized criteria and cross-checking, with minimal disagreements resolved through discussion.

VI. Conclusion

We conducted the first comprehensive empirical study on vulnerability-affected versions identification, evaluating 12 tools across 1,128 real-world C/C++ vulnerabilities. Our results show that no tool achieves over 45.0% accuracy, largely due to heuristic reliance, limited semantic understanding, and inflexible matching strategies. Even with ensemble methods, accuracy remains below 60.0%, indicating fundamental limitations in existing designs. Our benchmark and analysis provide practical guidance for tool development and establish a foundation for future work toward more accurate and resilient approaches.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable feedback. This work is partly supported by National Key R&D Program of China under Grant #2022YFB3103901, Chinese National Natural Science Foundation (Grants #62202462, #62302500, #62032010). This research is also partly supported by the National Research Foundation, Singapore, and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-GC-2023-008); by the National Research Foundation Singapore and the Cyber Security Agency under the National Cybersecurity R&D Programme (NCRP25-P04-TAICeN); and by the Prime Minister's Office, Singapore under the Campus for Research Excellence and Technological Enterprise (CREATE) programme. Any opinions, findings and conclusions, or recommendations expressed in these materials are those of the author(s) and do not reflect the views of National Research Foundation, Singapore, Cyber Security Agency of Singapore, Singapore.

References

[1] (2025) Vulnerable version study artifact. Research artifact. [Online]. Available: https://sites.google.com/view/vulnerable-version-study

[2] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying open-source license violation and 1-day security risk at large scale," *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[3] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "Libvdiff: Library version difference guided oss version identification in binaries," *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 791–802, 2024.

[4] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, and R. A. Beyah, "One bad apple spoils the barrel: Understanding the security risks introduced by third-party components in iot firmware," *IEEE Transactions on Dependable and Secure Computing*, vol. 21, pp. 1372–1389, 2022.

[5] ——, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in iot firmware," *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2022.

[6] S. Yang, Y. Xiao, Z. Xu, C. Sun, C. Ji, and Y. Zhang, "Enhancing oss patch backporting with semantics," *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023.

[7] R. Shariffdeen, X. Gao, G. J. Duck, S. H. Tan, J. L. Lawall, and A. Roychoudhury, "Automated patch backporting in linux (experience paper)," *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[8] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pp. 672–684, 2022.

[9] L. Zhang, C. Liu, S. Chen, Z. Xu, L. Fan, L. Zhao, Y. Zhang, and Y. Liu, "Mitigating persistence of open-source vulnerabilities in maven ecosystem," *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 191–203, 2023.

[10] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem," *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1046–1058, 2023.

[11] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *USENIX Security Symposium*, 2019.

[12] A. Anwar, A. A. Abusnaina, S. Chen, F. H. Li, and D. A. Mohaisen, “Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses,” *IEEE Transactions on Dependable and Secure Computing*, vol. 19, pp. 4255–4269, 2020.

[13] S. Wu, R. Wang, K. Huang, Y. Cao, W. Song, Z. Zhou, Y. Huang, B. Chen, and X. Peng, “Vision: Identifying affected library versions for open source software vulnerabilities,” *2024 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1447–1459, 2024.

[14] J. Zhou, M. Pacheco, J. Chen, X. Hu, X. Xia, D. Lo, and A. E. Hassan, “Colefunda: Explainable silent vulnerability fix identification,” *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 2565–2577, 2023.

[15] J. Zhou, M. Pacheco, Z. Wan, X. Xia, D. Lo, Y. Wang, and A. E. Hassan, “Finding a needle in a haystack: Automated mining of silent vulnerability fixes,” *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 705–716, 2021.

[16] C. Luo, W. Meng, and S. Wang, “Strengthening supply chain security with fine-grained safe patch identification,” *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 1084–1095, 2024.

[17] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, “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*, ser. CCS ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 426–437.

[18] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser, “How long do vulnerabilities live in the code? a Large-Scale empirical measurement study on FOSS vulnerability lifetimes,” in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 359–376.

[19] L. Bao, X. Xia, A. E. Hassan, and X. Yang, “V-szz: Automatic identification of version ranges affected by cve vulnerabilities,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 2022, pp. 2352–2364.

[20] L. Tang, C. Ni, Q. Huang, and L. Bao, “Enhancing bug-inducing commit identification: A fine-grained semantic analysis approach,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 11, p. 3037–3052, Nov. 2024.

[21] Y. Cheng, L. K. Shar, T. Zhang, S. Yang, C. Dong, D. Lo, S. Lv, Z. Shi, and L. Sun, “Llm-enhanced static analysis for precise identification of vulnerable oss versions,” *arXiv preprint arXiv:2408.07321*, 2024.

[22] L. Tang, J. Liu, Z. Liu, X. Yang, and L. Bao, “Llm4szz: Enhancing szz algorithm with context-enhanced assessment on large language models,” *arXiv preprint arXiv:2504.01404*, 2025.

[23] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.

[24] Q. Sun, L. Xu, Y. Xiao, F. Li, H. Su, Y. Liu, H. Huang, and W. Huo, “Verjava: Vulnerable version identification for java oss with a two-stage analysis,” in *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2022, pp. 329–339.

[25] Y.-Q. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang, “Precise (un)affected version analysis for web vulnerabilities,” *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.

[26] S. Woo, H. Hong, E. Choi, and H. Lee, “{MOVERY}: A precise approach for modified vulnerable code clone discovery from modified {Open-Source} software components,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3037–3053.

