On the Quality of Vulnerability-Contributing Commit Datasets: A Systematic Review and Exploit-Based Evaluation

Vinicius Carvalho Lopes
University of Notre Dame, Notre Dame, IN, USA

Matthew Zampino
University of Notre Dame, Notre Dame, IN, USA

Christian Garcia
University of Notre Dame, Notre Dame, IN, USA

Joanna C. S. Santos
University of Notre Dame, Notre Dame, IN, USA

Adriana Sejfia
University of Edinburgh, Edinburgh, UK

Abstract

Background: Vulnerability-Contributing Commits (VCCs) are the code changes at which a software system turns from safe to unsafe with respect to a specific vulnerability. Accurate VCC identification is crucial for training vulnerability detection models, yet the reliability of existing VCC datasets remains underexplored. If these VCC datasets contain mislabeled commits, models trained on them may learn incorrect patterns.

Aims: We present a two-part study of VCC dataset quality: characterizing how VCCs are defined and identified across the literature, and empirically measuring the accuracy of existing VCC datasets.

Method: We review 38 papers and, based on their findings, propose an operational VCC definition based on security state transitions and testable through parent commit validation. We apply this definition in an exploit-based validation study on 6 datasets aggregating 3,105 VCC samples, of which 168 (5.41%) have publicly documented exploits enabling empirical validation.

Results: We found terminological and definitional inconsistencies: 8 distinct terms with varying definitions, 15.8% of papers lacking explicit definitions, and confusion between commits that *contain* versus commits that *introduce* vulnerabilities. Among the 168 validated samples, we observe a 79.2% false positive rate, rising to 88.4% in the subset whose verdicts rest entirely on runtime exploit reproduction, with 47.0% being commits labeled as VCCs although the vulnerability was already exploitable in the parent commit.

Conclusions: Existing VCC datasets contain labeling errors detectable through exploit-based parent commit validation. Our findings call for standardized VCC definitions and parent commit validation to distinguish VCC commits from those that merely modify already-vulnerable code.

2012 ACM Subject Classification Security and privacy → Software security engineering

Keywords and phrases vulnerability-contributing commits, software security, dataset quality, systematic literature review, exploit validation

Digital Object Identifier 10.4230/LIPIcs.ESEM.2026.95

Category Technical Track Paper

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.21904512`

20th International Symposium on Empirical Software Engineering and Measurement (ESEM 2026).
Editors: Robert Feldt, Maria Paasivaara, Daniel Mendez, Stefan Wagner, and Marvin Muñoz Barón; Article No. 95; pp. 95:1–95:20

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Software security remains a persistent concern as applications become increasingly integrated into critical infrastructure and daily life [33]. Software vulnerabilities, which are defects that can be exploited to violate security policies [48], can lead to serious consequences ranging from system crashes to sensitive data breaches. Prior works have developed various techniques to address this challenge, including code reviews [22], automated static analysis [60], fuzz testing [6, 29], and machine learning (ML) approaches for vulnerability detection [38, 50].

A promising direction in this field focuses on detecting vulnerabilities at the commit level. A *Vulnerability-Contributing Commit* (VCC) is commonly described as a code change that introduces a vulnerability into a software system [27, 40]. Identifying VCCs when they occur enables developers to address security issues while the code context is still fresh, rather than discovering vulnerabilities months or years later after exploitation [38, 71].

While ML models have shown promise for VCC detection [38, 45, 50], they depend heavily on the *training data quality*. If VCC datasets contain mislabeled commits or samples that cannot be validated due to obsolete dependencies and missing build documentation, models trained on them may learn incorrect patterns, leading to poor real-world performance. Without build documentation to reproduce the project's historical state at the labeled commit, mislabeled samples cannot be empirically detected before entering training, and models risk learning incidental commit features rather than security-relevant changes.

Several challenges complicate VCC dataset construction. First, tracing a vulnerability back to the commit at which it first became exploitable is inherently difficult. The widely used SZZ algorithm [62] and its variants [2, 5, 7, 36, 51, 53, 65] rely on `git blame` heuristics that can produce incorrect results when code has been refactored or when multiple commits contribute to a vulnerability [25, 27]. For example, if a vulnerable function was introduced in commit A and later reformatted in commit B, SZZ traces the eventual fix back to B, blaming a cosmetic change rather than the commit where the vulnerability first appeared. Second, there is no consensus on what precisely constitutes a VCC: existing literature uses different terminology and definitions [35, 40, 45, 50], making it difficult to compare datasets or combine them. Third, validating whether a vulnerability actually becomes exploitable at a labeled VCC requires reproducing historical software environments, which is often difficult due to missing dependencies and incomplete documentation [27].

These dataset construction challenges create three research gaps. *First*, while prior work has acknowledged that VCC definitions vary [27], this inconsistency has not been systematically characterized, leaving open whether datasets constructed under different definitions are compatible or whether models trained on one generalize to others. *Second*, despite known limitations of labeling approaches (*e.g.*, SZZ), the actual false positive rates in VCC datasets remain unmeasured, so ML models may be trained on data of unknown quality with performance metrics that do not reflect real-world effectiveness. *Third*, although researchers recognize the need to distinguish vulnerability-introducing commits from those that merely modify existing vulnerable code [27], no operational validation methodology exists, so SZZ-style labels are treated as ground truth and over-attribution remains undetected.

To address these gaps, this work includes a two-part study. To address the first gap, we conduct a *systematic literature review (SLR)* to investigate ***how VCCs are defined and termed across the literature***, quantifying the extent of terminological fragmentation, and synthesize a precise VCC definition that provides a standard criterion against which datasets can be compared. To address the second and third gaps, we apply our definition through an ***exploit-based validation study***. Using publicly available exploits from Exploit-DB [47], we reproduce vulnerabilities in Docker containers that recreate historical software states.

For each successfully exploited vulnerability, we perform *parent commit validation*, testing whether the same exploit succeeds on the parent commit. This provides a rigorous verification procedure grounded in runtime analysis and empirically measures false positive rates in existing datasets. Rather than replacing SZZ-based candidate identification, we provide a complementary verification method for filtering candidates and establishing reliable ground truth. Therefore, this work makes the following contributions:

- A systematic literature review of 38 papers (2005–2025) examining VCC terminology, definitions, challenges, and dataset construction practices (**Section 4**).
- An *operational* definition of VCC grounded in security state transitions, together with recommendations for improving dataset construction practices. Through this definition, we developed a novel validation methodology incorporating parent commit testing to verify that the exploitability transition occurs at the labeled commit (**Section 5**).
- An empirical validation of 3,105 VCC samples aggregated from 6 datasets, revealing a 79.2% false positive rate among the 168 samples with documented exploits (5.41% of 3,105; see § 6.2), of which 47.0% are commits labeled as VCCs although the vulnerability was already exploitable in the parent commit (**Section 6**).

2 Background & Related Work

2.1 Vulnerabilities & Vulnerability Contributing-Commits (VCCs)

Software vulnerabilities are caused by defects that can be exploited by attackers to negatively impact the software system's confidentiality, integrity, or availability [41]. These vulnerabilities may be assigned a **CVE ID** (Common Vulnerabilities and Exposures identifier), which uniquely identifies a publicly disclosed vulnerability in the CVE catalog [46]. Since different vulnerabilities can share a similar *root cause*, *consequence*, or *fix pattern* [42], they can also be assigned a vulnerability type, *i.e.*, a **CWE ID** from the Common Weakness Enumeration catalog [41]. These IDs are used as a common vocabulary to refer to a specific security incident (CVE ID) or a group of vulnerabilities (CWE ID).

A **Vulnerability-Contributing Commit (VCC)** is a term used in the literature to refer to a commit that introduces a vulnerability in software, *i.e.*, a defective code change that contributes to the root cause of a vulnerability [8, 27, 40]. These commits may also be referred to as *"vulnerable code changes"* [8,9], *"vulnerability-inducing commits"*, *"vulnerability-introducing commits"*, or *"fix-inducing changes"* [2, 10, 19], although some researchers prefer *"contributing"* over *"inducing"* [39,40]. In this work, we use **VCC** as the umbrella term for the remainder of this paper; characterizing the term precisely is itself an open problem, which we discuss in § 4 and resolve with a formal, operationalizable definition in § 5.

An example of a VCC obtained from the Vulnerability History project [39] is shown in Listing 1. This vulnerability (CVE-2010-4476) is caused by the use of the `Double.parseDouble()` method. This method (in an earlier JRE version) could be exploited by attackers to conduct a denial of service (DoS) attack through a String (*e.g.*, `"2.2250738585072012e-308"`) that triggered an infinite loop while converting this string to a floating point number [46]. This vulnerability was introduced in the first commit of this file (revision **302975**).

