Back to the Roots: Assessing Mining Techniques for Java Vulnerability-Contributing Commits

TORGE HINRICHS, Hamburg University of Technology, Germany

EMANUELE IANNONE, Hamburg University of Technology, Germany and University of Salerno, Italy

TAMÁS ALADICS, University of Szeged, Hungary and FrontEndART Ltd., Hungary

PÉTER HEGEDŰS, University of Szeged, Hungary and FrontEndART Ltd., Hungary

ANDREA DE LUCIA, University of Salerno, Italy

FABIO PALOMBA, University of Salerno, Italy

RICCARDO SCANDARIATO, Hamburg University of Technology, Germany

Context: Vulnerability-contributing commits (VCCs) are code changes that introduce vulnerabilities. Mining historical VCCs relies on SZZ-based algorithms that trace from known vulnerability-fixing commits. *Objective*: Although these techniques have been used, e.g., to train just-in-time vulnerability predictors, they lack systematic benchmarking to evaluate their precision, recall, and error sources. *Method*: We empirically assessed 12 VCC mining techniques in Java repositories using two benchmark datasets (one from the literature and one newly curated). We also explored combinations of techniques, through intersections, voting schemes, and machine learning, to improve performance. *Results*: Individual techniques achieved at most 0.60 precision but up to 0.89 recall. The precision rose to 0.75 when the outputs were combined with the logical AND, at the expense of recall. Machine learning ensembles reached 0.80 precision with a better precision–recall balance. Performance varied significantly by dataset. Analyzing "fixing commits" showed that certain fix types (e.g., filtering or sanitization) affect retrieval accuracy, and failure patterns highlighted weaknesses when fixes involve external data handling. *Conclusion*: Such results help software security researchers select the most suitable mining technique for their studies and understand new ways to design more accurate solutions.

CCS Concepts: • **Security and privacy → Software security engineering**; **Vulnerability management**; • **Software and its engineering → Software configuration management and version control systems**; **Software defect analysis**; • **General and reference → Evaluation**.

Additional Key Words and Phrases: Mining Software Repositories, Software Vulnerability, Software Analytics, Machine Learning

ACM Reference Format:
Torge Hinrichs, Emanuele Iannone, Tamás Aladics, Péter Hegedűs, Andrea De Lucia, Fabio Palomba, and Riccardo Scandariato. 2024. Back to the Roots: Assessing Mining Techniques for Java Vulnerability-Contributing Commits. 1, 1 (September 2024), 41 pages. https://doi.org/XXXXXXX.XXXXXXX

Authors' Contact Information: Torge Hinrichs, torge.hinrichs@tuhh.de, Hamburg University of Technology, Hamburg, Hamburg, Germany; Emanuele Iannone, emanuele.iannone@tuhh.de, Hamburg University of Technology, Hamburg, Hamburg, Germany and University of Salerno, Fisciano, Campania, Italy; Tamás Aladics, aladics@inf.u-szeged.hu, University of Szeged, Szeged, Hungary and FrontEndART Ltd., Szeged, Hungary; Péter Hegedűs, hpeter@inf.u-szeged.hu, University of Szeged, Szeged, Hungary and FrontEndART Ltd., Szeged, Hungary; Andrea De Lucia, adelucia@unisa.it, University of Salerno, Fisciano, Campania, Italy; Fabio Palomba, fpalomba@unisa.it, University of Salerno, Fisciano, Campania, Italy; Riccardo Scandariato, riccardo.scandariato@tuhh.de, Hamburg University of Technology, Hamburg, Hamburg, Germany.

Manuscript submitted to ACM

1 Introduction

Software vulnerabilities are security weaknesses in source code that external attackers may exploit to cause loss or harm [36]. Studying and understanding them represents a fundamental challenge toward creating a sound science of cybersecurity [37]. In the recent past, multiple research communities, including the software engineering one [17, 49], have been investigating the problem of software vulnerabilities under different perspectives, contributing to the field through the definition of taxonomies [29], the understanding of how vulnerabilities are introduced and fixed [23, 36, 55, 86], how they impact source code quality and user's satisfaction [2, 27, 62, 91], and the identification of factors influencing the introduction of vulnerabilities in source code [11, 35, 52]. At the same time, several automated approaches have been devised, with notable examples in the fields of machine learning-based vulnerability prediction [39, 46, 72, 73, 80] and automated program repair [10, 28].

Such a consistent body of knowledge has often been enabled by the availability of open-source data, which provided researchers with the possibility of collecting large quantities of information concerned with the characteristics and evolution of software vulnerabilities in large ecosystems available in public repositories (e.g., GitHub). More particularly, most empirical studies and automated approaches exploited the mining of the so-called *vulnerability-contributing commits* (VCCs), i.e., code changes that introduced vulnerabilities in source code [34]. This concept represents a variant of the better-known *bug-inducing commits* (BIC), though with a focus on security vulnerabilities. Such kinds of commits are helpful in going back to the *origin* ("root") of a known issue disclosed via a vulnerability report (e.g., a CVE) to gain a greater understanding of the causes behind it.

Mining those VCCs is indeed crucial to identifying how vulnerabilities are introduced and fixed and what bad security practices developers apply, other than to feed machine learning solutions aimed at predicting new, unknown vulnerabilities being introduced. However, the task of detecting VCCs is all but trivial: File renames, tangled commits, and non-functional changes are just some challenges of mining vulnerability data effectively [34, 40]. Consequently, there might be multiple conditions biasing the conclusions of empirical studies and the performance of automated solutions. Furthermore, automated static analysis tools, such as CodeQL [26] and SonarQube [75], are not effective for retrieving VCCs. These tools primarily generate security warnings that do not directly pertain to the known vulnerabilities being analyzed. Additionally, they struggle to scale due to the large volume of commits in software repositories and the presence of files that cannot be built successfully. To face these challenges, automated VCC mining techniques have been proposed, with most of them relying on the well-known SZZ algorithm [93], which can identify bug-inducing commits using the *git-blame* mechanism on the lines modified by the known bug-fixing commit. More recently, researchers have been proposing extensions of the SZZ algorithm that target the peculiarities of software vulnerabilities to increase the accuracy of VCC detection.

Despite the notable advances in automated VCC detection, our research highlights a **lack of empirical investigations and benchmarks into the actual capabilities of the mining instruments currently available**. Extending the current knowledge on VCC detection may have relevant implications for the entire research community: Researchers may not only be provided with insights that help them estimate the bias of empirical studies and automated approaches but also with an instrument that may lead them to take an informed decision on the best mining approaches to implement in specific research contexts. We believe that an investigation into the performance of VCC detection approaches is of interest to software security researchers. Nevertheless, we also argue that practitioners, i.e., software developers, may benefit from knowing which are the reliable mining approaches to precisely find the VCCs of past vulnerabilities that affected the projects in which they are involved, increasing their awareness of mistakes to avoid in the future.

In this paper, we bridge this knowledge gap by presenting an empirical comparison of VCC mining techniques. We specifically focus on Java, in an effort to provide a thorough and language-specific analysis that captures the unique patterns and characteristics of vulnerability-contributing commits in Java-based systems. We carried out mixed-method research [16], employing a literature survey, mining software repositories, and content analysis. More specifically, we first conduct a preliminary multivocal literature review [25], finding a collection of 12 mining techniques to benchmark and a dataset that can be used as a baseline containing manually-validated VCCs. Then, we addressed two main research questions aiming at (1) assessing the quantitative performance of VCC mining techniques on the dataset found in literature and a new one we curated for this work and (2) looking at the capabilities of the techniques from a qualitative perspective in order to elicit the reasons behind their success and failure. The investigation revealed that existing techniques for mining VCCs are imprecise and exhibit different behaviors depending on the benchmark selected. Employing different kinds of combinations can make them more precise, although this comes at the expense of recall. Importantly, our findings show that combining multiple VCC mining techniques—whether through logical intersections, voting schemes, or lightweight learning-based ensembles—may lead to significantly improved precision at a relatively limited cost in recall. This motivates the use of multiple techniques in combination rather than in isolation, a practice that can enhance the reliability of mined VCC datasets for both academic and industrial applications, and can yield better trade-offs between precision and recall depending on the user's goal, e.g., building a training set for vulnerability prediction versus curating a precise dataset for qualitative analysis. As such, the findings of our study may be valuable for researchers aiming to construct more robust ground truths and for practitioners seeking to understand or trace the origins of vulnerabilities with greater confidence.

Besides, we observed that there are some recurring *fixing actions* appearing in the input vulnerability-fixing commit that can lead the techniques to more correct classification, though this is highly influenced by the benchmark dataset used. These insights may guide researchers in designing new VCC mining techniques or refining existing ones to better account for the structural patterns and contextual cues present in real-world fixing commits.

To sum up, our work provided the following key contributions:

(1) A novel dataset of manually validated VCCs that can be used to benchmark VCC mining techniques.
(2) An empirical comparison of the performance of existing VCC mining techniques and their combinations on two datasets of known vulnerabilities with manually-validated VCCs.
(3) An analysis connecting the outputs of the VCC mining techniques with the changes happening in the vulnerability-fixing commits.
(4) An online appendix [33] with all data and scripts employed in the context of the empirical study, which can be used to replicate and reproduce our work, other than building on top of our findings.

2 Research Statement

The *goal* of our investigation was to assess the performance of existing automated mining techniques for retrieving vulnerability-contributing commits of known (i.e., disclosed) vulnerabilities, with the *purpose* of providing insights into their capabilities, limitations, and possible improvements. The *perspective* of this study is mainly given through the software security researcher's point of view, who is interested in estimating the effectiveness of mining techniques for VCCs to evaluate the current support they provide when mining security-relevant data from software repositories.

To reach our goal, we conducted mixed-method research [16], mixing both quantitative and qualitative methods as explained in the following. We start by surveying the literature to identify a collection of automated mining techniques that could retrieve the commits that contributed to the introduction of a vulnerability [51]. More specifically, we first conduct a preliminary *multivocal literature survey* [25] to identify the candidate automated mining techniques. Besides, while searching for suitable techniques, we also looked for labeled datasets that have been used to evaluate them, i.e., datasets containing explicit mappings of vulnerabilities to their contributing commits. It is important to note that this survey does not aim to map the current state of the art comprehensively but serves as an instrumental step in selecting the basic instruments (i.e., techniques and datasets) for the assessment.

After this preliminary step, we proceeded with the design of the experimentation that allowed us to *compare* the techniques, reflect on their effectiveness, and experiment with mechanisms to improve them. We mapped such an experimentation onto the following two research questions:

> 🔍 **RQ$_1$.** *How effective are VCC mining techniques on common datasets?*
>
> ***RQ$_{1.1}$.*** *How effective do **individual** VCC mining techniques perform on common datasets?*
>
> ***RQ$_{1.2}$.*** *How effective do **combined** VCC mining techniques perform on common datasets?*

After the quantitative performance assessment, we went deeper into finding possible factors that could affect the retrieval capabilities of the techniques. Namely, we looked into how the changes made in the vulnerability-fixing commits affected the correctness of the output returned by the techniques.

> 🔍 **RQ$_2$.** *What are the connections between the changes in fixing commits and the outputs of VCC mining techniques?*

The literature survey is reported in Section 3, along with the collection of techniques and datasets found. The research method employed to address **RQ$_1$** is described in Section 4, while its results are reported in Section 5. The design and the results of in-depth analysis to answer **RQ$_2$** are reported in Section 6.

3 Selection of Study Objects

To find the techniques for mining VCCs and the datasets used to validate them, we conducted a *multivocal literature review* [25]. Such a process aims to find reproducible automated techniques for retrieving VCCs, leveraging any metadata associated with known vulnerabilities. We opted for a multivocal literature review for two main reasons, i.e., to (1) reduce the risk of missing relevant techniques [5] and (2) avoid the publication bias [92]. To implement such a process, we followed well-established research guidelines to conduct it [25] and the empirical software engineering guidelines defined by Wohlin et al. [87]. This study adheres to the *ACM/SIGSOFT Empirical Standards*, namely the *"General Standard"* and *"Systematic Reviews"* guidelines.[1]

Considering the goal of our study, we opted to run a *rapid review process* [13], allowing us to collect practical evidence of suitable techniques and datasets in a timely manner. Indeed, our goal was not to provide an exhaustive list of all existing techniques and datasets that could somehow be related to VCC mining, but only to identify those that are clearly related to studies involving the retrieval of VCCs.

3.1 Data Sources and Search Strategy

We selected two data sources to conduct our search, i.e., Google Scholar and Google standard search engine, as done in previous grey literature surveys [44] In preparing the search query, we started from the main keywords directly

[1] https://github.com/acmsigsoft/EmpiricalStandards

connected to our problem of interest, i.e., *"vulnerability"*, *"contributing"*, and *"commit"*. From these keywords, we considered synonyms previously used in literature that refer to the same (or a similar) concept, like *"introducing"* or *"inducing"* as alternative names for *"contributing"*. Due to the commonalities between vulnerabilities and canonical defects [12, 53, 54], we also looked for techniques originally meant to work for bugs (i.e., retrieving bug-inducing commits [93]). Hence, we also included *"bug"*, *"defect"*, and *"flaw"* as alternatives for *"vulnerability"*. In the end, we built the following search query:

Search Query

(vulnerability *OR* bug *OR* defect *OR* flaw) *AND* (contributing *OR* introducing *OR* inducing) *AND* commit

The same query was run on both Google Scholar and Google Search on March 24, 2023, inspecting each page until reaching saturation, i.e., when the results started to become completely irrelevant to our context [25]. We searched in *incognito* mode to avoid our personal search history biasing the results. The first set of results comprised 402 entries from Google Scholar and 113 entries from Google Search, totaling 515 results.

3.2 Eligibility Criteria

Our goal was to select only the most appropriate pieces of work that could be easily employed in our context. Hence, we discarded the results that violated at least one of the following criteria:

- C_1 The result can be downloaded from publisher websites (e.g., ACM Digital Library, IEEExplore, ScienceDirect)[2], university archives, or other pre-print open archives (e.g., ArXiv, HAL).[3]
- C_2 The result has not already been found in previous entries (i.e., not duplicated).
- C_3 The result is written in English.
- C_4 The result is about retrieving commit-related data from software repositories.

The adherence to the eligibility criteria was assessed by the first two authors of this paper, who equally split their workload. For those cases where it was difficult to assess whether the result was relevant (especially according to criterion C_4), they jointly checked those cases until they reached an agreement. The process required 30 person-hours, which terminated with 92 relevant results. The complete list of resources after the search query and those resulting after applying the eligibility criteria can be found in our online appendix [33].

The two authors then proceeded to map the **automated techniques** used in the 92 results to **retrieve the origin of a bug or vulnerability**. In this collection, all the results were scientific papers, some of which had this exact goal, i.e., presenting novel techniques to mine bug-inducing or vulnerability-contributing commits. Nevertheless, the authors also found papers presenting empirical studies on the life of bugs (or vulnerabilities), inspecting what happens between their introduction and their fix, or also studies elaborating on the challenges and limitations of mining commits that introduce and fix bugs or vulnerabilities. Since they did not know whether this second body of papers could introduce new techniques as a secondary contribution, they inspected them in full. At this point, the authors created a *catalog* of unique techniques. Namely, if two papers used the same technique (e.g., one introduced it for the first time, and the second one used it without changes), the technique was only counted once. At the same time, a conservative approach we adopted: If a paper uses a technique from another paper and makes non-negligible changes, that technique, even if

[2] Whenever possible, the existing agreements between the authors' affiliations and publishers were used to access paid publications. Otherwise, the pre-preprint versions were considered, if available.
[3] This criterion excludes results that do not provide any scientific contribution, such as bug reports, library documentation pages, and Q/A discussions.

unnamed, is counted as a separate technique (even if that paper did not have the explicit intention of introducing a new technique). There were also papers presenting machine learning-driven prediction models that detect commits likely introducing a bug or a vulnerability. Such models require training and testing on labeled data, i.e., a ground truth that determines the correct set of introducing commits from which the model(s) learned. In such cases, the authors considered the technique that was used to build the ground truth dataset. The prediction model itself was not considered as a technique as they could only flag whether a commit is potentially contributing to an unknown security issue—indeed, the goal of our study was to find the origin of *known vulnerabilities*.