[27] S. Woo, E. Choi, H. Lee, and H. Oh, “{V1SCAN}: Discovering 1-day vulnerabilities in reused {C/C++} open-source software components using code classification techniques,” in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 6541–6556.

[28] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, “MVP: Detecting vulnerabilities using Patch-Enhanced vulnerability signatures,” in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1165–1182.

[29] S. Kim, S. Woo, H. Lee, and H. Oh, “Vuddy: A scalable approach for vulnerable code clone discovery,” in *2017 IEEE symposium on security and privacy (SP)*. IEEE, 2017, pp. 595–614.

[30] M. Wen, R. Wu, Y. Liu, Y. Tian, X. Xie, S.-C. Cheung, and Z. Su, “Exploring and exploiting the correlations between bug-inducing and bug-fixing commits,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 326–337.

[31] G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto, “Evaluating szz implementations through a developer-informed oracle,” *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 436–447, 2021.

[32] Y. Lyu, H. J. Kang, R. Widyasari, J. Lawall, and D. Lo, “Evaluating szz implementations: An empirical study on the linux kernel,” *IEEE Trans. Softw. Eng.*, vol. 50, no. 9, p. 2219–2239, Sep. 2024.

[33] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, “Automatic identification of bug-introducing changes,” *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, pp. 81–90, 2006.

[34] D. A. Da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, “A framework for evaluating the results of the szz approach for identifying bug-introducing changes,” *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2016.

[35] E. C. Neto, D. A. Da Costa, and U. Kulesza, “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)*. IEEE, 2018, pp. 380–390.

[36] P. Bludau and A. Pretschner, “Pr-szz: How pull requests can support the tracing of defects in software repositories,” in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 1–12.

[37] L. Tang, L. Bao, X. Xia, and Z. Huang, “Neural szz algorithm,” in *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’23. IEEE Press, 2024, p. 1024–1035.

[38] J. Jang, A. Agrawal, and D. Brumley, “Redebug: finding unpatched code clones in entire os distributions,” in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 48–62.

[39] H. Li, H. Kwon, J. Kwon, and H. Lee, “A scalable approach for vulnerability discovery based on security patches,” in *Applications and Techniques in Information Security: 5th International Conference, ATIS 2014, Melbourne, VIC, Australia, November 26-28, 2014. Proceedings 5*. Springer, 2014, pp. 109–122.

[40] T. Luo, C. Ni, Q. Han, M. Yang, J. Wu, and Y. Wu, “Poster: Patchgen: Towards automated patch detection and generation for 1-day vulnerabilities,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 1656–1658.

[41] W. Kang, B. Son, and K. Heo, “Tracer: Signature-based static analysis for detecting recurring vulnerabilities,” in *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1695–1708.

[42] S. Feng, Y. Wu, W. Xue, S. Pan, D. Zou, Y. Liu, and H. Jin, “{FIRE}: Combining {Multi-Stage} filtering with taint analysis for scalable recurring vulnerability detection,” in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 1867–1884.

[43] S. Xu, J. Dong, W. Cai, J. Li, A. Shaghaghi, N. Sun, and S. Ma, “Enhancing security in third-party library reuse–comprehensive detection of 1-day vulnerability through code patch analysis,” *Network and Distributed System Security (NDSS) Symposium*, 2025.

[44] K. Huang, C. Lu, Y. Cao, B. Chen, and X. Peng, “Vmud: Detecting recurring vulnerabilities with multiple fixing functions via function selection and semantic equivalent statement matching,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS ’24. Association for Computing Machinery, 2024, p. 3958–3972.

[45] L. Lin, J. Ye, C. Wang, and R. Wu, “Paten: Identifying unpatched third-party apis via fine-grained patch-enhanced ast-level signature,” *IEEE Transactions on Software Engineering*, vol. 51, no. 4, pp. 990–1006, 2025.

[46] S. Wi, S. Woo, J. J. Whang, and S. Son, “Hiddencpg: Large-scale vulnerable clone detection using subgraph isomorphism of code property graphs,” in *Proceedings of the ACM Web Conference 2022*, ser. WWW ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 755–766.

[47] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, “Facilitating vulnerability assessment through poc migration,” *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

[48] Z. Zhang, Y. Hao, W. Chen, X. Zou, X. Li, H. Li, Y. Zhai, Z. Qian, and B. Lau, “Symbisect: Accurate bisection for fuzzer-exposed vulnerabilities,” in *USENIX Security Symposium*, 2024.

[49] C. Rezk, Y. Kamei, and S. McIntosh, “The ghost commit problem when identifying fix-inducing changes: An empirical study of apache projects,” *IEEE Transactions on Software Engineering*, vol. 48, no. 9, pp. 3297–3309, 2022.

[50] L. Pellegrini, V. Lenarduzzi, and D. Taibi, “Openszz: a free, open-source, web-accessible implementation of the szz algorithm,” in *Proceedings of the 28th International Conference on Program Comprehension*, 2019, pp. 446–450.

[51] Atlassian, “Jira — issue & project tracking software — atlassian,” 2025, [Online; accessed 2025-05-31]. [Online]. Available: https://www.atlassian.com/software/jira

[52] Y. He, Y. Wang, S. Zhu, W. Wang, Y. Zhang, Q. Li, and A. Yu, “Automatically identifying cve affected versions with patches and developer logs,” *IEEE Transactions on Dependable and Secure Computing*, vol. 21, no. 2, pp. 905–919, 2024.