2.2 Automated VCC Detection

Prior works have studied the characteristics of VCCs [8, 9, 40] and ways to automatically detect them [50]. **Just-in-time (JIT) vulnerability detection** is a term commonly used to refer to approaches that detect vulnerabilities present in a commit being pushed into a code

java/org/apache/catalina/connector/Request.java

```
2455 +           // Extract the quality factor for this entry
2456 +           double quality = 1.0;
2457 +           int semi = entry.indexOf(";q=");
2458 +           if (semi >= 0) {
2459 +               try {
2460 +                   quality = Double.parseDouble(entry.substring(semi + 3));
2461 +               } catch (NumberFormatException e) {
2462 +                   quality = 0.0;
2463 +               }
2464 +               entry = entry.substring(0, semi);
2465 +           }
```

Listing 1 VCC example from the Apache Tomcat project (CVE-2010-4476)

repository [38, 71]. VCC detection's main advantage is giving developers immediate feedback, helping them consider ways to improve the code's security while having the code change context still fresh in mind [38]. However, there are two main challenges in VCC detection. First, a commit is an *incomplete* part of a program, so VCC detection must reason about its impact on the program as a whole. Second, a commit may include *casualty changes* [59], *i.e.*, unrelated changes that do not affect the security of the program, introducing noise during VCC detection. The difficulty is the entanglement of casualty changes with security-relevant edits in the same diff, *e.g.*, a method rename can touch dozens of lines without changing behavior, while a single deleted check introduces a vulnerability. Distinguishing them requires reasoning about program semantics rather than the surface diff.

2.3 VCC Dataset Construction

The SZZ algorithm [62], the primary approach for VCC dataset construction, uses `git blame` to trace fixed lines back to their last-modifying commits. Numerous SZZ variants [2, 5, 7, 36, 51, 53, 65] have been proposed to improve accuracy: Kim *et al.* [31] introduced annotation graphs, and Costa *et al.* [20] developed a framework for systematic evaluation. However, studies reveal limitations in SZZ-based construction. Rosa *et al.* [55, 56] found accuracy variations across implementations, while Hinrichs *et al.* [26] assessed 12 VCC mining techniques for Java, finding none sufficiently accurate.

For security-specific datasets, Perl *et al.* [50] created the VCCFinder dataset by manually linking CVEs to VCCs. Paul *et al.* [49] constructed a Chromium VCC dataset using SZZ-based tracing from security patches. Riom *et al.* [54] revisited VCCFinder with updated methodology, while Coskun *et al.* [19] profiled developer characteristics associated with vulnerability introduction. Recently, Cannavale *et al.* [12] demonstrated that the choice of SZZ variant significantly impacts JIT vulnerability prediction performance, and Cheng *et al.* [17] proposed VERCATION for vulnerable version identification using static analysis and LLMs. However, these datasets rely on either automated SZZ labeling (which may produce false positives) or limited manual verification. Our work provides the first systematic exploit-based validation quantifying the actual false positive rates in these datasets.

2.4 Dataset Quality Assessment

Prior work has examined the limitations of SZZ-based bug-inducing commit identification and challenges in VCC datasets. Herbold *et al.* [25] analyzed how tangled commits and feature interactions introduce SZZ errors, while Rosa *et al.* [55] compared SZZ implementations against developer-informed ground truth and reported precision ranging from 48% to 67%. In the vulnerability domain, Lomio *et al.* [38] surveyed JIT vulnerability detection approaches but did not empirically validate the underlying datasets, Bao *et al.* [5] proposed V-SZZ for identifying affected vulnerability version ranges rather than validating VCC accuracy, and Nguyen *et al.* [43] introduced a unified evaluation framework for JIT vulnerability prediction models while focusing on model comparison instead of dataset validation. Unlike these prior

works, we introduce an *exploit-based validation approach*: we reconstruct historical software states and verify whether the vulnerability actually becomes exploitable at labeled VCCs. Specifically, we test parent commits to distinguish commits at which a vulnerability first becomes *exploitable* from those at which it was *already exploitable*, a distinction that prior dataset construction approaches have not systematically operationalized.

3 Research Questions (RQs)

RQ1 ***What terminology and definitions are used in the literature to characterize VCCs?***

Before evaluating VCC datasets or detection techniques, we must understand how prior work conceptualizes VCCs. Inconsistent terminology and definitions can yield incompatible datasets and undermine downstream tools. Thus, we catalog terminology and definition patterns to identify sources of confusion and areas needing standardization.

RQ2 ***What are the key challenges in identifying and labeling VCCs?***

Accurate VCC identification is crucial for building reliable training data and for understanding vulnerability lifecycles. While prior work [27] has noted challenges in VCC identification, these have not been systematically characterized. Understanding them can help explain errors in existing datasets and guide future methodological improvements.

RQ3 ***How accurate are existing VCC datasets when subjected to exploit-based validation?***

VCC datasets are widely used to train and evaluate vulnerability detection tools, yet their accuracy is often assumed rather than validated. Mislabeled commits can propagate errors to downstream models. We use exploit-based validation of security state transitions to provide empirical evidence of dataset quality beyond metadata-based heuristics.

Fig. 1 shows an overview of our two-part study, which is detailed in the next sections.

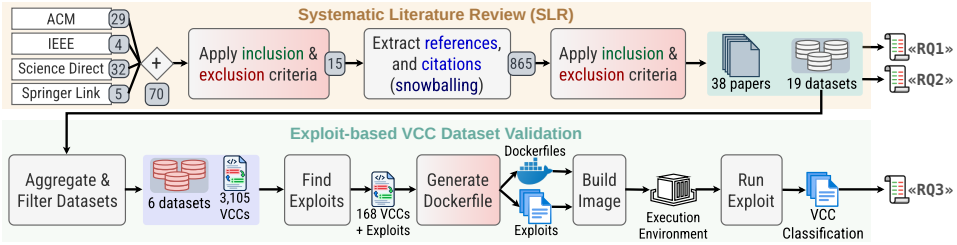

Figure 1 Overview of our two-part study. **Part 1:** the literature review to answer RQ1 and RQ2. **Part 2:** the exploit-based validation, aggregating 6 datasets into 3,105 samples, screening to the 168 with documented exploits, and testing each labeled commit and its parent to answer RQ3.

4 A Systematic Review of VCC-Related Literature

To answer RQ1 & RQ2, we first conducted a systematic review of VCC-related literature following the guidelines of Kitchenham and Charters [32].

4.1 Systematic Literature Review Methodology

The review consisted of three phases: planning, conducting, and reporting.

Planning Phase. We first established a review protocol defining: (i) the SLR's objectives, (ii) the search string, (iii) the digital libraries to query, and (iv) inclusion and exclusion criteria. The objectives of our SLR are fourfold: (1) to identify papers that describe or apply the concept of VCCs; (2) to analyze how VCCs are defined across different studies; (3) to

identify the key challenges researchers face when identifying and labeling VCCs; and (4) to collect and examine the VCC datasets referenced/provided by these papers.

We searched for papers containing any of the following terms (combined with `OR`): *"vulnerability contributing commit," "vulnerability introducing commit," "vulnerability injecting commit," "vulnerability fix inducing commit," "just in time vulnerability detection," "vulnerable commit,"* or *"vulnerable code change."* We applied this query to four major digital libraries (ACM Digital Library, IEEE Xplore, SpringerLink, and ScienceDirect).

Table 1 Inclusion and Exclusion Criteria for Paper Selection

Inclusion Criteria	Exclusion Criteria
I1 Describes a dataset of VCCs	**E1** Duplicate studies
I2 Describes an approach to detect, repair, or test VCCs	**E2** Not written in English
I3 Develops techniques to detect VCCs	**E3** Less than 4 pages of text
I4 Studies the problem of VCC identification	**E4** Secondary studies (*e.g.*, surveys)
I5 Describes techniques to identify VCCs from patches	**E5** Does not provide a VCC-related approach or dataset

We used Parsifal [24] to organize the review process. For each accepted paper, we extracted: (i) a summary of objectives and research questions, (ii) terminology used for vulnerable code changes, (iii) verbatim definitions of VCCs, (iv) challenges discussed in VCC identification, and (v) dataset availability and access information.