Afterward, the authors selected only the techniques whose source code was publicly accessible (through online appendices or code repositories) so that they could be relaunched without re-implementing from scratch—which could have introduced drifts from the original technique. At the same time, they also kept track of the datasets used to validate it. In particular, they were searching for datasets of *human-validated vulnerability-contributing commits* linked to the related known vulnerabilities, ignoring datasets related to general bugs. This part required 50 person-hours, ending with 17 unique techniques and one suitable dataset of validated VCCs. The complete list of techniques can be found in our online appendix [33].

3.3 Techniques and Datasets Found

The source code of the 17 candidate techniques was assessed to ensure they could be launched and accept data of known vulnerability as input without problems, making minor adjustments if needed to make the technique run without errors, ensuring the changes did not alter the core logic in any way. This phase revealed that five techniques, i.e., Locus [83], V0Finder [88], PR-SZZ [6], Neural-SZZ [78] and Sem-SZZ [79] could not be used. Locus and PR-SZZ could not be employed for vulnerabilities with the provided implementation as they strictly require bug reports or pull requests, respectively, to work properly. The replication for V0Finder failed due to missing external dependencies that we could not solve or circumvent. In detail, V0Finder was built for C/C++ and uses ctags for indexing the source files. Porting this to a generalized version that also works with Java code caused errors that could not be resolved without extensive effort. Moreover, approaches such as Neural-SZZ [78] and Sem-SZZ [79] initially appeared as valid candidates but were ultimately excluded as replicating them would have required substantial modifications to the original code. In particular, Sem-SZZ is not presenting a tool, but rather a collection of scripts that are closely tailored to the research questions of the original work, which makes it hard to extract the core approach without heavily modifying the code base—and thus, risking the alteration of the behavior. The remaining 12 techniques were launched on example vulnerability data, confirming that they could be run without issues.

Coincidentally, they all share the same base mechanism, i.e., the SZZ algorithm [93]. In essence, starting from a bug-fixing commit (found in bug reports or from other sources), SZZ retrieves the commits that “induced” that fix—i.e., led to a problem that the fixing commit solved—by looking at the changes made, i.e., the lines added, deleted and modified. Such commits have also been referred to *‘fix-inducing’*, *‘bug-inducing’*, or *‘fix-introducing’* commits—here we treat all of these as equivalent ways to refer to commits that made real contributions to the emergence of a bug. Most of these techniques were used for retrieving information for change classification [42] or defect prediction [21, 24, 38, 41] problems. Yet, they have also been applied in the software security domain, adopting minor or major adjustments to make them retrieve commits contributing to a vulnerability [3, 36, 46, 60, 90]—leveraging the fact that vulnerabilities share some characteristics with canonical bugs [12, 53, 54]. In the following, we describe how each technique works—as presented in the original paper—and how it was reproduced. Table 1 summarizes the techniques selected.

Table 1. Selected techniques for mining vulnerability-contributing commits (VCCs) resulting from our literature review.

Name	Reference	How It Works	How We Run It
B-SZZ	Śliwerski et al. [93]	`git annotate` on lines deleted in fixing commits.	Implementation in PySZZ.
AG-SZZ	Kim et al. [43]	Annotation graph traversal from lines deleted in fixing commits. Ignores comments, blanks, and other aesthetical changes.	Implementation in PySZZ.
MA-SZZ	da Costa et al. [18]	Same as AG-SZZ but ignores meta-changes.	Implementation in PySZZ.
R-SZZ	Davies et al. [19]	Same as MA-SZZ but selects the most recent commit only.	Implementation in PySZZ.
L-SZZ	Davies et al. [19]	Same as MA-SZZ but selects the largest commit only.	Implementation in PySZZ.
RA-SZZ	Neto et al. [56]	Same as MA-SZZ but ignores lines involved in refactorings.	Implementation in PySZZ with RefactoringMiner 2.0.[‡]
DJ-SZZ	Williams and Spacco [85]	Same as AG-SZZ but ignores semantic-preserving changes.	Implementation in SZZUnleashed `79f369fe` revision.[*]
VF-SZZ[†§]	Perl et al. [60]	Same as B-SZZ but ignores changes to non-source files and blames the lines around new hunks.	Implementation in archeogit `v0.3.0-alpha`.[**]
VCC-SZZ[†§]	Iannone et al. [36]	Same as VF-SZZ but ignores build and test files, comments, blanks, merges, and blames more lines around new hunks.	Implementation in the appendix of [36].
RS-SZZ[†§]	Aladics et al. [1]	Same as DJ-SZZ but selects the N most relevant candidate commits according to a *relevance score*.	Implementation in the appendix of [1] with $N = 1$.
V-SZZ[§]	Bao et al. [3]	Same as B-SZZ but repeats `git blame` until reaching the origin commit. Ignores comments, blanks, aesthetical changes, cherry-pickings, and merges.	Integration of the appendix of [3] within PySZZ.
TC-SZZ	Lyu et al. [47]	Traces all commits in the change history of lines modified or deleted in bug-fixing commits.	Implementation in the appendix of [47].

[†] Names assigned in this work.
[‡] Refactoring Miner: https://github.com/tsantalis/RefactoringMiner/releases/tag/2.0.0
[§] Designed specifically for vulnerabilities.
[*] SZZUnleashed: https://github.com/wogscpar/SZZUnleashed/commit/79f369fe
[**] archeogit: https://github.com/samaritan/archeogit/releases/tag/v0.3.0-alpha

B-SZZ (*'Base SZZ'*). The original SZZ algorithm was proposed by Śliwerski, Zimmermann, and Zeller (hence, the algorithm's acronym) [93]. The original approach first links a given bug report (e.g., from Bugzilla) to its fixing commit by looking for an explicit mention of the bug report identifier in commit messages, e.g., *'Fixed Bug 42233'*. After finding the bug-fixing commit, `git diff` is run to obtain the list of changed lines in each non-binary file modified in the commit, made only of added (+) and deleted (−) lines—indeed, a modified line is expressed as if it was deleted and re-added in a new form. Then, `git annotate`[4] is executed on each modified file to obtain the commits that last modified the lines deleted in the fixing commit. Such annotated commits are candidate fix-inducing commits since they added the lines that were ultimately deleted in the fixing commit, likely being responsible for the bug. The final set of fix-inducing commits is made only of commits made before the bug was reported, discarding all those happening after the bug report date. After two decades since the release of this algorithm, two key changes have been made in modern-day re-implementations, such as in PyDriller [76] or SZZUnleashed [7]. First, the bug report-commit link is omitted, as the fixing commit can be retrieved with more sophisticated mechanisms [45, 70, 77, 89]. Second, `git annotate` is replaced with the equivalent `git blame`[5] command.

▶ **Reproduction of B-SZZ.** PySZZ [67] provides a built-in implementation of B-SZZ.

[4] `git annotate` documentation: https://git-scm.com/docs/git-annotate
[5] `git blame` documentation: https://git-scm.com/docs/git-blame

AG-SZZ (*'Annotation Graph SZZ'*). An extension of the original B-SZZ was proposed by Kim et al. [43] to reduce the number of false positives it produces under certain corner cases. AG-SZZ relies on *annotation graphs*, i.e., a data structure that maps the lines modified among the commits, able to track each line's change history in a file. Each node represents a line in a certain revision, and the edges connect nodes of different revisions, indicating that a line originates from another by modification or movement. This data structure allows one to (i) find the function or method to which the lines belong and (ii) ignore irrelevant changes to comments, blank lines, and other aesthetic modifications. In the end, this graph consists of N disjoint sets of nodes, each representing the lines in the N file's versions. Hence, AG-SZZ builds the annotation graph of each file modified in the bug-fixing commit at the line level instead of relying on the raw output from `git annotate` (or `git blame`). After building the graph, it performs a backward depth-first search from the lines changed in the fixing commit until reaching the first commit that introduced them. Unlike B-SZZ, it does not aim to find the "latest" changes that contributed to the bug but pushes toward the "origin" of the bug.

▶ **Reproduction of AG-SZZ.** PySZZ [67] provides a built-in implementation of AG-SZZ.

MA-SZZ (*'Meta-change Aware SZZ'*). An extension of AG-SZZ was proposed by da Costa et al. [18] to overcome its limitations when dealing with *meta-changes*. A meta-change is a commit that does not represent a real modification to the source code, either because it was already made in other commits or because it only modified file metadata. Thus, it is unlikely to contribute to the insertion of a bug. Specifically, MA-SZZ builds on top of AG-SZZ but also ignores commits that copy changes from one branch into another—i.e., merge commits—and commits that change execution permissions to files.

▶ **Reproduction of MA-SZZ.** PySZZ [67] provides a built-in implementation of MA-SZZ.

R-SZZ (*'Recent SZZ'*). Davies et al. [19] proposed a filter for any SZZ-based algorithm for selecting only the most recent commit among the set of candidate bug-inducing commits. Hence, this approach only returns one bug-inducing commit, i.e., the commit that made the final contribution to the bug before "exposing" it.

▶ **Reproduction of R-SZZ.** PySZZ [67] provides a built-in implementation made on top of MA-SZZ.

L-SZZ (*'Large SZZ'*). Davies et al. [19] proposed another filter for any SZZ-based algorithm for selecting only the largest commit—measured with the number of modified files—among the set of candidate bug-inducing commits. Like R-SZZ, this approach only returns one bug-inducing commit, i.e., the commit that contributed the most to the bug.

▶ **Reproduction of L-SZZ.** PySZZ [67] provides a built-in implementation made on top of MA-SZZ.

RA-SZZ (*'Refactoring Aware SZZ'*). An extension of MA-SZZ was proposed by Neto et al. [56] to further reduce the number of false positives by ignoring the changes made by refactoring. Such code transformations should not actually contribute to a bug, as they should not affect the modified code's external behavior. Thus, when building the annotation graph, RA-SZZ ignores the lines directly involved in a refactoring operation in the bug-fixing commit. The refactoring operations can be detected automatically via tools like RefDiff [74] or RefactoringMiner [81].

▶ **Reproduction of RA-SZZ.** PySZZ [67] provides a built-in implementation of RA-SZZ, which employs RefactoringMiner version 2.0 to detect the refactored code.

DJ-SZZ (*'DiffJ SZZ'*). Williams and Spacco [85] proposed an extension of AG-SZZ to ignore the changes that do not affect the code behavior, like non-executable lines (e.g., `import` statements), variables renaming, or parameters reordering. DJ-SZZ detects such changes leveraging a DiffJ,[6] a syntax-aware diff tool for comparing Java files. The

[6] DiffJ repository: https://github.com/jpace/diffj

original implementation of DJ-SZZ has never been released, but it has been re-implemented within SZZUnleashed [7], an open-source command-line tool. This version differs from the original approach as it (i) is built on top of RA-SZZ and (ii) does not filter out non-impactful changes due to the absence of cross-language parsers to detect such changes.

▶ **Reproduction of DJ-SZZ.** SZZUnleashed [7] provides a built-in implementation of DJ-SZZ.

VF-SZZ (*'VccFinder SZZ'*). Perl et al. [60] designed a heuristic method to build a labeled dataset of VCCs to train and test their machine learning prediction model (called *VccFinder*) to predict commits potentially introducing an unknown vulnerability. This heuristic is essentially a modified version of B-SZZ as it relies on the blaming of deleted lines in files touched in the fixing commit—ignoring documentation files. However, one major change has been introduced that extends the number of cases in which the `git blame` is run; namely, the method also blames the *context* around each "continuous block" (i.e., hunk) of added lines of code. The context consists of one line before the first line in the block and one line after the last one. The rationale behind this idea, as described in the original paper [60], was to enlarge the set of vulnerability-contributing commits retrieved by adhering to the assumption that security fixes often consist of adding extra code just before an access to a variable or after a function call [9, 11]. Hence, blaming the lines around the new code blocks might help go back to the commit that had the chance to add these checks but did not, and so, contributing to the vulnerability. This method was not disclosed as a variant of SZZ, but it leverages the same core mechanism. Since this method had no specific name, we refer to VF-SZZ for easy reference throughout the rest of the paper. The original implementation of this approach has never been released, but it has been re-implemented within archeogit, an open-source command-line tool.[7]

▶ **Reproduction of VF-SZZ.** archeogit provides a built-in implementation of VF-SZZ.

VCC-SZZ (*'Vulnerability-Contributing Commit SZZ'*). Iannone et al. [36] designed a similar method to the one by Perl. et al. [60] (i.e., VF-SZZ here called). This technique essentially behaves like VF-SZZ with some adjustments: (1) it ignores build and test files, (2) it does not blame deleted empty lines and comments, (3) it skips merge commits during the traverse of commits, and (4) blames three context lines (instead of one, as done by in VF-SZZ) around continuous blocks of added, and (5) does not blame any context line of continuous blocks that entirely add new functions or methods. The fifth adjustment follows the rationale that functions and methods can be added anywhere in a file; therefore, they do not have a real context around as the lines before and after function/method declarations are likely blank lines or the start or end of another function/method declaration. Just as for VF-SZZ, we refer to this method with a custom name, i.e., VCC-SZZ, for easy reference throughout the rest of the paper.

▶ **Reproduction of VCC-SZZ.** The appendix of its original paper contains the executable scripts to run VCC-SZZ.

RS-SZZ (*'Relevance Score SZZ'*). Aladics et al. [1] re-adapted DJ-SZZ for mining vulnerability-contributing commits from vulnerability-fixing commits. Given a set of commits fixing a given known vulnerability, it first runs DJ-SZZ—in its SZZUnleashed flavor—and then ranks the obtained candidate VCCs according to a *relevance score*, measuring how much a VCC contributed to the vulnerability. Specifically, for each file changed in a VCC, RS-SZZ computes the similarity of such file with its version in the fixing commit (i.e., the ratio of identical lines on the total lines) multiplied by the ratio of changes made on that file on the total changes on all the files changed in the VCC. All the contribution scores are summed to form the VCC's relevance score. After ranking the candidate VCCs, it selects the most N relevant commits, where N is chosen arbitrarily.

[7] archeogit repository: https://github.com/samaritan/archeogit

Reproduction of RS-SZZ. The appendix of its original paper contains the executable scripts to run RS-SZZ. We made it return the most relevant VCC ($N = 1$).

V-SZZ (*'Vulnerability SZZ'*). An extension of B-SZZ was proposed by Bao et al. [3] aiming at tracing the likely origin of vulnerabilities. V-SZZ consists of repeated executions of `git blame` on the lines modified in the vulnerability-fixing commit until reaching the first commits that added those lines. The set of commits that introduced those lines constitutes the candidate vulnerability-introducing commits. V-SZZ is aided by an AST-based line mapping technique [22], that ignores empty and comments lines, aesthetical changes, cherry-picked code (i.e., changes copied from commits in other branches), and merge commits.

Reproduction of V-SZZ. The appendix of its original paper contains the executable scripts to run V-SZZ.

TC-SZZ (*'Tracing-Commit SZZ'*). TC-SZZ [47] represents another variant of the SZZ algorithm, aimed at enhancing the identification of bug-inducing commits by conducting a comprehensive analysis of the change history associated with modified lines. Conventional SZZ techniques typically rely on a single invocation of `git blame` to determine the commit directly preceding a bug-fixing change. In contrast, TC-SZZ employs an iterative application of `git blame` to track the entire history of each altered or removed line, progressing back to the original commit. This method results in a sequence of potential commits, including intermediate "descendant" commits and the initial commit, all of which could have contributed to the introduction of the bug. The overarching goal of TC-SZZ is to identify bug-inducing changes that are obscured deeper within the change history, thereby enhancing recall and offering a richer context for debugging intricate systems such as the Linux kernel (the application for which TC-SZZ has been designed).

Reproduction of TC-SZZ. The appendix of its original paper contains the executable scripts to run TC-SZZ.