Conducting Phase. After collecting 70 papers from all four databases, we removed duplicates and applied the inclusion and exclusion criteria (shown in Table 1) in two stages: first based on titles, abstracts, and keywords, then through a full-text review. This first pass yielded 15 accepted papers. For these 15 papers, we performed one round of *forward* and *backward* snowballing, extracting their cited references and citing papers, which contributed 865 additional candidates. We did not iterate the snowball beyond this single round, since the four digital libraries already provided broad coverage of the relevant venues. We applied a keyword-based pre-filter on titles and abstracts to remove papers clearly outside the scope of VCC research, reducing the snowballed pool to 309 papers requiring detailed screening. Two computer science researchers (2 years of experience) independently screened these 309 papers following the criteria in Table 1. A senior software security researcher (4+ years of experience) verified their decisions, with disagreements resolved through discussion. This screening retained 23 of them, which combined with the 15 from the database search resulted in a final corpus of 38 papers. The Cohen's κ [18] between the verifier and each reviewer was 0.71 and 0.65, both indicating *substantial agreement* [34][1], and the 26 disagreements (8.41%) were resolved through discussion. Of these 38 papers, 19 provided VCC datasets.

Reporting Phase. We recorded each paper's own term, definition and stated challenges verbatim, then aggregated these records into three categories: *terminology* grouped by morphological variants of the same term (papers using no dedicated term formed a *generic terms* category), *definitions* by the dimension along which they disagreed, and *challenges* by shared underlying cause. The same two researchers coded independently, working from the extraction records and the grouping rules above rather than from a predefined category list, so that the categories emerged from the papers themselves. The senior author verified the coding, and differences were resolved by discussion. Per-paper codes are in Table 3 and Table 4, with the full extraction sheet in the replication package.

[1] We note that the inter-rater agreement reported here reflects agreement between extractors and a more experienced verifier rather than between equally-trained peers.

4.2 RQ1 Results: Terminology and Definitions

As shown in Table 2, our SLR yielded **38 papers** spanning from 2005 to 2025.

Table 2 Year-wise distribution of accepted papers ($n = 38$).

Year (#)	Papers
2005–2015 (4)	[8, 40, 50, 62]
2017–2020 (7)	[16, 23, 27, 37, 57, 67, 71]
2021–2022 (10)	[5, 11, 19, 28, 35, 38, 49, 54, 68, 69]
2023–2025 (17)	[1, 4, 12, 13, 15, 17, 26, 30, 43–45, 58, 61, 63, 64, 70, 72]

Terminology Usage. Our analysis revealed **8** distinct terms used in the literature to refer to commits that introduce vulnerabilities. These terminologies are listed in Table 3.

Table 3 Terminology usage across VCC-related papers ($n = 38$).

Terminology	Papers	Total
Vulnerability-Contributing Commits (VCC)	[1, 4, 8, 12, 13, 15, 19, 26, 27, 35, 40, 45, 49, 50, 54, 57, 67, 71]	18
Just-In-Time (JIT) vulnerability detection	[38, 43, 64]	3
Vulnerability-Inducing/Introducing Commits (VIC)	[5, 17, 30, 44]	4
Security patches / Patch commits	[11, 58, 72]	3
Vulnerability Introduction Detection (VID)	[70]	1
Vulnerability-relevant / -related commits	[16, 68]	2
Fix-Inducing Changes (FIC)	[62]	1
Generic terms (e.g., “vulnerable code”)	[23, 28, 37, 61, 63, 69]	6

Definition Patterns. Grouping the verbatim definitions by the dimension along which they disagreed revealed three inconsistency types:

- **Temporal ambiguity**: Some works define VCCs as *“the commit that introduced a vulnerability”* [26, 27, 50, 54] while others specify *“the previous commits which last modify [vulnerable] statements”* [45] or *“the earliest commit that modified the vulnerable lines”* [5].
- **Attribution scope**: Definitions vary between commits that *“contributed to the introduction of a post-release vulnerability”* [40] or *“likely to have contributed to the introduction of a vulnerability”* [38] versus commits that *“contain software vulnerabilities”* [35].
- **Attribution granularity**: Some approaches identify a single VCC per vulnerability, such as *“the earliest commit as the inducing commit”* [5], while others acknowledge that *“forming a vulnerability in a code version could require one or more VCCs over time”* [45].

4.3 RQ2 Results: Challenges in VCC Identification

We found five recurring challenges in VCC identification and labeling (Table 4): **21** papers discussed at least one of these challenges, while the remaining **17** [1, 4, 8, 11, 16, 23, 28, 37, 44, 49, 50, 57, 62, 67–69, 72] used existing labeling approaches or raised tangential limitations (*e.g.*, class imbalance, scope) without thorough examination of VCC identification accuracy.

Table 4 Challenges in VCC identification across VCC-related papers ($n = 38$).

Challenge	Papers	#	%
C1: SZZ Algorithm Limitations	[5, 12, 17, 19, 26, 27, 35, 45, 61, 64, 70]	11	28.9%
C2: Lack of Ground Truth Validation	[12, 15, 27, 30, 35, 40, 63, 64, 70, 71]	10	26.3%
C3: External Data Source Unreliability	[5, 19, 30, 58, 63]	5	13.2%
C4: Multi-Commit Vulnerability Introduction	[13, 15, 45, 70]	4	10.5%
C5: Reproducibility Barriers	[26, 38, 43, 54]	4	10.5%

Note: Some papers discuss multiple challenges.

C1: SZZ Algorithm Limitations. We found **11 papers** documented limitations in how SZZ traces and attributes vulnerability-introducing changes. At the *line-tracing* level, SZZ’s *“last modified”* heuristic assumes the commit that last touched a line introduced the defect, an assumption that breaks down in several ways. V-SZZ [5] found that over 50% of vulnerabilities were introduced in initial or foundational commits that SZZ cannot trace, because these

commits create code rather than modify existing lines. SZZ also cannot handle fixes that consist entirely of added lines, since there are no deleted lines to trace back from [17, 26, 70]. The *"inappropriate patch assumption"*, where prior work assumes patches contain only vulnerability-relevant changes and must include deletion lines, further distorts SZZ-style tracing [61]. Moreover, at the *commit-identification* level, tangled and multi-purpose commits that mix vulnerability-introducing code with unrelated changes cause SZZ to attribute the vulnerability to the entire commit [26,45]. Cosmetic changes (*e.g.*, comments, refactoring, and variable renames) cause SZZ to flag commits that merely touched but did not introduce the vulnerability [19]. In the adjacent setting of non-functional bugs, rather than vulnerabilities, empirical studies of SZZ report false positive rates as high as 79% [52], with mislabeling persisting even after curation [35] (only 85% of automatically mined VCCs were valid after manual verification), making manual validation infeasible at scale. The resulting noise is substantial: Coskun *et al.* [19] report that SZZ-based tracing initially identified 402 candidate inducing commits, which filtering reduced to 187, and Sun *et al.* [64] needed to manually filter mislabeled commits before their data was usable.

C2: Lack of Ground Truth Validation. We found **10 papers** highlighting the absence of validated ground truth as a fundamental limitation in VCC research. Manual labeling, the most direct path to validated ground truth, faces practical constraints: it requires "hundreds of man-hours" [40] even for small datasets, and VulDigger [71] noted the lack of VCC ground-truth datasets as a core obstacle. Moreover, when manual verification is attempted, the results expose labeling errors in automatic methods. Le *et al.* [35] reported that only 85% of their automatically mined VCCs were valid after manual verification, meaning 15% were mislabeled from the start. Manual verification is itself imperfect: Meneely *et al.* [40] observed that some vulnerabilities could not be bisected due to missing context, and Charoenwet *et al.* [15] acknowledged that manual CWE mapping introduces human bias into VCC datasets, impacting study results. Despite this evidence, most datasets rely on SZZ-based identification with no manual verification step. Cannavale *et al.* [12] explicitly acknowledged evaluating model performance *"without validating absolute VCC correctness against manual ground truth"*, leaving labeling accuracy unverified. CommitShield [70] pointed to label noise from incomplete commit context, and Sun *et al.* [63] stated directly that *"our ground truth is imperfect"*, noting that vulnerabilities may span multiple repositories and files, with patching commits that are *"incomplete or contradictory"*.

C3: External Data Source Unreliability We found **5 papers** that identified problems with the external data sources (primarily NVD and CVE databases) that VCC identification pipelines depend on as input. V-SZZ [5] found that 99 of 172 NVD entries (57.6%) had incorrect version information, meaning the starting point for many VCC traces is already wrong. Hash4Patch [58] reported that only 28.75% of vulnerabilities in NVD have a link to their patch, limiting the availability of confirmed fixing commits needed as entry points for VCC identification. They also described a "chicken-and-egg problem": building a vulnerability dataset requires patches, but finding patches requires an existing dataset.

The traceability gap extends beyond missing links. Sun *et al.* [63] noted that *"the repository of the vulnerable product is not the mandatory information in the NVD,"* which creates uncertainty when even locating the correct codebase. Jiang *et al.* [30] found no explicit link either from a CVE to its fix commit or from a fix commit onward to the corresponding vulnerability-inducing commit, so the traceability gap compounds across both hops. Coskun *et al.* [19] found that SmartSHARK [66] did not include the vulnerabilities their prediction model required, forcing manual extraction.