To make the identified mining techniques functional for our study, we made minor adjustments to let them accept a set of vulnerability-fixing commits of a known vulnerability as input and return the set of commit hashes deemed to contribute to that vulnerability. At the same time, the adjustments we made did not fix or improve any of the heuristics adopted by the techniques. Namely, we did not place additional filters to reduce the noisy result, i.e., falsely attributed VCCs, or handle special circumstances, such as history rewrites. Hence, if a technique is already able to withstand cases like refactorings, such as RA-SZZ [56], we retain its output as is, which is expected to suffer from fewer false positives. The goal of this stage of the study was to assess the current performance of the existing techniques without any external intervention, which would require a dedicated follow-up study. Table 1 reports how we run the techniques to answer the main research questions.

Pre-Study – Techniques. A total of 12 techniques can be employed to mine VCCs. All of these leverage the same core mechanism of SZZ, i.e., they start from the fixing commits to find the likely set of commits that "induced" those fixes. Four were specifically designed for vulnerabilities (i.e., VF-SZZ, VCC-SZZ, RS-SZZ, and V-SZZ), while the others inherently work for traditional bugs.

The literature search revealed that four out of 12 techniques have been developed to mine VCCs rather than bug-inducing commits. Therefore, we inspected how these four techniques have been validated, i.e., how their effectiveness has been assessed and which data were employed. We found that only three out of four, i.e., VF-SZZ, VCC-SZZ, and V-SZZ, have been validated employing standard performance metrics. Nevertheless, the first two only had their precision assessed, i.e., the resulting set of VCCs was subject to a manual inspection to determine whether the technique had made no false positive classifications. Such an assessment prevents the computation of false negatives and other metrics like the recall. Only V-SZZ was validated by adopting the common performance metrics for information retrieval algorithms, i.e., precision, recall, and F1-score. To do so, the authors prepared an oracle set of VCCs that V-SZZ was expected to find. Such a set consisted of a sample of 72 vulnerabilities selected from Project-KB [63], each containing the link to the validated set of fixing commits. Project-KB consists of more than 200 real-world open-source Java projects affected by more than 600 publicly disclosed vulnerabilities. The dataset was curated manually by its original authors to link each vulnerability to its corresponding publicly accessible fixing commits. Hence, each entry in the dataset represents a known vulnerability described by (1) its CVE (Common Vulnerabilities and Exposure) identifier and (2) a set of URLs pointing to the fixing commit made on the project repository to patch the vulnerability.

By following the model proposed by Rodríguez-Pérez et al. [66], the authors leveraged any information related to the vulnerability—e.g., the CVE description, the fixing commit message—and went back in previous commits found with `git blame` on the lines modified in the fixing commit. Then, each commit matched was inspected to assess whether the vulnerability was already there. If not, they repeated the `blame` on the matched line and navigated until they found the originating commit. According to the Rodríguez-Pérez et al. model [66], only one VCC can exist; hence, if multiple candidate VCCs were found, only the earlier one is flagged as the real VCC. The replication package of V-SZZ's paper [3] contains the labeled dataset that we employed in this study to assess the effectiveness of similar VCC mining techniques. Section 4.2 describes how we reviewed the dataset and prepareeit d for our purpose.

Pre-Study – Datasets. Only one of the four techniques specifically designed to mine VCCs, i.e., V-SZZ, employed and released a labeled dataset with VCCs mapped to vulnerabilities.

4 Empirical Comparison Design (RQ$_1$)

4.1 Experimental Subjects

We assessed the performance of the 12 reproducible techniques individually, running each in isolation from the others, and checked whether they could retrieve the expected set of VCCs linked to a given vulnerability. Each technique was run on the labeled dataset used to validate V-SZZ (henceforth, the "**V-SZZ dataset**") [3] and another dataset designed for this study, both described in Section 4.2 This initial comparison was necessary to answer **RQ**$_{1.1}$. We faithfully run the techniques following the replication steps described in Section 3. Since each technique was based on a variant of the SZZ algorithm, each required that each vulnerability be linked to its set of fixing commits to be executed successfully.

Afterward, to address **RQ**$_{1.2}$, we merged the capabilities of each technique to form new ones consisting of the *AND* combination of their result sets. Namely, given two techniques A and B returning the set of contributing commit VCC$_A$ and VCC$_B$ of a given vulnerability, the combined results consist of the intersection of the two sets, i.e., VCC$_A \cap$ VCC$_B$. Following this strategy, we made groups of N techniques (with $N = 2...5$) and treated them as separate ones. Then, we removed any redundant combination caused by symmetric and reflexive relationships, leading to 55 pairs, 165 triples, 330 quadruple-wise combinations, and 462 quintuple-wise combinations. In the end, we obtained a total of $1,012$ combined techniques. Our empirical comparison focused only on combinations with $N <= 5$, excluding higher-order combinations (with $N > 5$), as we observed that the intersection of multiple techniques caused the precision to drop as the techniques kept being combined, explaining why we stopped at $N = 5$. The results for the left-out combinations are reported in our online appendix [33].

Additionally, we added two **special techniques** employing all 12 individual techniques. The first is governed by a *voting mechanism*, i.e., a commit is flagged as a VCC if it was flagged as VCC by the largest number of individual techniques (ex aequo). For example, if commits α and β are flagged as VCC by four individual techniques and commits

γ and δ by two techniques, the final set is made only by commits α and β. Both α and β were flagged as VCC as they were flagged by the same maximum number of techniques, i.e., four. The second special technique is governed by a *machine learning classifier* that uses the techniques' opinions as the features to decide whether a given commit should be flagged as a VCC. We experimented with three popular traditional learning algorithms, i.e., Random Forest (RF) [8], Naïve Bayes (NB) [71], and K-Nearest Neighbors (KNN) [15], all trained and tested on the entirety of commits flagged as VCC by at least one of the individual techniques. Each commit was represented as an 12-length binary feature vector, where the i-th feature denotes the prediction made by the i-th technique—i.e., 1 if the technique marked the commit as VCC (a.k.a. positive instance), 0 otherwise (a.k.a. negative instance). The target variable (forming the ground truth) is essentially the actual nature of the commit, i.e., 1 if it is a VCC, 0 otherwise, directly stemming from the same input dataset the individual techniques used (see Section 4.2). The evaluation was made using a stratified 10-fold cross-validation technique: we performed 10 rounds of validations each time using 90% instances for the training and the remaining 10% for testing the model performance.

4.2 Dataset Preparation

To compare all the techniques participating in the experimentation, we had to provide a "common ground" where all could be run for a fair comparison. The V-SZZ dataset [3], found in the context of the **pre-study** (Section 3.3), conformed to our requirements and appeared to be used as-is. The dataset was released as a JSON file containing the fixing commits of each vulnerability. Each fixing commit is then associated with a set of a few commits, among which the real VCC is flagged explicitly. After checking its actual content, we found that two out of the 72 vulnerabilities, i.e., CVE-2013-0239 and CVE-2018-17192, did not really contain any references to their VCCs—i.e., none of the commits reported are flagged as true VCCs. Hence, we removed those vulnerabilities from the dataset as they could not provide any ground truth for the evaluation, resulting in 70 vulnerabilities. At this point, we analyzed the filters adopted by the authors to prepare the dataset, which discarded the vulnerabilities whose fixing commits were (1) made only of new lines [64] or (2) deleting more than five lines. The rationale behind the former stands in the need for deleted lines to run the `git blame` and find the candidate set of VCCs; while for the latter, the authors explain the difficulty in the presence of irrelevant lines of code not connected to the real fix, which would induce the `git blame` to retrieve too many commits and increase the manual workload too much. Besides, all vulnerabilities without reference to the affected versions (e.g., linking the Common Platform Enumeration and the `git` tags) were also discarded. Despite understanding the design decisions made, we suspect such filters caused the removal of too many relevant vulnerabilities, involuntarily creating a different problem with less adherence to the real one. In other words, we believe this subset of vulnerabilities sampled from Project-KB [63] provides a partial view of the characteristics of the vulnerability fixing process.

Recognizing the value and importance of the V-SZZ dataset, we also compared the selected mining techniques on a different dataset of labeled VCCs with the three filters not applied. To this end, we randomly selected 100 vulnerabilities from the Project-KB dataset—the same source used for the V-SZZ dataset—and conducted a *manual inspection activity* to retrieve the set of commits that contributed to these vulnerabilities.

Before starting, each vulnerability was linked with the content of the CVE Record mined through the National Vulnerability Database (NVD).[8] Such information was deemed fundamental for understanding the issue better and enabling the search for the contributing commit. The CWE (Common Weakness Enumeration) information was also

[8] NVD website: https://nvd.nist.gov/

included if available. Then, we designed the activity into six steps, which were reiterated for each vulnerability sampled.

Step 1. The inspector examined all the metadata linked to the vulnerability, i.e., the CVE Record and the CWE, to understand the context of the issue and what to focus on while looking for the VCC.

Step 2. A set of Python scripts was used to facilitate a set of repetitive actions, i.e., opening the GitHub pages of the fixing commits linked to the vulnerability, which displays the diff between the fixed version and the previous one, assumed to be vulnerable, the commit message, and other meta information (the timestamp, the author, etc.).

Step 3. The inspector analyzed the changes made in the fixing commits using their experience in software development, security vulnerabilities, and mining software repositories. In particular, the inspector was aware of the recurring actions taking place when a certain type of vulnerability is fixed. For example, they are aware of the fact that *input validation* vulnerabilities (e.g., CWE-20) are often patched by writing new code to entirely prevent the use of invalid data (e.g., by adding an `if` guard statement) or invoking a routine to sanitize the invalid data (either using a custom procedure or reusing one from a library or framework) [9, 11]. Such knowledge helped localize the lines directly responsible for the vulnerability; the inspector marked them as “suspect vulnerable”. It is worth noting that not all the lines deleted in a fix commit are direct responsible of the vulnerability, as (1) there might be other irrelevant changes in the commit [31] and (2) several vulnerability types are commonly fixed by adding new code, nut just deleting the “bad code” [9, 11]. This step produced detailed and granular information about the fixed location, augmenting the information missing from Project-KB.

Step 4. The repository was cloned locally and checked out to the last vulnerable version—i.e., the version just before the fixing commit. Then, the `git blame` command was run on all the lines the inspector marked as suspect vulnerable, allowing the retrieval of the commits that last modified the marked lines. The inspector considered the cases where the files could change names while traversing their history and tuned the `git blame` command first, detecting cases of file renaming or moving to other directories. The inspector could mark any line from the vulnerable version of the code, not only those modified in the fix. This marks a clear difference with the approach adopted by Bao et al. [3] when building their dataset, where all and only the deleted lines were subject to the `git blame`. Besides, the inspector did not follow their stopping criterion, i.e., the repeated runs of `git blame` terminated as soon as there was clear evidence the vulnerability was introduced, without necessarily returning to the originating commit.

Step 5. The GitHub pages of the blamed commits were opened using the Python scripts used for the fixing commits, allowing the inspector to analyze the changes and assess whether they actually contributed to the introduction of the vulnerability. If a commit did not contribute to the vulnerability in any way, the `git blame` command was re-run on the same line retrieved by the previous run of `git blame`. This was reiterated until the commit that created the file with the marked line was reached.

Step 6. The entire set of commits after all the `git blame` executions was further re-inspected to assess which was the right VCC. Differently from the V-SZZ dataset relying on the model proposed by Rodríguez-Pérez et al. [66], the inspector did not limit to flagging a single commit as VCC as it is quite common that vulnerabilities are introduced due to many changes [36, 51].

The labeling activity was performed by a security-trained researcher, requiring 250 person/hour effort due to the complexity of understanding the specific issue caused by the vulnerability, comprehending the fixing commit(s), and analyzing the candidate commits to determine which was a VCC. Ultimately, the dataset comprised 100 VCCs, one per

Table 2. Key characteristics of the two benchmark datasets. 'V' indicates the number of vulnerabilities.

Characteristic	V-SZZ Dataset (V = 70)				Our Dataset (V = 100)			
	Avg	Med	Min	Max	Avg	Med	Min	Max
Per Project								
★ Github Stars	9,725	2,349	4	557,798	7,240	2,184	24	57,799
Vulnerabilities	1.71	1	1	13	1.41	1	1	7
Vulnerabilities w/ CWE	-	-	-	-	1.34	1	1	7
Per Vulnerability								
# Fix Commits	1.31	1.0	1	3	1.89	1.0	1	13
+ lines in Fix	77.33	12.5	0	618	234.41	49.0	0	4,655
− lines in Fix	4.04	3.0	1	40	58.22	9.0	0	2,494
# VCCs	2.14	2.0	1	7	1.00	1.0	1	1
+ lines in VCC	22,095.91	181.0	1	35,0029	35,423.80	411.0	0	644,212
− lines in VCC	824.93	11.0	0	23,452	717.69	25.0	0	56,793
📅 Timespan (year, rounded)	2017	2016	2011	2018	2017	2018	2008	2020

vulnerability. To mitigate the risk of making mistakes during this labeling activity, another security-trainer researcher was involved in running the same process on a random subset of 30 vulnerabilities. The second inspector found the same vulnerabilities as the first inspector, except for one (CVE-2016-3082), on which they agreed with the first inspector's findings. In the end, the reliability between the two inspectors measured with Cohen's kappa [14] was 0.96, indicating very strong agreement. Our labeled dataset of VCCs can be found in the online appendix [33].

After completing, we measured some key characteristics of both the V-SZZ dataset and ours, reported in Table 2. We observe that for the V-SZZ dataset, the 72 vulnerabilities affected a total of 41 projects, with an average of 1.76 per project, while for our dataset, the 100 vulnerabilities affected 71 projects, with an average of 1.41. However, two vulnerabilities had to be removed due to missing vulnerability-introducing commits. This finding was also previously reported in other papers. In the V-SZZ dataset, 82% of vulnerabilities had exactly one fixing commit, with only in rare cases (1) reaching three fixing commits. Similar numbers were also seen for our data: only 8.4% of vulnerabilities had more than one fixing commit, reaching one extreme case of 13 fixing commits in CVE-2018-8009 affecting Hadoop.

Additionally, we enriched the V-SZZ dataset with the corresponding CWE information, which was not provided in the original dataset. This leads to 25 unique CWE identifiers. On the other hand, for our dataset, we identified 49 unique CWE identifiers, indicating more diversity compared to the V-SZZ dataset. In both datasets, fixing commits predominantly consist of newly added lines, i.e., 77.33 on average in the V-SZZ dataset and 243.41 in ours, which is also noticeably higher than the number of deleted lines. The discrepancy between the two datasets about the average number of added lines in fixing commits supports our choice not to discard fixing commits made of new lines only, like in the V-SZZ dataset: their absence drifts the distribution of fixing commits too far from the real one.

Looking at the number of VCCs, we see that more than half of the vulnerabilities needed more than one VCC to introduce it, reaching seven in one case. The same did not happen for our dataset, where a single VCC fully introduced all vulnerabilities. We point out that the inspector was not instructed to limit the search to a single VCC, as, in general, it is known that multiple VCCs can be necessary to introduce a vulnerability [51]. Nevertheless, the inspector found no cases requiring more than one VCC in our dataset. We also found that in some cases (i.e., 4 out of 70 in the V-SZZ dataset and 9 out of 100 in ours), the VCC corresponded to the repository's *initial commits*. With a closer inspection, we observed that all these commits consisted of migrating the entire project from a different versioning system, e.g., `Subversion`. This inevitably affected the observed distribution of added lines, creating many outliers and explaining the high average size (more than 22k or 35k, depending on the dataset). For this, it was impossible to retrieve the real VCC as the original change history analysis was lost, forcing us to consider this initial commit as the only possible VCC.

Both datasets were used to build the instances for training and validating the machine learning models. For the V-SZZ dataset, the 70 VCCs from the ground truth formed the positive instances, while all VCCs mistakenly flagged by the 12 techniques formed the negative instances, resulting in 282. The same idea was applied to our dataset, obtaining 100 positive and 2.731 negative instances.

4.3 Performance Assessment

To assess the performance of the techniques, we employed two common metrics that have been used in the past to compare SZZ-like approaches, i.e., *precision* and *recall* [57, 68]. Such metrics represent the traditional mechanism used to evaluate information retrieval algorithms. The former measures the number of VCCs correctly classified over the total number of predictions performed; the latter complements precision by indicating the amount of correctly classified VCCs over the total number of ground truth VCCs. In other words, the precision of a technique t_i equals the number of VCCs correctly predicted by t_i divided by the total number of VCCs returned by t_i. At the same time, its recall equals the number of VCCs correctly predicted by t_i divided by the total number of VCCs in the ground truth.

We could assess some key aspects of the evaluated techniques through these performance indicators. In particular, the precision allowed us to assess their overall error in detecting VCCs, i.e., their tendency to make erroneous predictions. This is likely the most insightful indicator to consider in our case. Indeed, most researchers employ SZZ-like algorithms to feed prediction models [21, 46] or investigate properties through empirical studies [18, 68]. Hence, should a mining technique have low precision, it would improperly flag several vulnerability-free commits as vulnerability-contributing (i.e., false positives), biasing any conclusions researchers may draw. On the other hand, the recall could measure the number of real VCCs the technique actually retrieves; The recall provides researchers with orthogonal information: it indicates the number of VCCs the algorithm missed, possibly informing them that their approach would provide statistically invalid samples. On top of these two metrics, we computed the *F1 score* (a.k.a. F-measure) that computes, i.e., the harmonic mean of precision and recall, providing an aggregated indication of the balance between the two.

Due to the use of the 10-fold cross-validation for the ML-driven combined technique, we could test whether the three selected classifiers were statistically better than others on a given performance metric. Hence, we ran paired Wilcoxon signed-rank tests [84] to compare the ML models on the same 10-fold splits. Specifically, we ran a one-sided test for each pair of ML models (e.g., RF vs. KNN) for each indicator metric (precision, recall, F1 score) for both datasets, resulting in 18 test executions altogether. The null hypothesis H_0 ("the performance scores of *ML-A* are not statistically larger than the performance scores *ML-B*") was tested with a significance level of 0.05, adjusted to 0.005 after applying the Bonferroni correction [4]. We remark that the same statistical procedure could not have been carried out for the individual mining techniques or the other combinations since they only provide a single score per metric. Namely, each technique had only one precision score, one recall score, and one F1 score; without distributions of scores to compare, no statistical test can be run.

Lastly, we computed the ratio between the number of predicted VCCs and the total number of vulnerabilities—i.e., 70 for the V-SZZ dataset and 100 for our dataset—for each technique. While such an indicator did not actively contribute to determining the technique with the best predictive capabilities, it indicates the average number of VCCs predicted for each vulnerability, i.e., an additional piece of information that helps contextualize the performance according to the traditional metrics. We called this metric *volume* $Vol(t_i)$.

Used Hardware. 2x 36 CPU (2x Xeon Platinum 8352V) with 512 GB RAM with 4 NVidia Tesla A100.

5 Empirical Comparison Results (RQ$_1$)

The results of our benchmark study are reported below. We discuss the results of each research question in separate subsections.

5.1 Individual Techniques (RQ$_{1.1}$)

Table 3 shows the performance metrics computed after running the 12 VCC mining techniques on the two benchmark datasets. We immediately observed that L-SZZ [19] achieved the highest precision in the V-SZZ dataset, i.e., 0.60, followed by RS-SZZ [1] that reached 0.45. Apparently, the larger commits are highly likely to contribute to vulnerabilities. V-SZZ—the technique developed along with this dataset—achieved a noticeably high recall, i.e., 0.83, thus resulting in the highest F1-score (0.55). VCC-SZZ [36] also achieved the same F1 score, though with slightly higher precision at the cost of recall. This indicates that the two techniques behave similarly but balance the precision and recall slightly differently. From a broader perspective, even the precision of the best technique (L-SZZ) was not astonishing: almost half of the flagged VCCs do not really contribute to any vulnerability. Conversely, the recall score of V-SZZ can be considered sufficiently high, as a real VCC is missed in less than one out of five cases. Still, the technique that achieved the best recall of 0.89 was TC-SZZ. This finding is not surprising since TC-SZZ was designed to hit ghost commits, resulting in a higher recall than other techniques.

The tendency was different in our dataset, where the best precision, i.e., 0.34, was achieved by RS-SZZ, though with a noticeably lower value than the top precision in the V-SZZ dataset. This also allowed RS-SZZ to have the highest F1 score, 0.33, lower than the top F1 score in the V-SZZ dataset. Besides, we observe that RS-SZZ had a volume of 0.90, meaning that it failed to find any VCC in ten out of the 100 vulnerabilities due to apparent errors in its implementation. Here, L-SZZ—the most precise technique for the V-SZZ dataset—immediately follows RS-SZZ and is also tied with R-SZZ on 0.26. All the other techniques did not manage to reach a 0.20 precision score. The highest recall was achieved by VF-SZZ [60] with 0.73, though it scored an almost null precision (0.06).

We generally observe that all techniques achieved noticeably lower performance in our dataset for all the metrics. We reflected on the reasons that could have determined such results, and we looked back at the characteristics of the V-SZZ dataset (Table 2) and at its design decisions, which we elaborated in Section 4. The main reason could be that our dataset did not filter out any fixing commits, also retaining those that either deleted more than five lines or deleted no lines—which V-SZZ dataset discarded instead. The presence of such fixing commits might have hindered the retrieval performance of the experimented techniques. As a matter of fact, all the techniques strongly rely on the `git blame` mechanism, which is influenced by the amount of deleted lines in the fixing commits. Hence, fixing commits that deleted many lines increases the chance of returning many commits, overestimating the set of real VCCs, resulting in a noticeable drop in their precision. As an example, CVE-2015-5175 affecting project Apache CXF Fediz was fixed in commit `f65c961e`[9], which deleted more than 600 lines, mainly because a file with more than 400 lines was deleted. This led B-SZZ to retrieve 12 VCCs, and its direct improved versions, AG-SZZ, MA-SZZ, and RA-SZZ, retrieved ten. In fact, only one was the correct VCC they were supposed to retrieve. At the same time, most mining techniques have no mechanism for handling fixing commits that deleted no lines, except for VF-SZZ [60], specifically designed to retrieve

[9] https://github.com/apache/cxf-fediz/commit/f65c961ea31e3c1851daba8e7e49fc37bbf77b19

Table 3. Performance of the 12 individual commit mining techniques executed on the V-SZZ and our datasets. The best performance of each dataset in precision (*Pr*), recall (*Re*), and F1-score (*F*1) are depicted in **boldface**.

	V-SZZ Dataset				Our Dataset			
Technique	Vol	Pr	Re	$F1$	Vol	Pr	Re	$F1$
AG-SZZ	3.02	0.43	0.53	0.47	1.43	0.12	0.17	0.14
B-SZZ	3.77	0.38	0.64	0.48	1.77	0.16	0.29	0.21
DJ-SZZ	13.80	0.27	0.50	0.35	19.37	0.03	0.61	0.06
L-SZZ	0.70	**0.60**	0.47	0.53	0.38	0.26	0.10	0.14
MA-SZZ	3.16	0.40	0.50	0.44	1.49	0.11	0.16	0.13
R-SZZ	0.70	0.44	0.34	0.38	0.38	0.26	0.10	0.14
RA-SZZ	2.93	0.42	0.47	0.44	1.34	0.09	0.12	0.10
RS-SZZ	1.00	0.45	0.26	0.33	0.90	**0.34**	0.31	**0.33**
TC-SZZ	3.30	0.29	**0.89**	0.44	11.11	0.05	0.60	0.10
V-SZZ	4.48	0.41	0.83	**0.55**	2.18	0.14	0.30	0.19
VCC-SZZ	3.81	0.43	0.77	**0.55**	2.02	0.16	0.32	0.21
VF-SZZ	11.39	0.21	0.39	0.27	12.18	0.06	**0.73**	0.11

Vol=Volume, i.e., average number of VCCs predicted for each vulnerability.

as many VCCs as possible, even at the cost of returning many false positives, which explains its good recall on our dataset.

On the other side of the spectrum, VF-SZZ and DJ-SZZ [85] were the least precise techniques in both datasets, likely due to their tendency to return many VCCs, as indicated by their large Vol values. Nevertheless, V-SZZ, in its own dataset, managed to have a "high" relative precision despite retrieving at least four VCCs per vulnerability on average. To summarize, we could observe that the more commits returned, the higher the chances of making mistakes. Hence, it would be preferable for the technique to be "careful" in deeming a commit as VCC if the goal is to have a highly precise dataset of VCCs. Nevertheless, we cannot consider such performances satisfactory, highlighting the need to envision novel VCC mining techniques to improve their precision. However, should the goal be to have a set of commits that are meant to be further inspected manually, then the recall metric should be sufficiently high to indicate V-SZZ as the best choice for this task.

Lastly, we observed that all the algorithms specifically designed to collect VCC (i.e., VF-SZZ, VCC-SZZ, RS-SZZ, and V-SZZ) do not perform significantly better than the approaches designed for bug-inducing commits, though they still exhibit interesting results. In general, B-SZZ [93] is the top-3 best technique in terms of F1-score in both datasets. This finding aligns with other empirical studies that recommend using B-SZZ in many circumstances [18, 69]. Still, V-SZZ looks the best choice for the V-SZZ dataset (tied with VCC-SZZ), while RS-SZZ is preferred in our dataset. Both techniques were specifically designed for vulnerabilities, so they seem to achieve their intended goal of finding VCCs.

RQ$_{1.1}$ Summary. Existing mining techniques for VCCs are generally imprecise. The most precise technique in the V-SZZ dataset was L-SZZ, reaching 0.60; while in our datasets, RS-SZZ achieved 0.34. Any technique that returns a few commits has a higher chance of having higher precision. Regarding the recall, the results were more convincing: V-SZZ reached 0.83 in its own datasets, while VF-SZZ achieved 0.73 in our datasets—both techniques were designed for mining VCC. According to the F1 score, the best techniques were V-SZZ (or VCC-SZZ) for the V-SZZ dataset and RS-SZZ for our dataset. Nevertheless, the techniques designed for VCCs did not achieve significantly better results than those designed for bug-inducing commits.

5.2 Combined Approaches (RQ$_{1.2}$)

Tables 4 and 5 show the performance of the combined approaches on the V-SZZ and our dataset, respectively. For the sake of readability, the tables only report the top 10 N-wise combinations (with $N = 2, 3, 4, 5$) performing the best in precision. The full results can be found in our online appendix [33].

In the V-SZZ dataset, the combination of two techniques had a slight positive effect on the precision. Indeed, the pair ⟨L-SZZ+RS-SZZ⟩ achieved 0.68, which is a 13% improvement over the most precise individual technique (L-SZZ, which achieved 0.60) on the same dataset. In particular, we observe that L-SZZ is always involved in the most precise combinations, indicating that it is helpful in filtering many false positives. This further supports the fact that the real commits contributing to vulnerabilities are those that changed many lines, having higher chances of being retrieved by `blame`-based techniques. The same pair ⟨L-SZZ+RS-SZZ⟩ also suffered a slight drop in F1 score with respect to the sole L-SZZ, from 0.55 to 0.49. This happened due to the noticeably low recall of 0.39 due to the nature of conservative techniques like L-SZZ, having higher chances of filtering out real VCCs, negatively affecting the recall.

The pair with the highest F1 score was ⟨VCC-SZZ+V-SZZ⟩, with 0.60 (not visible in Table 4). These two techniques also achieved the best F1 score individually, both achieving 0.55. This happened at the cost of a slight recall drop, from 0.83 (by V-SZZ) to 0.74, the highest recall among all the pairs. It is worth noting that a drop in the recall is inevitable when combining techniques in this way, as the intersection cannot determine sets with additional VCCs from the individual sets. Hence, recall is always expected to decrease, though with variable effects.

It is possible to push the precision even further by employing higher order combinations ($N = 3, 4, 5$) until reaching 0.75 with ⟨L-SZZ+RS-SZZ+V-SZZ+RA-SZZ⟩, which is the intersection of the output sets of the most precise pair ⟨L-SZZ+RS-SZZ⟩ with V-SZZ and RA-SZZ. As expected, this comes at the cost of a reduction in recall, further dropping to 0.34. Besides, the more techniques are combined, the higher the impact on the recall and the F1 score as well. Thus, a maximum of two techniques should be combined to have a maximum F1 score (0.60).

The special technique made by the majority-voting schema ("Voting Combination" in Table 4) stands somewhere in the middle. Its recall (0.60) is among the top-5 recall scores among all combinations (pairs and triples), while its precision (0.51) is among the top-20 precision scores. This resulted in a 0.55 F1 score, which is lower than the highest one achieved by ⟨VCC-SZZ+V-SZZ⟩. Nevertheless, the majority voting schema appears to have made the smoothest balance between precision and recall: no other individual or combined techniques minimized the difference between the two values to this extent while keeping the two values at their possible maximum. With a further closer inspection, we found that this kind of voting scheme favors techniques that tend to make overestimation, like DJ-SZZ and VF-SZZ, as they tend to provide more votes than other, more careful techniques, like R-SZZ or L-SZZ. Consequently, the final result set is more likely to contain the VCCs voted by imprecise techniques, motivating the lower precision score.

Table 4. Performance of the combined VCC mining techniques on the **V-SZZ dataset**, grouped by the number of participating techniques. The combinations are sorted by precision (*Pr*) in each subgroup, and only the top 10 most precise are shown.

MA-SZZ	R-SZZ	V-SZZ	RA-SZZ	AG-SZZ	VCC-SZZ	TC-SZZ	RS-SZZ	DJ-SZZ	L-SZZ	B-SZZ	VF-SZZ				
Top-10 Pairwise Combinations												*Vol*	*Pr*	*Re*	*F1*
							✗		✗			0.40	0.68	0.39	0.49
					✗				✗			0.50	0.66	0.44	0.53
		✗							✗			0.59	0.65	0.44	0.53
						✗			✗			0.68	0.64	0.43	0.51
				✗					✗			0.65	0.63	0.47	0.54
								✗	✗			0.56	0.63	0.46	0.53
			✗						✗			0.63	0.62	0.44	0.52
									✗		✗	0.63	0.61	0.44	0.51
	✗								✗			0.33	0.61	0.33	0.43
		✗					✗					0.83	0.60	0.26	0.36
Top-10 Triple-wise Combinations															
		✗					✗		✗			0.36	0.72	0.37	0.49
						✗	✗		✗			0.39	0.71	0.36	0.48
					✗		✗		✗			0.31	0.70	0.37	0.49
			✗				✗		✗			0.36	0.69	0.36	0.47
				✗			✗		✗			0.37	0.69	0.39	0.50
			✗		✗				✗			0.45	0.69	0.41	0.52
							✗		✗	✗		0.32	0.68	0.30	0.42
							✗	✗	✗			0.40	0.68	0.39	0.49
✗							✗		✗			0.40	0.68	0.39	0.49
		✗	✗						✗			0.54	0.67	0.41	0.51
Top-10 Quadruple-wise Combinations															
		✗	✗				✗		✗			0.33	0.75	0.34	0.47
			✗			✗	✗		✗			0.35	0.74	0.33	0.46
			✗		✗		✗		✗			0.28	0.73	0.34	0.47
	✗	✗					✗		✗			0.21	0.72	0.30	0.42
		✗					✗	✗	✗			0.36	0.72	0.37	0.49
✗		✗					✗		✗			0.36	0.72	0.37	0.49
		✗		✗			✗		✗			0.35	0.72	0.37	0.49
		✗				✗	✗		✗			0.36	0.71	0.36	0.48
						✗	✗		✗		✗	0.37	0.71	0.34	0.46
					✗	✗	✗		✗			0.30	0.71	0.34	0.46
Top-10 Quintuple-wise Combinations															
		✗	✗				✗	✗	✗			0.33	0.75	0.34	0.47
✗		✗	✗				✗		✗			0.33	0.75	0.34	0.47
		✗	✗	✗			✗		✗			0.33	0.75	0.34	0.47
		✗	✗			✗	✗		✗			0.33	0.74	0.33	0.46
			✗			✗	✗	✗	✗			0.35	0.74	0.33	0.46
✗			✗			✗	✗		✗			0.35	0.74	0.33	0.46
		✗	✗		✗		✗		✗			0.27	0.74	0.33	0.46
		✗	✗				✗		✗		✗	0.31	0.74	0.33	0.46
			✗	✗		✗	✗		✗			0.34	0.74	0.33	0.46
✗			✗		✗		✗		✗			0.28	0.73	0.34	0.47
Voting Combination															
✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	2.32	0.53	0.60	0.56

Just like the individual technique, the results observed in our dataset are somewhat different. The pairwise combinations did not achieve more than 0.58 precision, i.e., 0.10 lower than the scores in the V-SZZ dataset. However, we found that the best pair was still ⟨L-SZZ+RS-SZZ⟩), the same as the V-SZZ dataset. Here, too, the most precise pair includes the two most precise techniques when launched in isolation. The higher order combinations, particularly from $N = 4$, jumped to 0.75 precision when ⟨R-SZZ+RS-SZZ+L-SZZ+VCC-SZZ⟩ are involved, on par with the most precise higher order combinations in the V-SZZ dataset. Such results mark the best option for retrieving a highly precise set of VCCs. Nevertheless, this comes at a vast reduction in recall, dropping to 0.06, which inevitably affected the F1 score as well. In general, the most precise combinations in our dataset have terribly lower recall scores than in the V-SZZ dataset. We

Table 5. Performance of the combined VCC mining techniques on **our dataset**, grouped by the number of participating techniques. The combinations are sorted by precision (*Pr*) in each subgroup, and only the top 10 most precise are shown.