C4: Multi-Commit Vulnerability Introduction We found **4 papers** noting that a vulnerability may result from code changes spread across multiple commits, challenging the assumption of a single VCC per vulnerability. Nguyen *et al.* [45] noted that a vulnerability may arise from one or more VCCs over time, yet simplified their approach to focus on single vulnerability-triggering commits, potentially omitting earlier contributing changes. Charoenwet *et al.* [15] and Cao *et al.* [13] both noted that a vulnerability may depend on code changes introduced across multiple commits, which complicates one-to-one VCC labeling and introduces ambiguity in ground truth construction. CommitShield [70] pointed out that the single-VCC assumption underrepresents multi-commit introduction patterns, increasing false negatives. Moreover, Cao *et al.* [13] also observed that the actual vulnerable changes within a VCC are typically small relative to the total set of code changes in the commit, making them difficult to identify even when the correct commit is labeled.

C5: Reproducibility Barriers We observed **4 papers** noting difficulties in reproducing VCC research artifacts or historical software states for validation. Riom *et al.* [54] and Lomio *et al.* [38] reported being unable to fully reproduce VCCFinder [50] due to missing details and artifacts. Hinrichs *et al.* [26] further identified tooling inconsistencies, missing dependencies, and required implementation changes that threatened reproducibility and potentially affected results. Nguyen *et al.* [43] described data curation for VCC research as *"repository-specific, error-prone, and labor-intensive"*, noting that this complexity results in limited experimental scale and inconsistent evaluation across models.

5 Exploit-Based Formalization of VCCs

Our SLR revealed inconsistent VCC terminology and definitions. We identified eight distinct terms, often used interchangeably, while 15.8% of papers gave no explicit definition and others reflected differing interpretations. Some adopted ***operational definitions*** describing *how* VCCs are identified rather than *what* they are (*e.g.*, *"a code modification identified through a semi-automatic process using git blame"* [71]). Others relied on ***containment definitions*** (*"commits whose changes contain software vulnerabilities"* [35]) or ***modification definitions*** (*"the previous commits which last modify [vulnerable] statements"* [45]), both of which identify commits that merely touched or contained vulnerable code rather than commits at which the vulnerability first became exploitable. These inconsistencies have practical consequences. SZZ-based approaches, which dominate dataset construction, trace fixed lines backward, potentially labeling commits that merely reformat, move, or extend pre-existing vulnerable code, even when the vulnerability predates them. This confusion between commits that contain vulnerable code and commits at which it becomes exploitable propagates mislabeled samples into training datasets.

These challenges (§ 4.3) shape the definition and the validation protocol that follow in this section and in § 6. C1 and C2 motivate validation itself, since SZZ mislabels and no validated ground truth exists, and testing the *parent* is what separates introducing from containing; C5 motivates the two-tier fallback, since historical build environments frequently cannot be reproduced. C3 motivates the ban on lineage-based reasoning, since unreliable NVD and CVE metadata means a verdict must rest on direct evidence at the commit, and C4 is why the definition anchors on exploitability rather than on responsibility for the vulnerability.

5.1 VCC Formal Definition

Our definition is grounded in the *security state transition* observed across a commit. Intuitively, a VCC is the commit at which the project's security status with respect to a specific

vulnerability changes from *safe* to *unsafe*; that is, the precise point where that vulnerability becomes *exploitable*. While the project may contain other unrelated vulnerabilities before or after this commit, the state transition refers only to the vulnerability in question. A vulnerability may also emerge from code changes spread across multiple commits [70], or depend on configuration and on properties of the surrounding execution environment. Rather than apportioning responsibility among such changes, we anchor the definition on a property that can be tested directly: whether the vulnerability is exploitable in a given state. This yields a deterministic criterion that separates VCCs from commits that merely contain or modify existing vulnerable code: **a VCC is the commit at which the vulnerability first becomes *exploitable***, that is, the earliest state in the project's history in which an attacker can compromise the system through that vulnerability, as formally defined below.

▶ **Definition 1 (Vulnerability-Contributing Commit – VCC).** *Let* $\mathbf{P}$ *be a software project,* $\mathbf{v}$ *be a specific vulnerability, and* $\mathbf{c}$ *be a commit with parent commit* $\mathbf{c}^-$*. We define* $\mathbf{c}$ *as a* ***VCC*** *for vulnerability* $\mathbf{v}$ *iff:* $\boldsymbol{VCC}(\mathbf{c}, \mathbf{v}) \iff \boldsymbol{Exploitable}(\mathbf{v}, \mathbf{P}(\mathbf{c})) \wedge \neg \boldsymbol{Exploitable}(\mathbf{v}, \mathbf{P}(\mathbf{c}^-))$*, where* $\mathbf{P}(\mathbf{c})$ *denotes the software state after applying commit* $\mathbf{c}$*.*

A commit is a VCC only if the vulnerability becomes exploitable at that commit and was not exploitable in the previous version. Two edge cases require special treatment:

- **Initial commits**: When c has no parent, the condition on $P(c^-)$ does not apply; any vulnerability present in $P(c)$ is, by construction, first exploitable at c.
- **Merge commits**: When c has multiple parents, our validation focuses exclusively on the main branch. We treat the merge commit itself as the single predecessor of the subsequent commit, without tracing into the merged feature branch history. This reduces the multi-parent case to a single-parent comparison.

5.2 Operationalization via Parent Commit Validation

Our definition can be operationalized through a parent commit validation procedure. Given a labeled commit c, we first reproduce the software state at $P(c)$ and verify whether vulnerability v is exploitable in that version. We then reproduce the parent state $P(c^-)$ and attempt to exploit the same vulnerability in the preceding version. A commit is classified as a **VCC** if the vulnerability is exploitable in $P(c)$ but not in $P(c^-)$, indicating that the exploitable state first appears at c. Conversely, if the vulnerability is exploitable in both versions, the commit is classified as a **false positive** because the vulnerability was already exploitable prior to the labeled commit. Finally, if the vulnerability is not exploitable even in $P(c)$, the label is also considered a **false positive**, indicating incorrect vulnerability attribution.

This methodology directly tests the state-transition criterion. Existing approaches (SZZ-based identification, CVE/NVD mapping, and manual labeling) can serve as *candidate generation* steps, efficiently identifying commits at which a vulnerability may have become exploitable. However, without parent commit validation, these candidates remain unverified. Our definition provides the criterion against which candidates must be tested.

6 Exploit-Based Validation of VCC Datasets

VCC datasets are widely used to train and evaluate vulnerability detection tools, yet their accuracy is often assumed rather than validated, allowing mislabeled commits to propagate errors into downstream models. To answer **RQ3**, we apply our exploit-based VCC validation (§ 5.2) to determine whether labeled commits in existing datasets satisfy Definition 1. This section describes our validation methodology (§ 6.1) and presents RQ3 results (§ 6.2).

6.1 RQ3 Methodology

Our protocol operationalizes the security-state-transition definition (Section 5): c is a VCC for v iff v is exploitable post-commit and not in its parent. We impose an explicit ban on *lineage-based reasoning*, meaning that a commit's verdict must come from direct evidence at the commit itself, never from its position in version history (e.g., date relative to a fix release). Each commit enters with a CVE identifier, commit hash, repository URL, and reference to a public exploit. The protocol is a two-tier cascade: **Tier 1** attempts ***runtime*** exploit reproduction, and **Tier 2** (***static***) runs only when Tier 1 cannot produce a runnable artifact. We restrict Tier 2 to the fallback role because static matching establishes pattern presence, not exploitability. Commits with an `EXPLOITABLE` child verdict are re-evaluated at the parent under the same protocol, and the joint (child, parent) pair determines the final classification. Since the two sides of a pair can be resolved at different tiers, we record the tier separately for the child and the parent verdict, and refer to a commit as *Tier-1-pure* when every side that was tested reached its verdict dynamically.

6.1.1 Dataset Aggregation and Exploit Retrieval

We applied the following inclusion criteria to the 19 datasets identified in the SLR (§ 4): **(i)** it contains vulnerable code changes (commits) rather than general bugs; **(ii)** it includes metadata (repository URL, commit hash, CVE identifier) needed for exploit validation; **(iii)** it contains commits from Java projects, for a consistent build and exploitation toolchain[2]. After applying these criteria, which are distinct from the paper-selection criteria in Table 1, we discarded 13 datasets. Then, after aggregating the samples in the remaining **6 datasets**, removing duplicated (repository URL, commit hash) pairs and removing samples without resolvable hashes, we obtained **3,105 aggregated VCC samples**. For each of these 3,105 samples, we searched Exploit-DB [47], the most comprehensive public exploit collection [3, 14, 21], to find associated exploits with reproduction steps. Of the 3,105 samples, we found associated exploits for **168 (5.41%)** of them[3]. This rate is consistent with prior findings that only 2–4% of CVEs in NVD correspond to exploits observed in real-world attacks [3].