MA-SZZ	R-SZZ	V-SZZ	RA-SZZ	AG-SZZ	VCC-SZZ	TC-SZZ	RS-SZZ	DJ-SZZ	L-SZZ	B-SZZ	VF-SZZ	*Vol*	*Pr*	*Re*	*F1*
Top-10 Pairwise Combinations															
	×						×					0.38	0.58	0.07	0.12
							×		×			0.40	0.58	0.07	0.12
	×								×			0.33	0.50	0.09	0.15
×							×					1.37	0.50	0.09	0.15
				×			×					1.34	0.50	0.10	0.17
					×		×					1.92	0.49	0.19	0.27
							×			×		1.75	0.47	0.17	0.25
			×				×					1.29	0.44	0.07	0.12
						×	×					0.96	0.40	0.29	0.34
		×					×					0.83	0.38	0.30	0.34
Top-10 Triple-wise Combinations															
					×		×		×			0.31	0.70	0.07	0.13
	×						×		×			0.22	0.67	0.06	0.11
							×		×		×	0.38	0.67	0.06	0.11
	×				×				×			0.26	0.64	0.07	0.13
	×				×		×					0.31	0.64	0.07	0.13
	×						×				×	0.37	0.60	0.06	0.11
		×					×		×			0.36	0.58	0.07	0.12
×	×						×					0.38	0.58	0.07	0.12
×							×		×			0.40	0.58	0.07	0.12
	×	×					×					0.36	0.58	0.07	0.12
Top-10 Quadruple-wise Combinations															
	×				×		×		×			0.18	0.75	0.06	0.11
	×						×		×		×	0.21	0.71	0.05	0.09
					×		×	×	×			0.31	0.70	0.07	0.13
					×		×		×	×		0.27	0.70	0.07	0.13
				×	×		×		×			0.31	0.70	0.07	0.13
					×	×	×		×			0.30	0.70	0.07	0.13
		×			×		×		×			0.30	0.70	0.07	0.13
	×				×				×	×		0.24	0.70	0.07	0.13
×					×		×		×			0.31	0.70	0.07	0.13
×	×						×		×			0.22	0.67	0.06	0.11
Top-10 Quintuple-wise Combinations															
	×				×		×	×	×			0.18	0.75	0.06	0.11
	×	×			×		×		×			0.18	0.75	0.06	0.11
	×			×	×		×		×			0.18	0.75	0.06	0.11
×	×				×		×		×			0.18	0.75	0.06	0.11
	×				×		×		×	×		0.16	0.75	0.06	0.11
	×				×	×	×		×			0.18	0.75	0.06	0.11
	×						×	×	×		×	0.21	0.71	0.05	0.09
	×	×					×		×		×	0.20	0.71	0.05	0.09
	×					×	×		×		×	0.21	0.71	0.05	0.09
	×		×		×		×		×			0.16	0.71	0.05	0.09
Voting Combination															
×	×	×	×	×	×	×	×	×	×	×	×	2.32	0.27	0.45	0.34

also observed the predominant role that VCC-SZZ [36] had with respect to the V-SZZ dataset. Indeed, when added to ⟨L-SZZ+RS-SZZ⟩ it causes an increment in precision from 0.58 to 0.70 without affecting the recall. The majority-voting schema ("Voting Combination" in Table 5) did not perform as well as in the V-SZZ dataset. The precision (0.26) is much lower than that of any other combination, though still higher than that of most of the individual techniques. On the contrary, the recall (0.43) is better than any other combination, despite being noticeably lower than VF-SZZ alone.

In summary, we observed that even imprecise techniques can be reconsidered if joined with other less precise techniques. The price in terms of recall is high, inevitably affecting the F1 score on cascade. Therefore, the AND

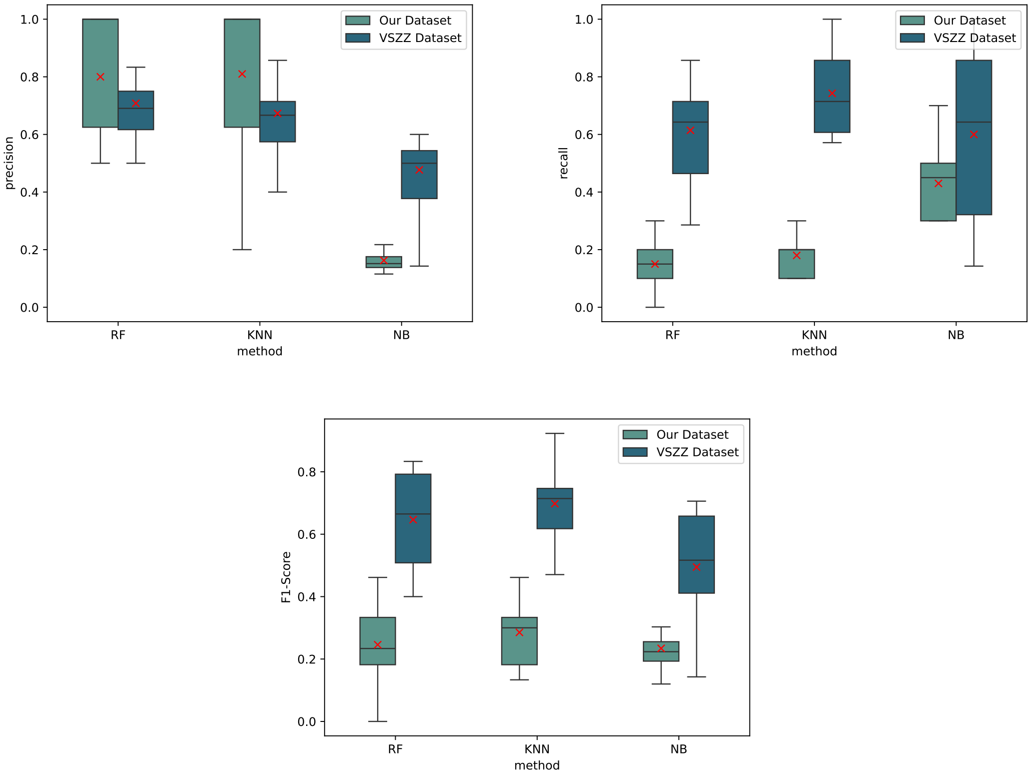


Fig. 1. Performance metrics of the machine learning classifiers scored on both datasets with the 10-fold cross-validation. The red crosses indicate the mean, while the horizontal lines represent the median.

combinations could help retrieve a minimal set of commits that are likely to be VCCs; this can be exploited to localize the origin of a vulnerability quickly and minimize human activity to ensure the commit contributes to the vulnerability.

Regarding the last special technique, i.e., the ML-driven approach leveraging the individual techniques' output, the average performance resulting from the 10-fold cross-validation is reported in Figure 1. In our dataset, the K-Nearest Neighbor (KNN) and Random Forest (RF) classifiers were not only the most precise ML-based approaches, but also the best approaches overall, reaching the average precision of 0.80 and 0.81, which corresponds to a ∼ 142% improvement over RS-SZZ alone. We also note that KNN and RF worked somewhat similarly on our dataset with respect to all of the reported metrics, with KNN having slightly less variance in terms of F1 score. Naïve Bayes (NB) classifier scored very low precision with 0.16, comparable to B-SZZ and other individual techniques; however, it also achieved the highest average recall (0.43) among the ML-based approaches, though still way below VF-SZZ or DJ-SZZ. For the V-SZZ dataset, we can observe a slightly lower precision for the RF (0.70) and KNN (0.67), while a sharp improvement for NB (0.48). Even in this case, RF and KNN were more precise than the best individual technique, L-SZZ, with a ∼ 12% improvement, but without outperforming the other combined approaches. For the case of recall, the trend is inverted; while NB had only a small growth, RF (0.64) and KNN (0.74) had a considerable increase. Nevertheless, these results are still behind the remarkable recall of TC-SZZ of 0.89. However, thanks to this better balance of precision and recall, the resulting F1 scores were higher than those seen in our dataset, with KNN having the highest value, outperforming any other individual or combiner technique. In summary, RF and KNN have generally good precision; the highest in our dataset, among the best in the V-SZZ dataset. Their recall is no better than other good individual techniques; nevertheless, KNN also had the best F1 score over all individual techniques and other combinations in the V-SZZ dataset.

Lastly, we verified whether a certain classifier was significantly better than another on a given performance metric through the use of paired Wilcoxon signed-rank tests [84]. The results support the fact that in our dataset, KNN and RF significantly outperformed NB in terms of precision, while NB significantly outperformed KNN and RF in terms of recall. The same did not happen for the F1: All three models behaved similarly. In the V-SZZ dataset, however, KNN and RF outperformed NB in both precision and F1 score, while NB did not outperform KNN and RF in recall (the null hypothesis could not be rejected). In summary, the statistical tests confirmed the conclusions we drew in this section.

RQ$_{1.2}$ Summary. Individual VCC mining techniques can be combined to boost their precision slightly. L-SZZ is the most suitable technique, always appearing in the most precise combinations evaluated in both datasets, reaching 0.75 precision with the help of RS-SZZ in both datasets. This does not come for free, as combining techniques leads to an inevitable large drop in the recall, particularly in our dataset. The special technique involving all techniques in a majority voting scheme achieved the smoothest balance between precision and recall in the V-SZZ dataset, but not in our dataset. The Random Forest and K-Nearest Neighbors are valid ways to aggregate technique outputs depending on the specific dataset: To maximize the precision on our dataset and to maximize the F1 score on the V-SZZ dataset.

6 Fixing Action Analysis (RQ$_2$)

6.1 Analysis Method

After evaluating the quantitative performance of the VCC mining techniques, we observed that precision and recall could only provide a shallow view of the circumstances that benefited or hindered each technique's performance. Therefore, we performed a comprehensive analysis of the cases in which the 12 techniques successfully returned all the expected VCC from the ground truth. This could also allow us to reflect on the cases in which a technique returned an incorrect VCC (i.e., a commit that was not part of the ground truth). We were particularly interested in the characteristics of the input that each technique processes; since they all accept the vulnerability fix commits, we looked at the characteristics of the fixing commits.

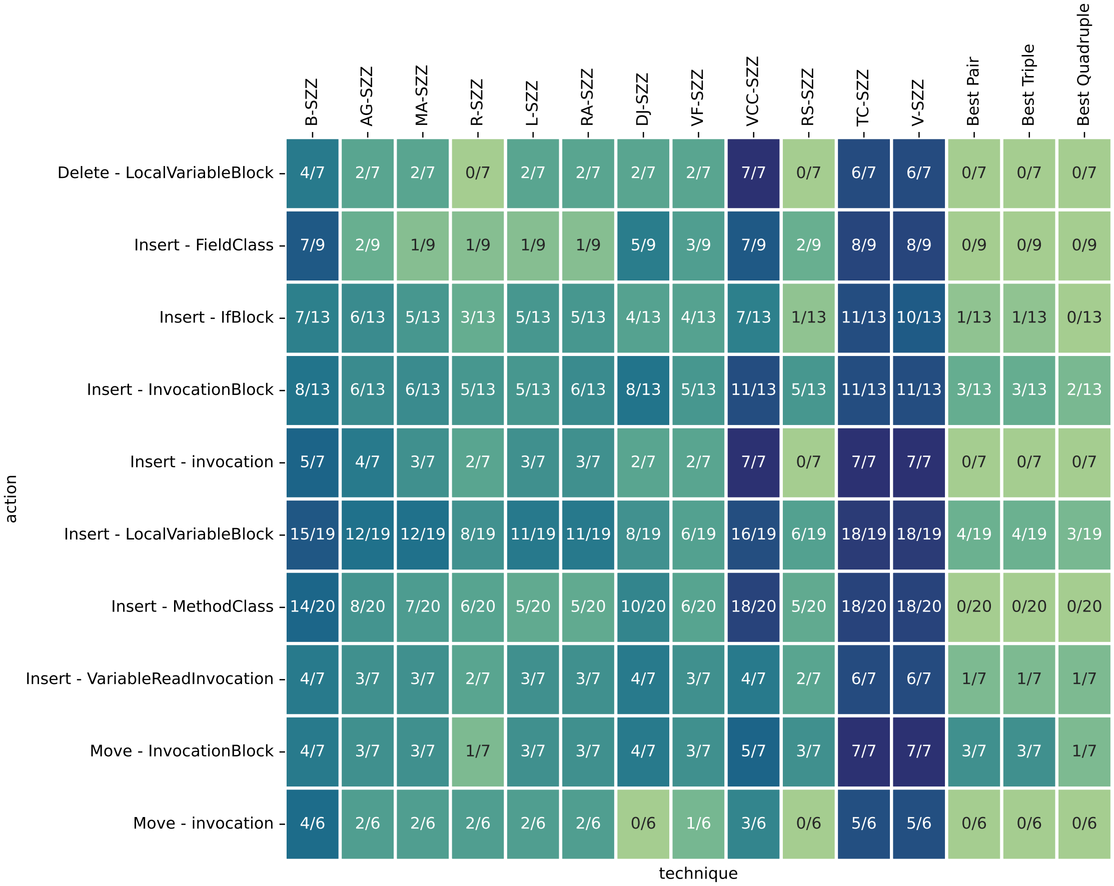
We analyzed the input fixing commits from two perspectives to provide a comprehensive view. On the one hand, we looked at the changes made to the source code from a **syntactic perspective**, extracting the kind of changes that occurred at the syntax tree level to patch the vulnerability—e.g., a class method was added or removed. We refer to these changes as *syntactic fixing actions*, which we extracted by following a similar approach defined by Canfora et al. [11] to identify the fixing actions that occurred in a vulnerability fix commit at the syntax-tree level. Specifically, we extracted the changes using Coming [48], a tool able to identify the differences between the Abstract Syntax Trees (ASTs) of the vulnerable (i.e., the version before the fix) and patched versions of each file involved in the fixing by utilizing the AST-based diff algorithm GumTree [20]. This resulted in a list of fixing actions that correspond to transformations developers performed to achieve the changes made in the commit. Each action can be of four different *types* applied to nodes in the ASTs: 'Insert', 'Delete', 'Move', 'Update'.

Table 6. Top 10 syntactic fix actions extracted from the V-SZZ and our dataset by Coming, sorted by number of occurrences in the fix commits contained in them.