6.1.2 Tier 1: Runtime-based Exploit Analysis (Dynamic)

Tier 1 reproduces the labeled commit's vulnerability by cloning the repository at the target commit, creating a Dockerfile containing the exploit script, building a Docker image, and trying two execution environments in sequence to run the application. We used GPT-4o and Claude Sonnet 4 to draft the initial Dockerfiles and exploit scripts. Before building the execution environment, we determined manually what the reproduction should do for that CVE, checked the Dockerfile's repository URL, commit hash, and build steps against the dataset record, and confirmed that the exploit script exercises the behavior described in the Exploit-DB write-up. Artifacts that diverged were manually corrected.

We then build the Docker image using two execution environments in sequence. The first is an ***at-hash source build*** using the project's native build tool (Maven, Gradle, or Ant). When at-hash dependency resolution fails (common for older projects whose pre-2014 Maven coordinates are no longer hosted on public mirrors), we fall back to a ***release-baseline***

[2] Cross-language replication (C/C++, Python) is left to future work.

[3] We note that exploit availability is not a dataset-level filtering criterion: it is applied per VCC sample aggregated from the 6 datasets. This is why the datasets start with far more samples than we can ultimately validate.

overlay: a pre-built release archive from an immutable mirror (*e.g.*, `archive.apache.org`, Maven Central, DockerHub, *etc*) is deployed as the surrounding runtime, and then at-hash `.class` files for the CVE-relevant source files are surgically replaced inside the deployed artifact. What executes at runtime is the commit's own bytecode for the CVE-relevant classes embedded in a release-baseline framework; the baseline is the release closest in date to the labeled commit. We record which execution environment was used for each verdict. If neither environment produces a runnable artifact, the protocol moves to Tier 2.

We execute the exploit on the runnable environment and record HTTP traces, container logs, and filesystem side-effects to identify three possible outcomes: *exploitable*, *not exploitable*, or *inconclusive*. Unambiguous indicators of unsafe behavior (RCE, directory traversal beyond the configured root, privilege escalation) yield `EXPLOITABLE`. A clean failure (sanitized response, blocking exception in framework code) yields `NOT_EXPLOITABLE`. An environment that never reached a runnable state yields `INCONCLUSIVE` with the failure reason recorded. These three verdict categories were uniformly assigned across all commits by a senior author. We also note that errors are self-limiting: a configuration that does not build yields a documented Tier 1 failure rather than a verdict, and an `EXPLOITABLE` verdict requires the exploit to actually execute in the container.

6.1.3 Tier 2: Pattern-Based Exploit Analysis (Static)

Tier 2 runs only after Tier 1 fails. The failure is recorded with its *error category* (*e.g.*, unavailable dependencies, build tooling, repositories, binary distributions, or exploit environments). The *error message* is also recorded alongside the Tier 2 verdict to confirm Tier 2 was not invoked as a shortcut. For each Tier 2 commit, we first derive a CVE-specific code pattern from the exploit description, naming concrete methods, classes, or files rather than generic strings. Patterns are derived from three complementary sources: (i) the Exploit-DB proof-of-concept, which identifies the vulnerable sink; (ii) the fixing commit's diff, which reveals the mitigation, that is, the call or flag the maintainers added to close the vulnerability, such as a canonical-path check before a file read; and (iii) the CVE advisory, which specifies the affected source files. Initial drafts of the vulnerable and mitigation patterns were generated with GPT-4o and Claude Sonnet 4 from these inputs and manually verified before admission: we inspected the source files named in the CVE advisory to locate the vulnerable construct by hand, and admitted a pattern only when it targeted that same construct, checking the mitigation term the same way against the API introduced by the fixing commit's diff. Patterns that did not match the construct we had located were rewritten or discarded. The labeled commit under evaluation was never shown during pattern construction, so a pattern cannot be tuned to the commit it is later used to judge. Patterns are broad enough to absorb minor refactors but narrow enough to avoid spurious matches. We then search the isolated checkout across documented globs. The verdict is determined by the joint presence of the vulnerable pattern and any recognizable mitigation (*e.g.*, *getCanonicalPath* with *startsWith*, input validation, or hardening flags such as *FEATURE_SECURE_PROCESSING*). Pattern present without co-located mitigation yields `EXPLOITABLE`; pattern absent or mitigated yields `NOT_EXPLOITABLE`; a refactored layout with no recognizable equivalent yields `INCONCLUSIVE`.

6.1.4 Parent Commit Analysis Step and VCC Classification

A child commit's verdict alone is insufficient to classify it as a true VCC: the parent must be tested to rule out SZZ over-attribution. We trigger the parent commit analysis step whenever the child commit verdict is `EXPLOITABLE`, regardless of tier. Initial commits (no

parent) are classified `TRUE_VCC` by construction, since no prior state exists in which the vulnerability could have been exploitable. Merge commits are reduced to the single-parent case by following the first-parent edge only, since our methodological question concerns the labeled commit as it appears on the main-line history. For every commit with a testable parent, we construct the same validation artifact at the parent commit hash and apply the full Tier 1 then Tier 2 cascade under the same lineage ban and tier discipline. The joint verdicts determine the final classification: an `EXPLOITABLE` child with a `NOT_EXPLOITABLE` parent, or an initial commit with no parent, yields `TRUE_VCC`; an `EXPLOITABLE` parent yields `FP_PARENT_EXPLOITABLE`; a `NOT_EXPLOITABLE` child yields `FP_NOT_EXPLOITABLE` with no parent test; and an `INCONCLUSIVE` verdict on either side leaves the case unresolved rather than promoted to a verdict, excluded from both counts in Section 6.2.

For each commit, we produce standardized validation artifacts: a *Dockerfile* and a *Compose* file building the historical runtime at the commit's hash, a script issuing the proof-of-concept against the running container, and a *result.txt* recording the verdict, its tier, and the supporting evidence. Child commits additionally carry the joint child and parent classification. Before a verdict is made, an automated check confirms that `result.txt` uses only the verdict labels defined in Sections 6.1.2–6.1.3, that the evidence type matches the tier (runtime logs for Tier 1, source patterns for Tier 2), that any Tier 1 build failure is documented before fallback to Tier 2, and that the written justification cites direct evidence at the commit being tested rather than the commit's position in version history.

6.2 RQ3 Results: Dataset Accuracy

We applied the methodology to the 168-commit corpus, drawn from 61 CVEs affecting over 30 projects in the Apache ecosystem (*e.g.*, Struts, Tomcat, Commons, Solr, ZooKeeper, Olingo, Tika, ActiveMQ, Geronimo, Syncope), the Spring family (*e.g.*, Framework, Security OAuth, Cloud Config), Jenkins, and others (complete list in the replication package). As shown in Table 5, we find that 133/168 (79.2%) labeled commits are SZZ false positives under the security-state-transition definition. This rate characterizes the validatable subset defined in § 6.1.1, *i.e.*, the 5.41% of the 3,105 aggregated samples whose CVEs have public exploit infrastructure, and not the aggregate corpus. Tier 1 was attempted against every commit; the 90/168 (53.6%) Tier 2 share reflects forced fallbacks after Tier 1 failed on dependency or build-tool failures (unresolvable Maven coordinates, dead HTTP mirrors, Ant-era classpaths under modern JDKs, broken SNAPSHOT chains), not a methodological choice to bypass dynamic validation. Each failure is documented with the exact build error before falling to Tier 2. Since Tier 2 establishes pattern presence rather than exploitability, Table 5 reports each outcome alongside the count whose verdicts rest entirely on runtime evidence.

Table 5 Joint child and parent verdict distribution with final classification ($n = 168$). The Tier-1-pure column counts commits for which every tested side reached its verdict through dynamic exploit reproduction ($n = 69$); percentages in that column are relative to 69.

Child commit verdict	Parent commit verdict	Count (%)	Tier-1-pure (%)	Final classification
`EXPLOITABLE`	`EXPLOITABLE`	79 (47.0%)	36 (52.2%)	`FP_PARENT_EXPLOITABLE`
`EXPLOITABLE`	`NOT_EXPLOITABLE`	30 (17.9%)	7 (10.1%)	`TRUE_VCC`
`EXPLOITABLE`	(initial, no parent)	5 (3.0%)	1 (1.4%)	`TRUE_VCC`
`NOT_EXPLOITABLE`	(not tested)	54 (32.1%)	25 (36.2%)	`FP_NOT_EXPLOITABLE`
`INCONCLUSIVE`	not applicable	0 (0.0%)	0 (0.0%)	not applicable
TRUE_VCC total		**35 (20.8%)**	**8 (11.6%)**	
False Positive total		**133 (79.2%)**	**61 (88.4%)**	

Tier breakdown: child step 78 *Tier 1,* 90 *Tier 2; parent step (*$n = 109$*)* 43 *Tier 1,* 66 *Tier 2. The* 69 *Tier-1-pure commits comprise* 43 *resolved dynamically on both sides,* 25 *with a dynamic* `NOT_EXPLOITABLE` *child verdict requiring no parent test, and* 1 *initial commit. No commit received a Tier 2 child verdict together with a Tier 1 parent verdict.*

Interpretation The dominant outcome is that the labeled commit does *not* satisfy Definition 1. The 133 false positives split across two failure modes. `FP_PARENT_EXPLOITABLE` (47%): the vulnerability is exploitable at the labeled commit, but it was already exploitable at its parent. The exploit runs, or the vulnerable pattern is present without mitigation, at both sides of the diff, so by construction the labeled commit does not exhibit the exploitability transition; the vulnerable state is inherited from its parent. This is precisely what SZZ-style backward tracing cannot detect, since SZZ traces from a fixing commit to the last commit that modified the fixed lines without testing whether the predecessor was already exploitable. At 47%, this is the most common labeling error in our corpus, not an edge case. `FP_NOT_EXPLOITABLE` (32.1%): the vulnerability is not exploitable at the labeled commit. Recurring patterns include commits touching a file or symbol whose name overlaps with the vulnerable artifact in a later refactor, fixing commits for one CVE misattributed to a different CVE, and merge commits SZZ blamed in place of the originating branch change. Only 35/168 (20.8%) commits validate as `TRUE_VCC` (5 initial commits plus 30 with `NOT_EXPLOITABLE` parents). The `INCONCLUSIVE` buckets are empty: every Tier 1 failure was resolved by Tier 2.

Sensitivity to evidential strength The false positive rate is *higher* in the subset supported by runtime evidence alone (88.4%, 95% Wilson CI 78.8–94.0%) than in the full corpus (79.2%, 95% Wilson CI 72.4–84.6%), and the difference between the two strata corresponds to $p = 0.020$ under Fisher's exact test. Since the 168 commits derive from only 61 CVEs, several commits often share a CVE and are not independent observations, so we read this contrast as descriptive of the two strata rather than as an inferential test. Thus, the combined 79.2% rate is conservative with respect to the tier question: restricting attention to the strongest evidence strengthens rather than weakens the finding. The most likely mechanism is that Tier 2 under-detects `FP_PARENT_EXPLOITABLE`. A parent whose file layout was later refactored may not match the CVE-specific pattern even though the vulnerable path is present, yielding a `NOT_EXPLOITABLE` parent verdict and promoting the pair to `TRUE_VCC`; commits involving a Tier 2 verdict validate as true VCCs at 27.3% against 11.6% in the Tier-1-pure subset. Combining the two tiers thus biases the corpus toward appearing *cleaner* than the dynamic evidence indicates, so conclusions drawn from the combined rate understate rather than overstate the labeling problem.

7 Discussion

Our findings have several implications for the state of the art in VCC research.

Terminology and Definition Standardization The 8 different terminologies and 3 definition patterns found in RQ1 lead to practical challenges: when different papers operationalize different concepts under the same "VCC" label, cross-study comparisons and dataset aggregation become unreliable. The confusion between *"contributed to"*, *"introduces"* and *"contains"* (attribution scope), and the disagreement over which commit in a chain to blame (temporal ambiguity), also explain the labeling errors observed in RQ3. We recommend that the community adopt "Vulnerability-Contributing Commit (VCC)" as the standard terminology, given its prevalence in the literature (47.4% of the reviewed papers), alongside a state-transition definition, as described in § 5. This definition replaces the temporal ambiguity and the varying attribution wording documented in § 4 with a single testable criterion. We retain the established modifier *"contributing"* for continuity with the existing literature rather than as a claim about causal responsibility: the criterion attached to the term tests where a vulnerability's exploitability status changes, and does not attempt to apportion responsibility among the code changes that jointly enable the vulnerability. We further advocate for parent

commit validation as a standard verification methodology for dataset construction, enabling empirically validated ground truth beyond SZZ-based candidate generation.

Dataset Quality and ML Training Only 5.4% of the 3,105 aggregated samples target CVEs with public exploit infrastructure, consistent with prior work showing that only 2-4% of CVEs in NVD correspond to exploits observed in real-world attacks [3]. This bound is set by the datasets rather than our validation steps, so no exploit-based protocol can speak to the labeling accuracy of the remaining 94.6%. Within the validated subset, the 79.2% false positive rate (Table 5) points not to random noise but to systematic confusion: SZZ traces back to commits that *touched* vulnerable lines, not commits at which the vulnerability first became *exploitable*. The 79 `FP_PARENT_EXPLOITABLE` commits (47.0%) demonstrate this directly. Models trained on such data risk learning to detect code modification rather than the transition into an exploitable state, and when training and test sets share this bias, a model can achieve high accuracy while misaligning with the research goal. The practical implication for dataset consumers is therefore conditional: the measured error rate applies to the validated subset, while the correctness of the unvalidated majority rests on the same SZZ heuristics that produced it, with no positive evidence that the majority is cleaner.

Reproducibility Barriers We also observed another structural barrier: 90 (53.6%) out of the 168 validated commits could not be brought to a runnable state under either Tier 1 environments. **Build complexity**, rather than project age, was the main reason: projects with self-contained build systems reproduced successfully regardless of commit age, while projects with heavy transitive dependency chains and obsolete package mirrors consistently failed, even for relatively recent commits. Any researcher attempting to reproduce these historical states would encounter the same failures. Future dataset construction should address both barriers: prioritizing CVEs with documented exploits and projects with reproducible build systems; this will yield smaller but Tier 1-validated corpora, providing stronger evidence than larger corpora based on unverified heuristics.

8 Threats to Validity

Internal Validity Our exploit-based validation depends on the correctness of exploit documentation from Exploit-DB. Errors in exploit guides could cause valid VCCs to be classified as false positives. We mitigated this by manually reviewing failed exploits and consulting multiple sources when available. Moreover, our use of LLMs to draft Dockerfiles, exploit scripts and patterns may have introduced errors. We addressed this through the per-artifact checks described in § 6.1.2 and § 6.1.3. These checks were carried out by a senior author, so they were applied uniformly across commits; all generated artifacts are in the replication package so the checks can be repeated independently. Moreover, the release-baseline overlay described in § 6.1.2 assumes that the framework code surrounding the CVE surface is stable between the labeled commit and the chosen release. Structural changes between the two states could therefore make an overlay verdict diverge from an at-hash source build. We mitigated this by choosing the release closest in date to the labeled commit and by recording the substrate provenance alongside every overlay verdict.

External Validity Our validation covers the 168 samples (5.41% of 3,105) with publicly documented exploits. The remaining 94.6% cannot be validated using publicly documented exploits: their CVEs lack the public exploit infrastructure that runtime validation requires. This limitation is inherent to the underlying datasets, not our methodology, and applies to any exploit-based validation protocol.

Another threat is that the validated subset may be unrepresentative. CVEs with publicly

documented exploits may differ from those without them, potentially affecting the accuracy of the underlying VCC labels. However, we have no evidence to determine the direction or magnitude of these differences. We therefore cannot determine whether the false-positive rate for the full population would be higher or lower than 79.2%, and we do not extrapolate beyond the validated subset. What carries beyond the subset is the mechanism rather than the magnitude: `FP_PARENT_EXPLOITABLE`, the most common error, arises from backward tracing without testing whether the predecessor was already exploitable, a limitation independent of exploit availability. Therefore, readers should treat 79.2% as a measurement on the 5.41% that can be checked, and as evidence that prior SZZ-based labeling procedure itself is unreliable, not as an estimate across all 3,105 samples. Moreover, our focus on 6 datasets may not represent all VCC datasets; the Java-only inclusion criterion (§ 6.1.1) means our measured rate characterizes Java VCC datasets rather than the field as a whole.

Construct Validity Tier 1 verdicts come from a runtime exploit attempt against the at-hash bytecode, while Tier 2 establishes only pattern presence, not exploitability. Of the 168 commits, 90 (53.6%) received a Tier 2 child verdict because Tier 1 failed, not because it was not attempted, so we report outcomes stratified by tier (Table 5). The 69 Tier-1-pure commits, whose verdicts rest entirely on runtime evidence, exhibit a higher false positive rate (88.4%) than the corpus as a whole (79.2%), so the combined 79.2% rate is conservative with respect to this threat: the weaker evidence dampens rather than inflates the reported rate. The residual threat therefore runs opposite to the direction usually assumed, namely that Tier 2 may under-count `FP_PARENT_EXPLOITABLE` at refactored parents and thereby overstate the number of true VCCs.