V-SZZ Dataset (V=70)		Our Dataset (V=100)	
Syntactic Fix Action	Nr.	Syntactic Fix Action	Nr.
Insert - MethodClass	20	Insert - MethodClass	45
Insert - LocalVariableBlock	19	Insert - InvocationBlock	42
Insert - IfBlock	13	Insert - IfBlock	40
Insert - InvocationBlock	13	Insert - FieldClass	35
Insert - FieldClass	9	Insert - LocalVariableBlock	34
Move - InvocationBlock	7	Move - VariableReadInvocation	25
Insert - Invocation	7	Move - InvocationBlock	24
Delete - LocalVariableBlock	7	Insert - AssignmentBlock	22
Insert - VariableReadInvocation	7	Delete - LocalVariableBlock	19
Move - Invocation	6	Delete - IfBlock	16
Others (N=115)	287	**Others**	833
Total (V=70)	314	**Total**	1,135

On the other hand, we looked at the conceptual changes made to the source code from a **semantic perspective**, relying on a manual inspection process, for example, a filter on the input was strengthened. We refer to these recurring changes as *semantic fixing actions*. Two authors of this paper led this inspection. Specifically, they reviewed all vulnerabilities (of both datasets) independently, assigning a *textual label* describing the main change made in the fixing commit. The two resulting sets of textual labels were equalized and merged into one single *normalized catalog* of semantic fixing actions. Then, the two inspectors did another round of inspection of all fixing commits to remap the initial label into the new ones defined in the normalized catalog. After this second pass, no new labels were defined. They reached an agreement of 97% and a very strong reliability of 0.96—measured with Cohen's Kappa [14]. After a joint review of the disagreement cases, the reviewers fully agreed on the labels. As the variations predominantly stemmed from minor labeling discrepancies, e.g., the first inspector categorized a fixing commit as "Implement Filter on External Input", while the second one as "Strengthen Filter on External Input".

Just like the performance analyses made in **RQ**$_{1.1}$ and **RQ**$_{1.2}$, both analyzes were performed on the two datasets. At this point, we analyzed how the mining techniques behaved with respect to the retrieved syntactic and semantic fixing actions. We counted the cases in which a technique predicted the **whole expected set of VCCs** correctly and then split the counting based on the actions occurring in the commits fixing a vulnerability. It is important to note that for this analysis, we look at the capabilities of the techniques to predict the right VCCs without considering the downsides of the presence of VCCs not in the expected set. This approach has an inevitable preference toward techniques that scored higher recall (see Table 3), since they are not penalized for the presence of noisy VCCs. We opted to count these cases rather than "full match" (i.e., when the predicted set of VCCs is exactly equal to the expected set) due to the limited number of cases in which the latter occurred. In these analyses, we also include the best (according to precision) pairs, triple, and quadruple results from **RQ**$_{1.2}$.



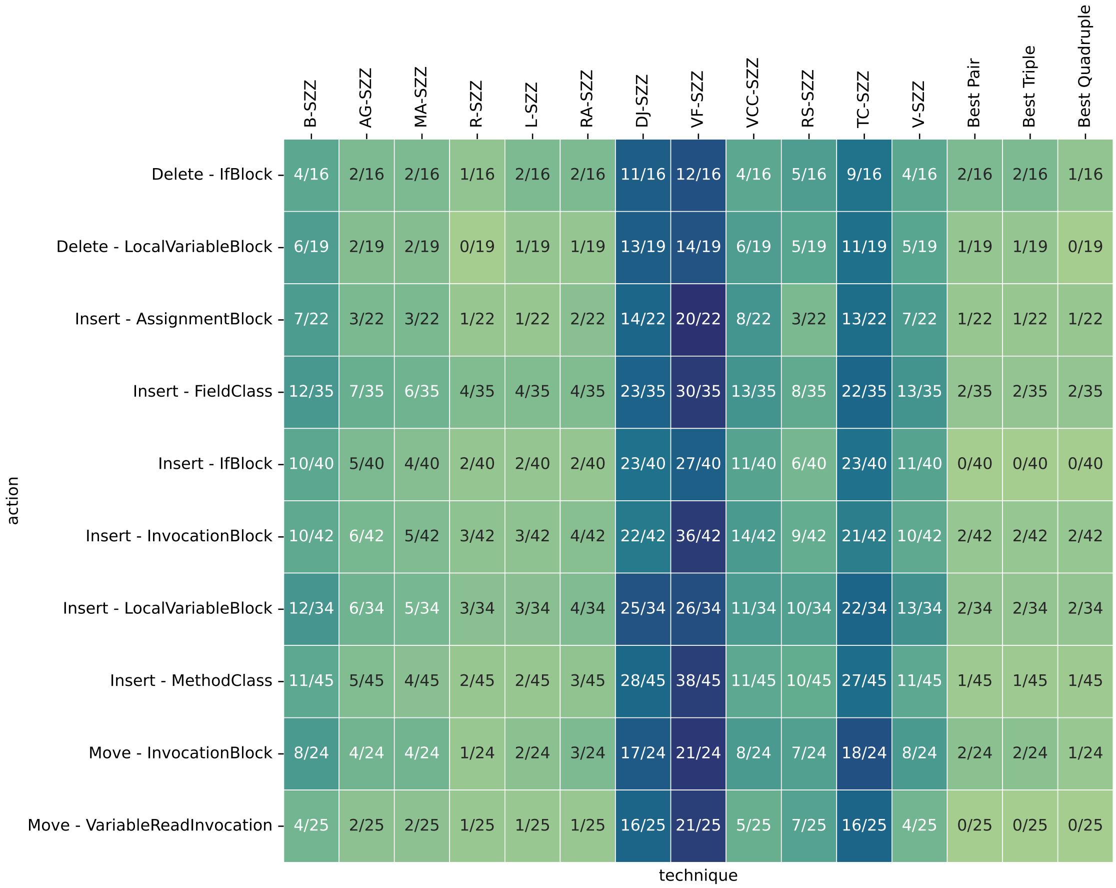
action	B-SZZ	AG-SZZ	MA-SZZ	R-SZZ	L-SZZ	RA-SZZ	DJ-SZZ	VF-SZZ	VCC-SZZ	RS-SZZ	TC-SZZ	V-SZZ	Best Pair	Best Triple	Best Quadruple
Delete - LocalVariableBlock	4/7	2/7	2/7	0/7	2/7	2/7	2/7	2/7	7/7	0/7	6/7	6/7	0/7	0/7	0/7
Insert - FieldClass	7/9	2/9	1/9	1/9	1/9	1/9	5/9	3/9	7/9	2/9	8/9	8/9	0/9	0/9	0/9
Insert - IfBlock	7/13	6/13	5/13	3/13	5/13	5/13	4/13	4/13	7/13	1/13	11/13	10/13	1/13	1/13	0/13
Insert - InvocationBlock	8/13	6/13	6/13	5/13	5/13	6/13	8/13	5/13	11/13	5/13	11/13	11/13	3/13	3/13	2/13
Insert - invocation	5/7	4/7	3/7	2/7	3/7	3/7	2/7	2/7	7/7	0/7	7/7	7/7	0/7	0/7	0/7
Insert - LocalVariableBlock	15/19	12/19	12/19	8/19	11/19	11/19	8/19	6/19	16/19	6/19	18/19	18/19	4/19	4/19	3/19
Insert - MethodClass	14/20	8/20	7/20	6/20	5/20	5/20	10/20	6/20	18/20	5/20	18/20	18/20	0/20	0/20	0/20
Insert - VariableReadInvocation	4/7	3/7	3/7	2/7	3/7	3/7	4/7	3/7	4/7	2/7	6/7	6/7	1/7	1/7	1/7
Move - InvocationBlock	4/7	3/7	3/7	1/7	3/7	3/7	4/7	3/7	5/7	3/7	7/7	7/7	3/7	3/7	1/7
Move - invocation	4/6	2/6	2/6	2/6	2/6	2/6	0/6	1/6	3/6	0/6	5/6	5/6	0/6	0/6	0/6

Fig. 2. Correct predictions divided per syntactic fixing action on the V-SZZ dataset.

6.2 Syntactic Fixing Action Analysis

Coming extracted a total of 292 fixing actions. We sorted by occurrences per dataset and reported the top 10 in Table 6, as most of the actions only occurred a few times, resulting in being of less relevance. We observed that for both datasets, the most prevalent action in fixing a vulnerability is the **insertion of new code elements**, mainly methods, invocations, and `if`-statements. Such a result was expected and in line with what has been observed in other empirical studies [11]. In fact, fixing a vulnerability often requires the creation of new filters and procedures to sanitize inputs or extra logic to handle corner cases. Figures 2 and 3 show the correct predictions made on the V-SZZ and our dataset, respectively, divided by the top 10 most frequent syntactic fixing actions.

Figure 2 shows that TC-SZZ and V-SZZ had the best overall performance, seeming to be the recommended technique in essentially all the cases—particularly those concerning moving code elements. They are followed by VCC-SZZ and B-SZZ. From a broader perspective, most techniques struggled when the fix commit deleted an `if`-statement or a local variable, but also when assignments are introduced. We found no clear trend relating the syntactic fixing actions to the technique performance, though we observed that the most recommended are those that achieved high



action	B-SZZ	AG-SZZ	MA-SZZ	R-SZZ	L-SZZ	RA-SZZ	DJ-SZZ	VF-SZZ	VCC-SZZ	RS-SZZ	TC-SZZ	V-SZZ	Best Pair	Best Triple	Best Quadruple
Delete - IfBlock	4/16	2/16	2/16	1/16	2/16	2/16	11/16	12/16	4/16	5/16	9/16	4/16	2/16	2/16	1/16
Delete - LocalVariableBlock	6/19	2/19	2/19	0/19	1/19	1/19	13/19	14/19	6/19	5/19	11/19	5/19	1/19	1/19	0/19
Insert - AssignmentBlock	7/22	3/22	3/22	1/22	1/22	2/22	14/22	20/22	8/22	3/22	13/22	7/22	1/22	1/22	1/22
Insert - FieldClass	12/35	7/35	6/35	4/35	4/35	4/35	23/35	30/35	13/35	8/35	22/35	13/35	2/35	2/35	2/35
Insert - IfBlock	10/40	5/40	4/40	2/40	2/40	2/40	23/40	27/40	11/40	6/40	23/40	11/40	0/40	0/40	0/40
Insert - InvocationBlock	10/42	6/42	5/42	3/42	3/42	4/42	22/42	36/42	14/42	9/42	21/42	10/42	2/42	2/42	2/42
Insert - LocalVariableBlock	12/34	6/34	5/34	3/34	3/34	4/34	25/34	26/34	11/34	10/34	22/34	13/34	2/34	2/34	2/34
Insert - MethodClass	11/45	5/45	4/45	2/45	2/45	3/45	28/45	38/45	11/45	10/45	27/45	11/45	1/45	1/45	1/45
Move - InvocationBlock	8/24	4/24	4/24	1/24	2/24	3/24	17/24	21/24	8/24	7/24	18/24	8/24	2/24	2/24	1/24
Move - VariableReadInvocation	4/25	2/25	2/25	1/25	1/25	1/25	16/25	21/25	5/25	7/25	16/25	4/25	0/25	0/25	0/25

Fig. 3. Correct predictions divided per syntactic fixing action on our dataset.

recall.Figure 3 focuses on our dataset, and it shows that DJ-SZZ and VF-SZZ stood out, performing well in essentially all the circumstances. Similarly, the rest of the techniques perform much less and do not seem to specialize in specific fixing actions. Like the V-SZZ dataset, we found no clear trend between the syntactic fixing actions and the technique performance; therefore, the most recommended are still those that achieved the highest overall correct predictions. Inspecting the best combinations, it stands out that in both datasets, the correct predictions dropped significantly. This result was expected as the combinations resulting from the intersection have lower chances of predicting the whole set of expected VCCs.

6.3 Semantic Fixing Action Analysis

Table 7 reports the full catalog of semantic fixing actions the inspectors found, 16 in total, and their occurrences in both datasets. Note that not all fixing actions occurred in both datasets; namely, in the V-SZZ dataset, three actions never occurred, while in our datasets, five actions are missing. We observed that the most frequent fixing actions in both datasets concern external input handling. This includes adding or modifying checks (most of them in the

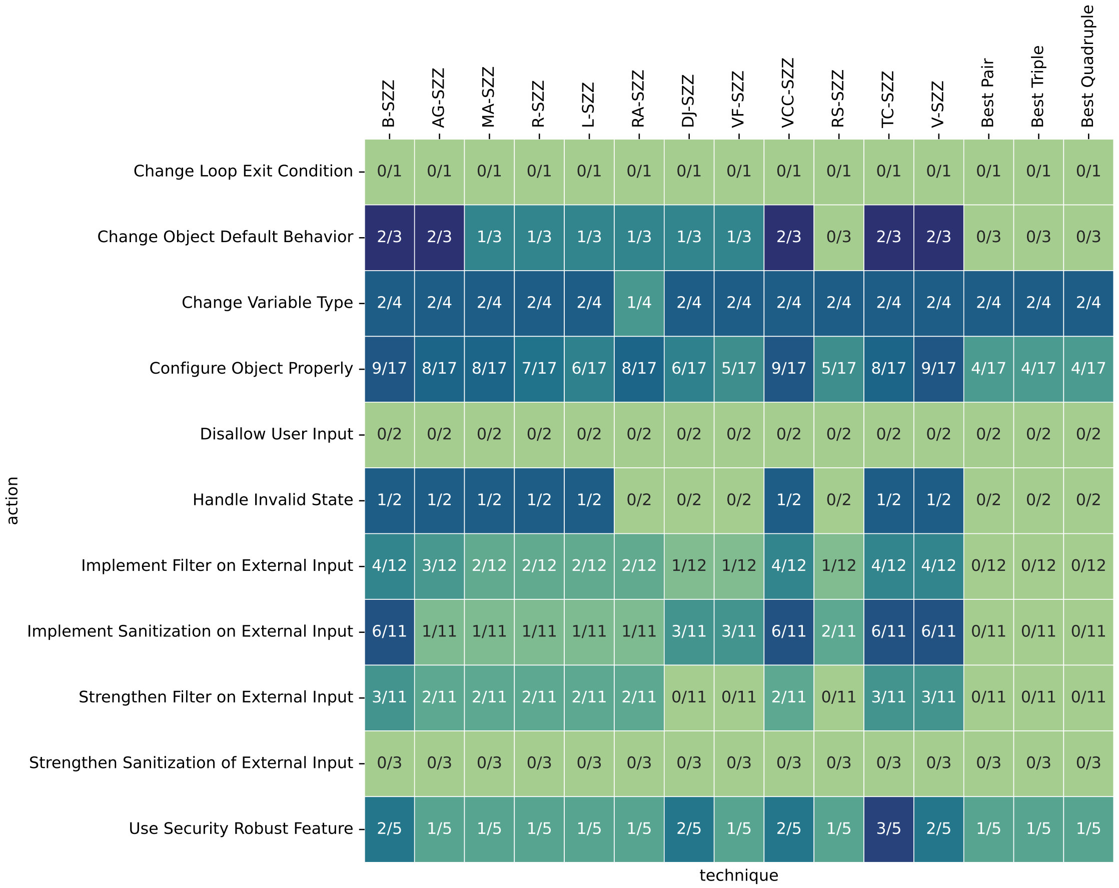
Table 7. Semantic fix actions extracted from the V-SZZ and our dataset by the two inspectors, sorted by number of occurrences in the fix commits contained in them.

V-SZZ Dataset (V=70)		Our Dataset (V=100)	
Semantic Fix Action	Nr.	Semantic Fix Action	Nr.
Configure Object Properly	17	Strengthen Filter on External Input	29
Implement Filter on External Input	12	Implement Filter on External Input	24
Strengthen Filter on External Input	11	Configure Object Properly	15
Implement Sanitization on External Input	11	Implement Sanitization on External Input	9
Use Security Robust Feature	5	Strengthen Sanitization of External Input	6
Change Variable Type	4	Handle Invalid State	4
Strengthen Sanitization of External Input	3	Invalidate Cached Information	3
Change Object Default Behavior	3	Change Object Default Behavior	3
Handle Invalid State	2	Hide Sensitive Information	3
Disallow User Input	2	Change Variable Type	1
Change Loop Exit Condition	1	Close Resource	1
Fix Concurrency Issue	1		
Disallow User Input	1		
Total	73	**Total**	98

form of `if`-statements or similar constructs) to validate or sanitize inputs originating from untrusted sources (such as user-supplied input or the content of external files). Another frequent action encountered was "Configure Object Properly," which consists of adjusting the values used to build an object, such as an XML parser or other objects related to configuring the whole application. The rest of the fixing actions occurred in fewer cases and represented more specific activities that could not be merged with other existing actions. Since the inspectors followed a bottom-up approach, we did not group them with other actions to preserve the inspectors' opinions. Figures 4 and 5 show the correct predictions made on the V-SZZ and our dataset, respectively, divided by semantic fixing action.