We further define dataset reliability in terms of exploit reproducibility and parent commit validation. This operationalization may not capture all aspects of reliability, such as completeness (whether datasets include all VCCs for covered projects) or representativeness (whether included VCCs reflect real-world distributions). We also evaluate the protocol without a comparison baseline. Our SLR identified no existing operational validation methodology for VCC labels, so no published technique is available to compare against, and the natural alternative, independent manual expert validation on a representative subset, is the procedure the literature itself reports as infeasible at scale (§ 4.3, C2). Establishing agreement between our verdicts and independent expert judgment on a subset is left for future work and the released artifacts are structured to support it. Lastly, our verdicts also rest on the documented Exploit-DB proof-of-concept for each CVE, so a `NOT_EXPLOITABLE` verdict records that this exploit does not fire rather than that no exploit could. A further exploit succeeding at a parent would move that pair from `TRUE_VCC` to `FP_PARENT_EXPLOITABLE` and never the reverse, so `TRUE_VCC` can only *decrease* as exploit coverage improves.

9 Conclusion

We presented a two-part study of vulnerability-contributing commits, combining a systematic literature review of 38 papers (2005–2025) with exploit-based validation of 168 VCC samples from 6 datasets identified in our SLR. The review uncovered terminology inconsistencies in the field (8 distinct terms, 3 conflicting definition patterns) alongside five recurring VCC identification challenges. To resolve this ambiguity, we proposed a formal VCC definition grounded in *security state transitions*: a commit is a true VCC if and only if the vulnerability is exploitable after the commit but not in its parent. Thus, our key methodological contribution, *parent commit validation*, operationalizes this definition by testing whether the exploitability transition occurs at the labeled commit. Applied to the 168 commits (5.41% of 3,105) with

documented exploits, it revealed a 79.2% false positive rate: in 47.0% of cases the vulnerability was already exploitable in the parent commit, and in 32.1% it was not exploitable at the labeled commit at all.

10 Data Availability

Our replication artifact is available at: `https://doi.org/10.5281/zenodo.21904512`.

References

1. T. Aladics, P. Hegedűs, and R. Ferenc. A comparative study of commit representations for JIT vulnerability prediction. *Computers*, 2024. doi:10.3390/computers13010022.
2. T. Aladics, P. Hegedűs, and R. Ferenc. A vulnerability introducing commit dataset for java: An improved szz based approach. In *17th Intl. Conf. on Softw. Technologies (ICSOFT)*, 2022. doi:10.5220/0011275200003266.
3. L. Allodi and F. Massacci. A preliminary analysis of vulnerability scores for attacks in wild: The ekits and sym datasets. In *2012 ACM Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, 2012. doi:10.1145/2382416.2382427.
4. V. Almeida and R. Andrade. Investigating vulnerability-fixing commits. *Journal of the Brazilian Computer Society*, 2025. doi:10.5753/jbcs.2025.4675.
5. L. Bao, X. Xia, A. E. Hassan, and X. Yang. V-SZZ: Automatic identification of version ranges affected by CVE vulnerabilities. In *44th Intl. Conf. on Software Engineering (ICSE)*, 2022. doi:10.1145/3510003.3510113.
6. S. Bekrar, C. Bekrar, R. Groz, and L. Mounier. Finding software vulnerabilities by smart fuzzing. In *4th Intl. Conf. on Software Testing, Verification and Validation (ICST)*, 2011.
7. M. Borg, O. Svensson, K. Berg, and D. Fankhänel. SZZ unleashed: An open implementation of the SZZ algorithm. In *3rd Intl. Workshop on Machine Learning Techniques for Softw. Quality Eval.uation (MaLTeSQuE)*, 2019. doi:10.1145/3340482.3342742.
8. A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *22nd Intl. symposium on foundations of software engineering*, 2014.
9. A. Bosu, J. C. Carver, M. Hafiz, P. Hilley, and D. Janni. When are oss developers more likely to introduce vulnerable code changes? a case study. In *IFIP International Conference on Open Source Systems*, pages 234–236. Springer, 2014.
10. L. Braz, E. Fregnan, V. Arora, and A. Bacchelli. An exploratory study on regression vulnerabilities. In *16th Intl. Symposium on Empirical Softw. Eng. and Measurement*, 2022.
11. G. Canfora, D. Di Nucci, G. Garofalo, E. Iannone, A. D. Lucia, and F. Palomba. Patchworking: Exploring the code changes induced by vulnerability fixing activities. *Information and Software Technology*, 2022. doi:10.1016/j.infsof.2021.106745.
12. A. Cannavale, E. Iannone, G. Di Lillo, F. Palomba, and A. De Lucia. The ground truth effect: Investigating SZZ variants in just-in-time vulnerability prediction. In *Software Engineering and Advanced Applications (SEAA 2025)*, Lecture Notes in Computer Science, 2025. doi:10.1007/978-3-032-04207-1_21.
13. S. Cao, X. Sun, X. Wu, D. Lo, L. Bo, B. Li, X. Liu, X. Lin, and W. Liu. Snopy: Bridging sample denoising with causal graph learning for effective vulnerability detection. In *39th Intl. Conf. on Automated Software Engineering (ASE)*, 2024. doi:10.1145/3691620.3695057.
14. R. Chadha, G. S. Shalom, V. K. Anand, and A. Goel. A study on exploit development. In *2022 7th Intl. Conf. on Computing, Communication and Security (ICCCS)*, 2022. doi:10.1109/ICCCS55188.2022.10079387.
15. W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude. An empirical study of static analysis tools for secure code review. In *33rd Intl. Symposium on Software Testing and Analysis (ISSTA)*, 2024. doi:10.1145/3650212.3680313.

16 Y. Chen, A. E. Santosa, A. M. Yi, A. Sharma, A. Sharma, and D. Lo. A machine learning approach for vulnerability curation. In *17th Intl. Conf. on Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387461.

17 Y. Cheng, T. Zhang, L. K. Shar, S. Yang, C. Dong, D. Lo, S. Lv, Z. Shi, and L. Sun. VERCATION: Precise vulnerable open-source software version identification based on static analysis and LLM. *IEEE Transactions on Software Engineering*, 2025. doi:10.1109/TSE.2025.3599581.

18 J. Cohen. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

19 T. Coskun, R. Halepmollasi, K. Hanifi, R. F. Fouladi, P. C. De Cnudde, and A. Tosun. Profiling developers to predict vulnerable code changes. In *18th Intl. Conf. on Predictive Models and Data Analytics in Software Engineering*, 2022.

20 D. A. da Costa, S. McIntosh, W. Shang, U. Kuber, and A. E. Hassan. A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 2017. doi:10.1109/tse.2016.2616306.

21 Y. Ding, T. Wei, H. Xue, Y. Zhang, C. Zhang, and X. Han. Accurate and efficient exploit capture and classification. *Science China Information Sciences*, 2017. doi:10.1007/s11432-016-5521-0.

22 A. Edmundson, B. Holtkamp, E. Rivera, M. Finifter, A. Mettler, and D. Wagner. An empirical study on the effectiveness of security code review. In *5th Intl. Symposium Engineering Secure Software and Systems (ESSoS)*, 2013.

23 J. Fan, Y. Li, S. Wang, and T. N. Nguyen. Ac/c++ code vulnerability dataset with code changes and cve summaries. In *Proceedings of the 17th international conference on mining software repositories*, pages 508–512, 2020.

24 V. Freitas. Parsifa, 2025. URL: https://parsif.al.

25 S. Herbold, A. Trautsch, F. Trautsch, and B. Ledel. Problems with SZZ and features: An empirical study of the state of practice of defect prediction data collection. *Empir. Softw. Eng.*, 2022. doi:10.1007/s10664-021-10092-4.

26 T. Hinrichs, E. Iannone, T. Aladics, P. Hegedűs, A. De Lucia, F. Palomba, and R. Scandariato. Back to the roots: Assessing mining techniques for Java vulnerability-contributing commits. *ACM Trans. Softw. Eng. Methodol.*, 2025. doi:10.1145/3769105.

27 K. Hogan, N. Warford, R. Morrison, D. Miller, S. Malone, and J. Purtilo. The challenges of labeling vulnerability-contributing commits. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 270–275. IEEE, 2019.

28 S. Jeon and H. K. Kim. Autovas: An automated vulnerability analysis system with a deep learning approach. *Computers & Security*, 106, 2021.

29 B. Jiang, Y. Liu, and W. K. Chan. Contractfuzzer: Fuzzing smart contracts for vulnerability detection. In *33rd Intl. Conf. on Automated Software Engineering*, 2018.