Figure 4 shows the behavior of each technique in the V-SZZ dataset. We observe that no technique stands out particularly, even when looking at those that achieved the highest recall, such as V-SZZ or TC-SZZ. The only interesting action is "Configure Object Properly", where the V-SZZ, B-SZZ, and VCC-SZZ made the highest number of correct predictions, though without large differences from other techniques. In our dataset, the story is quite different. Figure 5 shows that VF-SZZ and DJ-SZZ had the best overall performance, according to their high recall achieved in this dataset (Table 3). However, VF-SZZ appeared to struggle with some fixing actions. That is, it could only predict one out of six cases of "Strengthen Sanitization of External Input" cases, whereas all other techniques had a successful prediction rate in three out of six on average. In the same action, DJ-SZZ was correct in four out of six cases, above the average. To a lesser extent, similar behavior could be seen in the "Hide Sensitive Information" action: All techniques except DJ-SZZ were unable to predict the correct VCC consistently. Furthermore, it is noticeable that the original SZZ (B-SZZ) can keep up with the best performing techniques, such as "Configure Object Properly", "Implement Sanitization on External Input", or "Change Object Default Behavior", but drastically falls behind, especially in the other external input related categories like "Strengthen Filter on External Input". This becomes particularly clear when our dataset is taken into account. For the fixing actions, the same trend can be observed, that the best performing combinations significantly drop and in successfully detecting the fixing actions, likely due to the same reason.

Furthermore, we examined the frequency at which a technique failed to select the right VCCs for a vulnerability and related this to the semantic action identified in the fixing commit. This is achieved by first counting the number of VCCs

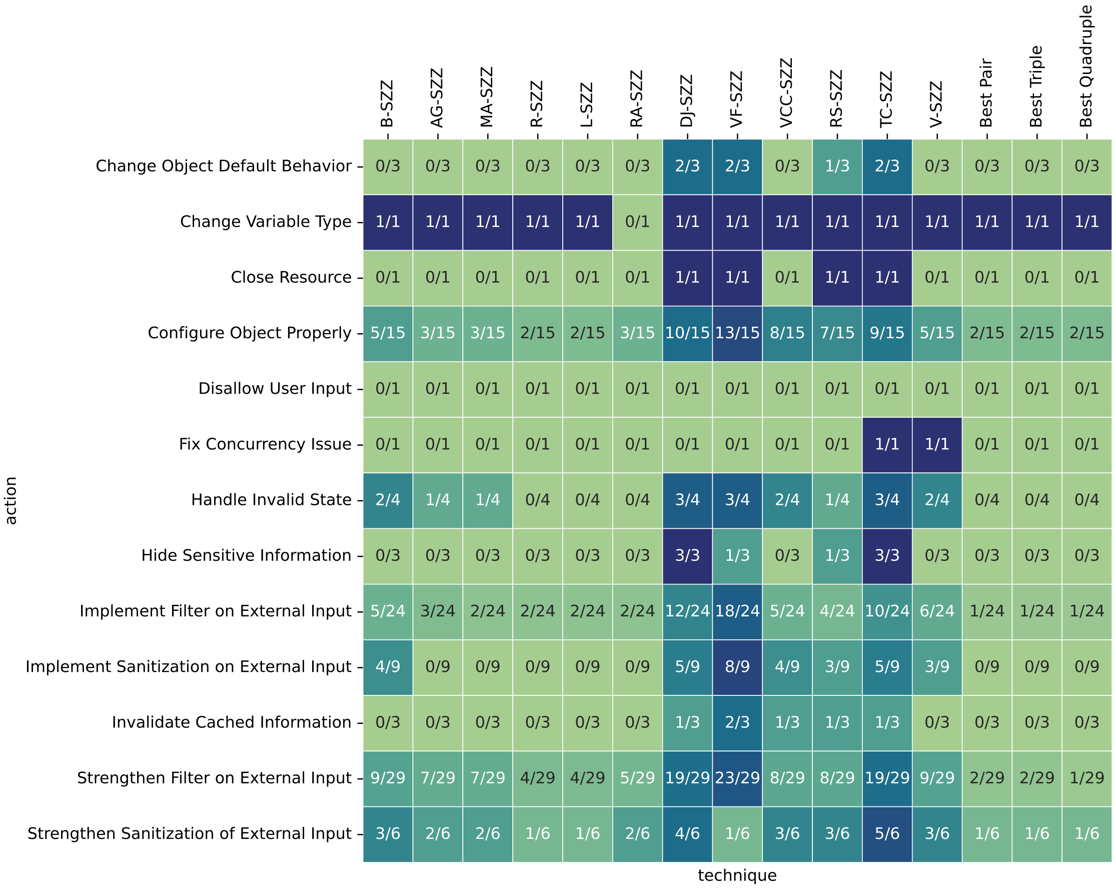


action	B-SZZ	AG-SZZ	MA-SZZ	R-SZZ	L-SZZ	RA-SZZ	DJ-SZZ	VF-SZZ	VCC-SZZ	RS-SZZ	TC-SZZ	V-SZZ	Best Pair	Best Triple	Best Quadruple
Change Loop Exit Condition	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Change Object Default Behavior	2/3	2/3	1/3	1/3	1/3	1/3	1/3	1/3	2/3	0/3	2/3	2/3	0/3	0/3	0/3
Change Variable Type	2/4	2/4	2/4	2/4	2/4	1/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4	2/4
Configure Object Properly	9/17	8/17	8/17	7/17	6/17	8/17	6/17	5/17	9/17	5/17	8/17	9/17	4/17	4/17	4/17
Disallow User Input	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2
Handle Invalid State	1/2	1/2	1/2	1/2	1/2	0/2	0/2	0/2	1/2	0/2	1/2	1/2	0/2	0/2	0/2
Implement Filter on External Input	4/12	3/12	2/12	2/12	2/12	2/12	1/12	1/12	4/12	1/12	4/12	4/12	0/12	0/12	0/12
Implement Sanitization on External Input	6/11	1/11	1/11	1/11	1/11	1/11	3/11	3/11	6/11	2/11	6/11	6/11	0/11	0/11	0/11
Strengthen Filter on External Input	3/11	2/11	2/11	2/11	2/11	2/11	0/11	0/11	2/11	0/11	3/11	3/11	0/11	0/11	0/11
Strengthen Sanitization of External Input	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3	0/3
Use Security Robust Feature	2/5	1/5	1/5	1/5	1/5	1/5	2/5	1/5	2/5	1/5	3/5	2/5	1/5	1/5	1/5

Fig. 4. Correct predictions divided per semantic fixing action on the V-SZZ dataset.

that a technique missed from the oracle set. If that number was greater than one, then we counted one failure for that technique. The results of this analysis are presented in Figures 6 and 7. Considering the failures on the V-SZZ dataset, clusters and patterns can be observed. First, nearly all techniques failed to identify vulnerabilities that were fixed with "Change Loop Exit Condition" action—consequently, also the combined approaches fail to identify the correct VCCs. R-SZZ fails significantly if the fixing commit contains an action related to input handling (sanitation or filtering). On the other hand, techniques like DJ-SZZ and TC-SZZ appeared to be robust when the fixing commit contains the said actions. In fact, they fail only in very few cases. Combining techniques also leads to a drastic increase in failures. This is also reasonable, since the combinations were created by the intersection of the techniques.

In our dataset, the paired technique ⟨RS-SZZ+L-SZZ⟩ recorded the highest number of failures, especially in the prevalent vulnerabilities fixed with actions "Strengthen Filter on External Input" and "Implement Filter on External Input". In fact, this combined technique consistently displayed many failures across all fixing actions. By inspecting the best triple, the failures are reduced by a good margin, becoming comparable to the other individual techniques. This is

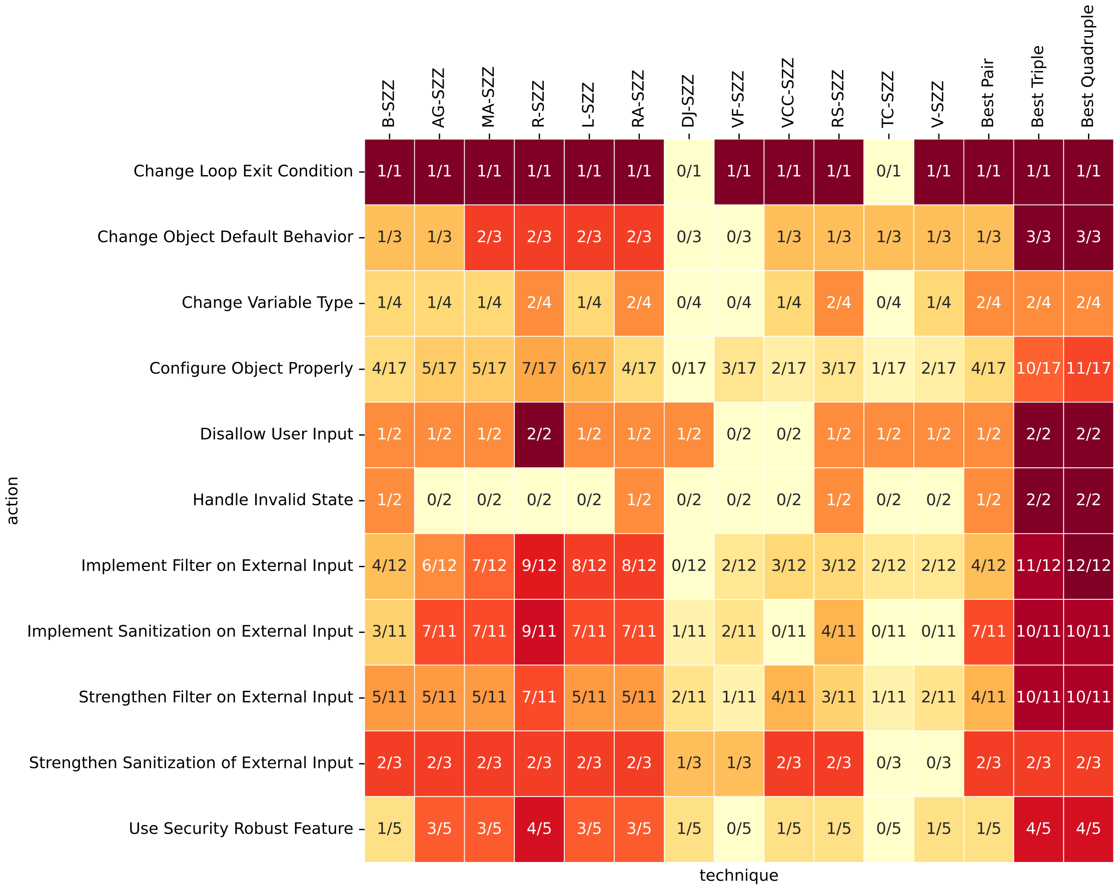


action	B-SZZ	AG-SZZ	MA-SZZ	R-SZZ	L-SZZ	RA-SZZ	DJ-SZZ	VF-SZZ	VCC-SZZ	RS-SZZ	TC-SZZ	V-SZZ	Best Pair	Best Triple	Best Quadruple
Change Object Default Behavior	0/3	0/3	0/3	0/3	0/3	0/3	2/3	2/3	0/3	1/3	2/3	0/3	0/3	0/3	0/3
Change Variable Type	1/1	1/1	1/1	1/1	1/1	0/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
Close Resource	0/1	0/1	0/1	0/1	0/1	0/1	1/1	1/1	0/1	1/1	1/1	0/1	0/1	0/1	0/1
Configure Object Properly	5/15	3/15	3/15	2/15	2/15	3/15	10/15	13/15	8/15	7/15	9/15	5/15	2/15	2/15	2/15
Disallow User Input	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Fix Concurrency Issue	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	1/1	1/1	0/1	0/1	0/1
Handle Invalid State	2/4	1/4	1/4	0/4	0/4	0/4	3/4	3/4	2/4	1/4	3/4	2/4	0/4	0/4	0/4
Hide Sensitive Information	0/3	0/3	0/3	0/3	0/3	0/3	3/3	1/3	0/3	1/3	3/3	0/3	0/3	0/3	0/3
Implement Filter on External Input	5/24	3/24	2/24	2/24	2/24	2/24	12/24	18/24	5/24	4/24	10/24	6/24	1/24	1/24	1/24
Implement Sanitization on External Input	4/9	0/9	0/9	0/9	0/9	0/9	5/9	8/9	4/9	3/9	5/9	3/9	0/9	0/9	0/9
Invalidate Cached Information	0/3	0/3	0/3	0/3	0/3	0/3	1/3	2/3	1/3	1/3	1/3	0/3	0/3	0/3	0/3
Strengthen Filter on External Input	9/29	7/29	7/29	4/29	4/29	5/29	19/29	23/29	8/29	8/29	19/29	9/29	2/29	2/29	1/29
Strengthen Sanitization of External Input	3/6	2/6	2/6	1/6	1/6	2/6	4/6	1/6	3/6	3/6	5/6	3/6	1/6	1/6	1/6

Fig. 5. Correct predictions divided per semantic fixing action on our dataset.

reasonable on the basis of the technique used to identify the “missed” samples, which is related to a slightly modified inverse recall metric.

RQ$_2$ Summary. Semantic and syntactic fixing actions show different trends when observing successful predictions, despite no clear trend being visible across the two datasets. However, for the failing cases, we could observe that the performance of the “Implement Filter/Sanitization” semantic actions is comparable to that of the “Insert - IfBlock” action, which is likely to be used in the same fixes. The best overall techniques were TC-SZZ, V-SZZ (based on the results on the V-SZZ dataset), DJ-SZZ, and V-SZZ (based on the results on our dataset). The number of correct predictions largely aligns with their recall scores. Looking at the failures related to the semantic actions, VCC-SZZ and VF-SZZ made the fewest number of incorrect predictions. Even more precise pairs, like ⟨RS-SZZ+L-SZZ⟩, can make many failures, requiring the addition of further techniques in the combination.