30 M. Jiang, J. Jiang, T. Wu, Z. Ma, X. Luo, and Y. Zhou. Understanding vulnerability inducing commits of the Linux kernel. *ACM Trans. Softw. Eng. Methodol.*, 2024. doi:10.1145/3672452.

31 S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead. Automatic identification of bug-introducing changes. In *21st Intl. Conf. on Automated Software Engineering (ASE)*, 2006. doi:10.1109/ase.2006.23.

32 B. A. Kitchenham and S. Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 2007.

33 I. V. Krsul. *Software vulnerability analysis*. Purdue University, 1998.

34 J R Landis and G G Koch. The measurement of observer agreement for categorical data. *Biometrics*, pages 159–174, 1977.

35 T. H. M. Le, D. Hin, R. Croft, and M. A. Babar. Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning. In *2021 36th Intl. Conf. on Automated Software Engineering (ASE)*, 2021.

36 V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri. OpenSZZ: A free, open-source, web-accessible implementation of the SZZ algorithm. In *28th Intl. Conf. on Program Comprehension (ICPC)*, 2020. doi:10.1145/3387904.3389295.

37 F. Li and V. Paxson. A large-scale empirical study of security patches. In *2017 ACM SIGSAC Conf. on Computer and Communications Security (CCS)*, 2017. doi:10.1145/3133956.3134072.

38 F. Lomio, E. Iannone, A. De Lucia, F. Palomba, and V. Lenarduzzi. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software*, 188, 2022.

39 A. Meneely. What is a vcc? https://vulnerabilityhistory.org/articles/vccs, 2023.

40 A. Meneely, H. Srinivasan, A. Musa, A. R. Tejeda, M. Mokary, and B. Spates. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *2013 Intl. Symposium on Empirical Software Engineering and Measurement*, 2013.

41 MITRE. CWE Glossary. https://cwe.mitre.org/documents/glossary/index.html, May 2023.

42 M. Monperrus. A critical review of" automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *36th Intl. Conf. on Software Engineering*, 2014.

43 D. Nguyen, M. Tran-Duc, T. Le-Cong, T. Huynh Minh Le, M. A. Babar, and Q.-T. Huynh. VulGuard: An unified tool for evaluating just-in-time vulnerability prediction models. In *41st Intl. Conf. on Software Maintenance and Evolution (ICSME)*, 2025.

44 S. Nguyen et al. Commit-level, neural vulnerability detection and assessment. In *31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2023. doi:10.1145/3611643.3616346.

45 S. Nguyen, T.-T. Nguyen, T. T. Vu, T.-D. Do, K.-T. Ngo, and H. D. Vo. Code-centric learning-based just-in-time vulnerability detection. *Journal of Systems and Software*, 214, 2024. doi:10.1016/j.jss.2024.112014.

46 NIST. National Vulnerability Database - NVD, 2026. Accessed: 2026-08-17. URL: https://nvd.nist.gov/.

47 OffSec. Exploit Database. https://www.exploit-db.com/, 2025. Accessed: 2025-11-10.

48 A. Ozment. Improving vulnerability discovery models. In *2007 ACM workshop on Quality of protection*, 2007.

49 R. Paul, A. K. Turzo, and A. Bosu. A dataset of vulnerable code changes of the Chromium OS project. In *2021 43rd Intl. Conf. on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2021. doi:10.1109/ICSE-Companion52605.2021.00113.

50 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 *22nd ACM SIGSAC Conf. on Computer and Communications Security*, 2015. doi:10.1145/2810103.2813604.

51 J. Pokropiński, J. Gasiorek, P. Kramarczyk, and L. Madeyski. SZZ unleashed-RA-C: An improved implementation of the SZZ algorithm and empirical comparison with existing open source solutions. In *Developments in Info. and Knowledge Mgmt. for Business Apps.*, 2021. doi:10.1007/978-3-030-77916-0_7.

52 S. Quach, M. Lamothe, Y. Kamei, and W. Shang. An empirical study on the use of SZZ for identifying inducing changes of non-functional bugs. *Empir. Softw. Eng.*, 2021. doi:10.1007/s10664-021-09970-8.

53 A. Rani, R. Robbes, C. Parnin, and A. Begel. On refining the SZZ algorithm with bug discussion data. *Empirical Software Engineering*, 2024. doi:10.1007/s10664-024-10511-2.

54 T. Riom, A. Sawadogo, K. Allix, T. F. Bissyandé, N. Moha, and J. Klein. Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits. *Empir. Softw. Eng.*, 2021. doi:10.1007/s10664-021-09944-w.

807 **55** G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. Evaluating SZZ implementations through a developer-informed oracle. In *2021 43rd Intl. Conf. on Software Engineering (ICSE)*, 2021. doi:10.1109/icse43902.2021.00049.

810 **56** G. Rosa, L. Pascarella, S. Scalabrino, R. Tufano, G. Bavota, M. Lanza, and R. Oliveto. A comprehensive evaluation of szz variants through a developer-informed oracle. *Journal of systems and software*, 202, 2023.

813 **57** A. Sabetta and M. Bezzi. A practical approach to the automatic classification of security-relevant commits. In *2018 Intl. Conf. on Softw. Maintenance and Evolution (ICSME)*, 2018. doi:10.1109/icsme.2018.00058.

816 **58** S. Scalco and R. Paramitha. Hash4Patch: A lightweight low false positive tool for finding vulnerability patch commits. In *21st Intl. Conf. on Mining Software Repositories (MSR)*, 2024. doi:10.1145/3643991.3644871.

819 **59** A. Sejfia, Y. Zhao, and N. Medvidović. Identifying casualty changes in software patches. In *29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021.

822 **60** A. Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *2010 Intl. Conf. on computational intelligence and security*, 2010.

825 **61** Y. Shi, Y. Zhang, T. Luo, X. Mao, and M. Yang. Precise (un)affected version analysis for web vulnerabilities. In *37th Intl. Conf. on Automated Software Engineering (ASE)*, 2023. doi:10.1145/3551349.3556933.

828 **62** J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 2005. doi:10.1145/1082983.1083147.

830 **63** J. Sun, J. Chen, Z. Xing, Q. Lu, X. Xu, and L. Zhu. Where is it? tracing the vulnerability-relevant files from vulnerability reports. In *46th Intl. Conf. on Softw. Eng. (ICSE)*, 2024. doi:10.1145/3597503.3639202.

833 **64** X. Sun, M. Zhou, S. Cao, X. Wu, L. Bo, D. Wu, B. Li, and Y. Xiang. HgtJIT: Just-in-time vulnerability detection based on heterogeneous graph transformer. *IEEE Transactions on Dependable and Secure Computing*, 2025. doi:10.1109/TDSC.2025.3586669.

836 **65** L. Tang, L. Bao, X. Xia, and Z. Huang. Neural SZZ algorithm. In *2023 38th Intl. Conf. on Automated Software Engineering (ASE)*, 2023. doi:10.1109/ase56229.2023.00037.

838 **66** A. Trautsch, F. Trautsch, and S. Herbold. Msr mining challenge: The smartshark repository mining data. *arXiv preprint arXiv:2102.11540*, 2021.

840 **67** L. Wan. Automated vulnerability detection system based on commit messages, 2019. doi:10.32657/10220/48651.

842 **68** H. Wang, G. Ye, Z. Tang, S. H. Tan, S. Huang, D. Fang, Y. Feng, L. Bian, and Z. Wang. Combining graph-based learning with automated data collection for code vulnerability detection. *IEEE Transactions on Information Forensics and Security*, 2021. doi:10.1109/tifs.2020.3044773.

846 **69** S. Wu, B. Chen, M. Sun, R. Duan, Q. Zhang, and C. Huang. DeepVuler: A vulnerability intelligence mining system for open-source communities. In *20th Intl. Conf. on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2021. doi:10.1109/trustcom53373.2021.00090.

850 **70** Z. Wu, Y. Zhao, C. Wei, Z. Wan, Y. Liu, and H. Wang. CommitShield: Tracking vulnerability introduction and fix in version control systems. In *47th Intl. Conf. on Software Engineering: Companion Proceedings (ICSE)*, 2025. doi:10.1109/ICSE-Companion66252.2025.00082.

853 **71** L. Yang, X. Li, and Y. Yu. Vuldigger: A just-in-time and cost-aware tool for digging vulnerability-contributing changes. In *Global Communications Conf. (GLOBECOM)*, 2017.

855 **72** F. Zuo, X. Zhang, Y. Song, J. Rhee, and J. Fu. Commit message can help: Security patch detection in open source software via transformer. In *2023 IEEE 24th Intl. Conf. on Software Engineering Research, Management and Applications (SERA)*, 2023. doi:10.1109/sera57763.2023.10197730.