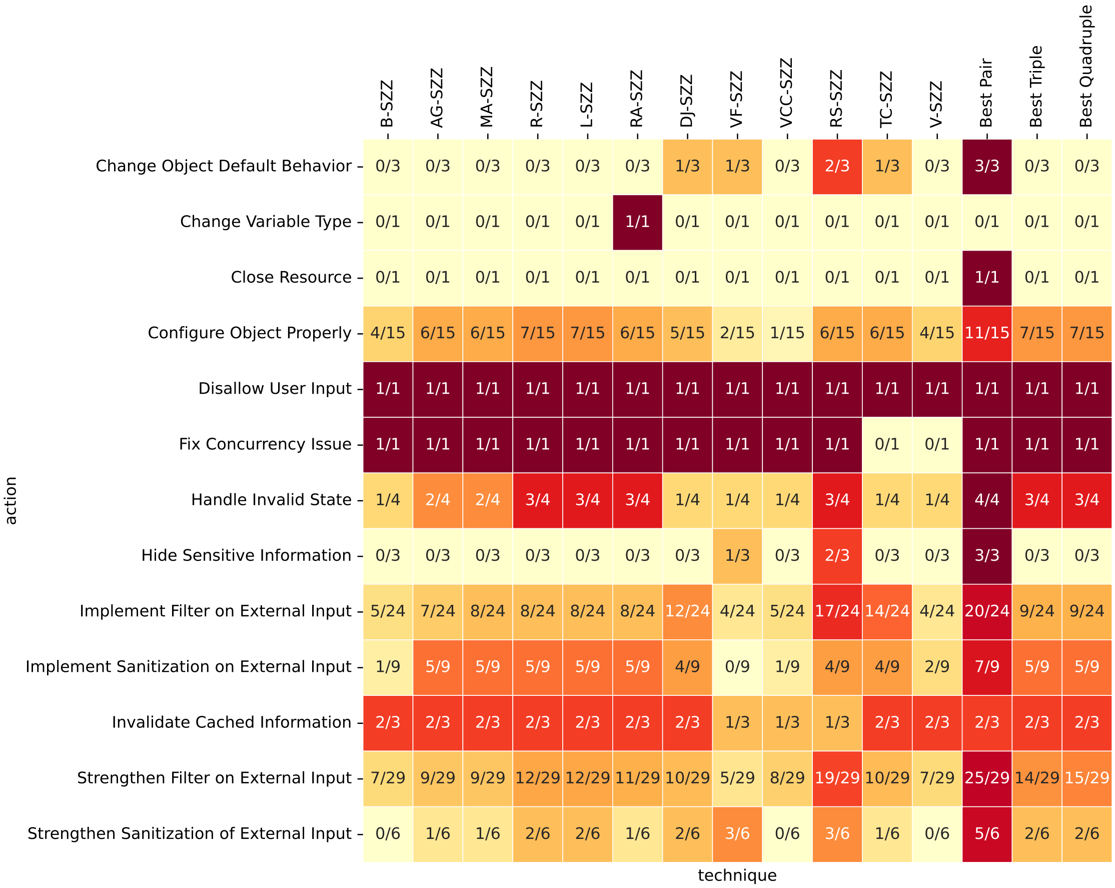
	B-SZZ	AG-SZZ	MA-SZZ	R-SZZ	L-SZZ	RA-SZZ	DJ-SZZ	VF-SZZ	VCC-SZZ	RS-SZZ	TC-SZZ	V-SZZ	Best Pair	Best Triple	Best Quadruple
Change Loop Exit Condition	1/1	1/1	1/1	1/1	1/1	1/1	0/1	1/1	1/1	1/1	0/1	1/1	1/1	1/1	1/1
Change Object Default Behavior	1/3	1/3	2/3	2/3	2/3	2/3	0/3	0/3	1/3	1/3	1/3	1/3	1/3	3/3	3/3
Change Variable Type	1/4	1/4	1/4	2/4	1/4	2/4	0/4	0/4	1/4	2/4	0/4	1/4	2/4	2/4	2/4
Configure Object Properly	4/17	5/17	5/17	7/17	6/17	4/17	0/17	3/17	2/17	3/17	1/17	2/17	4/17	10/17	11/17
Disallow User Input	1/2	1/2	1/2	2/2	1/2	1/2	1/2	0/2	0/2	1/2	1/2	1/2	1/2	2/2	2/2
Handle Invalid State	1/2	0/2	0/2	0/2	0/2	1/2	0/2	0/2	0/2	1/2	0/2	0/2	1/2	2/2	2/2
Implement Filter on External Input	4/12	6/12	7/12	9/12	8/12	8/12	0/12	2/12	3/12	3/12	2/12	2/12	4/12	11/12	12/12
Implement Sanitization on External Input	3/11	7/11	7/11	9/11	7/11	7/11	1/11	2/11	0/11	4/11	0/11	0/11	7/11	10/11	10/11
Strengthen Filter on External Input	5/11	5/11	5/11	7/11	5/11	5/11	2/11	1/11	4/11	3/11	1/11	2/11	4/11	10/11	10/11
Strengthen Sanitization of External Input	2/3	2/3	2/3	2/3	2/3	2/3	1/3	1/3	2/3	2/3	0/3	0/3	2/3	2/3	2/3
Use Security Robust Feature	1/5	3/5	3/5	4/5	3/5	3/5	1/5	0/5	1/5	1/5	0/5	1/5	1/5	4/5	4/5

Fig. 6. Incorrect predictions divided per semantic fixing action on the V-SZZ dataset.

7 Discussion and Implications

The results of our study highlighted numerous points worth discussing and several implications for researchers and practitioners, which we report in the following.

On the Performance of Existing VCC Mining Techniques. One of the key findings of our study concerns the relatively low performance of existing VCC mining techniques. According to our results, none of the experimented approaches, when taken alone, can go beyond 34% precision on our dataset. Curiously, many of the experimented techniques developed to overcome the limitations of the original SZZ (i.e., B-SZZ [93]). For example, ignoring the aesthetic changes, meta-changes, or refactoring operations, failed to achieve the intended goal, as the original SZZ is one of the most precise techniques and also has the highest F1-score, particularly in the V-SZZ dataset. We observed this phenomenon in our investigation with AG-SZZ, built on top of B-SZZ, and MA-SZZ, which was built on top of AG-SZZ. Both techniques were developed specifically to reduce the rate of false positives [18, 43]. In both datasets, we observed a precision drop. This could only be caused in two cases: either the technique introduced more false positives against their original claims, or the number of correctly predicted values decreased. With a more profound analysis, we found



action \ technique	B-SZZ	AG-SZZ	MA-SZZ	R-SZZ	L-SZZ	RA-SZZ	DJ-SZZ	VF-SZZ	VCC-SZZ	RS-SZZ	TC-SZZ	V-SZZ	Best Pair	Best Triple	Best Quadruple
Change Object Default Behavior	0/3	0/3	0/3	0/3	0/3	0/3	1/3	1/3	0/3	2/3	1/3	0/3	3/3	0/3	0/3
Change Variable Type	0/1	0/1	0/1	0/1	0/1	1/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1
Close Resource	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	0/1	1/1	0/1	0/1
Configure Object Properly	4/15	6/15	6/15	7/15	7/15	6/15	5/15	2/15	1/15	6/15	6/15	4/15	11/15	7/15	7/15
Disallow User Input	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1
Fix Concurrency Issue	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	1/1	0/1	0/1	1/1	1/1	1/1
Handle Invalid State	1/4	2/4	2/4	3/4	3/4	3/4	1/4	1/4	1/4	3/4	1/4	1/4	4/4	3/4	3/4
Hide Sensitive Information	0/3	0/3	0/3	0/3	0/3	0/3	0/3	1/3	0/3	2/3	0/3	0/3	3/3	0/3	0/3
Implement Filter on External Input	5/24	7/24	8/24	8/24	8/24	8/24	12/24	4/24	5/24	17/24	14/24	4/24	20/24	9/24	9/24
Implement Sanitization on External Input	1/9	5/9	5/9	5/9	5/9	5/9	4/9	0/9	1/9	4/9	4/9	2/9	7/9	5/9	5/9
Invalidate Cached Information	2/3	2/3	2/3	2/3	2/3	2/3	2/3	1/3	1/3	1/3	2/3	2/3	2/3	2/3	2/3
Strengthen Filter on External Input	7/29	9/29	9/29	12/29	12/29	11/29	10/29	5/29	8/29	19/29	10/29	7/29	25/29	14/29	15/29
Strengthen Sanitization of External Input	0/6	1/6	1/6	2/6	2/6	1/6	2/6	3/6	0/6	3/6	1/6	0/6	5/6	2/6	2/6

Fig. 7. Incorrect predictions divided per semantic fixing action on our dataset.

that the number of false positives is lower than B-SZZ, as expected, though some VCCs are totally missed, explaining the precision drop. This has a serious and worrisome implication for the samples' reliability in most empirical studies involving mining VCCs. We believe software security researchers should assess the performance of the VCC mining tasks before employing them in larger-scale analyses. In this way, they can better assess the validity of conclusions drawn from their experimentation. We argue about the actual effectiveness of the existing SZZ variants, demanding more solid experimentation to ensure that they actually reduce the number of false positives as expected and keep the true positives unaffected. In particular, assess the impact of the individual filters employed by the various techniques to mitigate noise, e.g., handle refactorings [56], and craft more robust ones. The insights provided by our study, e.g., the combination of multiple techniques, might be taken as input to investigate further the challenges of mining VCCs, apart from addressing the known limitations of SZZ-like techniques.

On the Need for Specific Mining Techniques for VCCs. Our manual analysis, made when building the benchmark dataset, revealed the existence of particular vulnerability fixes that made identifying VCCs more difficult. Hence, it was reasonable to ask whether the experimented techniques and approaches had trouble retrieving the correct VCCs.

To shed light on this aspect, we ran an additional qualitative analysis to understand the *fixing patterns* for which the techniques recurrently failed. We found that 48% of the erroneous classifications happened when the fixing actions consisted of *'adding a missing check or validation'*, irrespective of the assigned CWE. More in detail, in 82% of these cases, the missing checks were part of the precondition, translating into the addition of missing `if` statements. In such cases, the fixing commit added only new lines of code, which inevitably affected the behavior of the SZZ-based techniques. Indeed, one of the shortcomings of the `git blame`-based techniques is their unsuitability for commits without any deleted lines. The other more prominent erroneous classifications were caused either by *"changing configurations"* (16% of the cases), e.g., changing the parameters or options of frameworks, or *"solving improper sanitization"* (15% of the cases), e.g., removing special characters from HTTP headers or form inputs. This phenomenon represents a warning for the research community, as some types of vulnerabilities might be hard to identify, characterize, and predict; in this sense, empirical studies looking at vulnerabilities in the wild should be carefully crafted to consider the particular vulnerability type when collecting data automatically. This represents a call for novel mining techniques specifically tailored to recurrent fixing patterns, i.e., adding missing checks, modifying configuration files, etc.

On the Lack of Labeled Datasets of VCCs. Our literature search allowed us to discover the lack of datasets containing a validated set of VCCs. The only dataset we could find was connected to V-SZZ [3], about which we elaborated on its validity. This absence makes studies like this more challenging to enact, as they require an initial greater effort to create new datasets. Indeed, exploring the full change history of a project to find the correct and complete set of VCCs requires tremendous human effort. The approach presented by Rosa et al.[68, 69] could be employed to compensate for this absence, as it can be executed in a fully automated fashion. In essence, it consists of building a "developer-informed" oracle for evaluating SZZ variants through a textual analysis of the commit messages, aiming at finding explicit mentions of the report (e.g., GitHub issues) of the fixed bug and mentions to the likely "culprit" commits. The core assumption affirms that developers' words are to be considered more reliable, as they have better knowledge of the context than external inspectors. Nevertheless, we find two main limitations if we were to apply it in our context. First, the approach has only been employed in the context of traditional bugs and reports, so it should be reapplied for vulnerabilities and CVE records, which does not guarantee that the observed effectiveness holds. Second, the approach can find a highly precise set of commits contributing to the introduction of bugs (vulnerabilities in our context), but it still can miss other candidates. In other words, the approach is meant to have a reliable labeled dataset rather than a complete one. While the first limitation can be addressed by replicating the approach in the context of security bugs, we believe the second one represents a call for novel techniques that can find a balance between correctness and completeness.

On Selecting the Right VCC Mining Technique. The results of the pattern analysis in **RQ$_2$** indicate that it is hard to determine the most suitable VCC mining technique based on the syntactic characteristics of the input fixing commits. Furthermore, the results show that techniques performing well in general metrics also excel across most analyzed fixing actions, with few exceptions like "Disallow User Input" and "Strengthen Sanitization of External Input." However, these exceptions are based on very few cases in the dataset (2 and 3 cases), making it difficult to draw definitive conclusions. Another notable point is that the "Implement Filter on External Input" category appears to be one of the most challenging to predict accurately. This suggests that fixes in this class may introduce new code, likely if-statements, which the techniques struggle to analyze. This assumption is supported by the similar performance of the techniques on the syntactic fixing action "Insert-IfBlock".

On the Uses of VCC Mining Techniques: The Researcher's Perspective. Researchers in software security can leverage the potential of automated VCC mining for several uses. The most prominent application stands in the creation of a dataset of examples of VCCs to train (and validate) just-in-time vulnerability prediction models [46, 58, 60, 65]. That is, machine-learning models that are instructed to recognize VCCs of known vulnerabilities (since they stem from fixing commits) with the hope that they can generalize to identify commits contributing to "not known yet" vulnerabilities. While this research trend has been conducted for several years already, the ultimate results still appear suboptimal [46, 65], which is likely due to the ineffectiveness of the methods used to build the ground truth (often, SZZ-based techniques) and the extent to which they greatly affect the performance at testing time. In this respect, a focused analysis on the impact of the various ground truth methods—akin to the study by Fan et al. [21]—in the context of just-in-time vulnerability prediction is needed. The results of our benchmark study want to alert researchers to be more careful when considering existing automated methods to build a labeled dataset to train such models. Another use of VCC mining techniques was demonstrated by Bao et al. with V-SZZ [3], which was used to localize the software versions affected by a known vulnerability (disclosed via CVE). Namely, by exploiting the ability of V-SZZ to find VCCs accurately, the authors pinpointed the set of *release tags* containing the VCCs retrieved. This approach goes in the direction of circumventing the inaccuracies of the *"Known Affected Software Configurations"* reported on the National Vulnerability Database through CPE (Common Platform Enumartion) [30, 59]. The results of our benchmark study provide a hint of the most promising techniques that likely perform well in finding the right affected version ranges. Lastly, we argue that a good catalog of VCCs opens up other research applications, such as enabling the inference of patterns that indicate whether commits are about to introduce a vulnerability. That is, VCCs of similar vulnerability types might tend to make similar changes, revealing *"vulnerability contribution patterns"*—which can be seen as a sort of "mirrored" version of vulnerability fix pattern [9]. Such contribution patterns allow a better characterization of security vulnerabilities regarding the *root causes* leading to their introduction, particularly from a syntactic perspective—akin to the study on fix commits by Canfora et al. [11]—and for what concerns the development process [36, 51].

On the Uses of VCC Mining Techniques: The Developer's Perspective. Software developers have undoubtedly different objectives and priorities from researchers. In fact, they could benefit from historical vulnerability data if they can practically help them improve the security of their code and simplify their daily activities. Therefore, they could leverage automated VCC mining to expand their knowledge about past vulnerabilities that affected the projects in which they are working—particularly, such techniques are limited only to open-source projects. In particular, VCC mining techniques can be used to build a *knowledge base* of vulnerabilities and related bad commits that should be avoided in the future. Indeed, developers can learn lessons from the errors made in the past and collect them into an internal "wiki", which can be helpful for colleagues or newly hired developers to prevent them from repeating the same mistakes made in the past. This aligns with the objectives of the Vulnerability History Project [50], which aims to establish a "museum" of past mistakes (i.e., inserting vulnerable code) made in open-source projects. Concerning the VCC mining techniques themselves, developers are likely not concerned with their computational costs, as the techniques we benchmarked are all lightweight, requiring at most 5 minutes to find the VCCs of a vulnerability in a repository with 50,000 commits (not including download of the repository). Since the retrieval of VCCs is only meant to be run once per vulnerability (as soon as a fix commit is known), the computational cost is acceptable. In terms of retrieval accuracy, developers might be interested in the good recall scores achieved by the techniques specialized for vulnerabilities, i.e., VF-SZZ, V-SZZ, and VCC-SZZ—though they have conflicting results depending on the dataset. Recall looks appealing to developers as knowing all VCCs of a vulnerability gives full knowledge about the root causes behind it, even if this implies more false positives and longer manual inspection time. Nevertheless, due to the occasional use of such techniques in individual projects (as vulnerabilities occur rarely in the average project), the larger effort for reviewing more commits appears acceptable.

On the Future Research on VCC Mining. The results of our study, particularly **RQ**$_2$, highlighted some interesting horizons for VCC mining research. Our findings revealed that using machine learning to "ensemble" existing VCC mining techniques may significantly boost the individual approaches. More particularly, the *Random Forest* classifier was the best to combine the results of individual VCC detection approaches, being able to improve the precision of the mining task substantially. This represents a key result for the software engineering research community: researchers might want to use ensemble learning methods to boost the conclusion validity of their studies. At the same time, our findings provide a ground for further enhancements. Further research on the use of machine learning for VCC mining might be exploited to provide novel approaches and methods to identify vulnerabilities arising in the change history of software systems.

8 Threats to Validity

8.1 Construct Validity

To find VCC mining techniques and datasets, we chose Google Scholar and Google Search to cover a wide spectrum of both peer-reviewed and gray literature, reducing the risk of missing essential work. The search term was curated by concatenating relevant keywords for the topic, along with their synonyms. The search process was performed until saturation (results became irrelevant to the topic). Lastly, we mitigate the risk of losing precious pieces of information by letting the two reviewers jointly decide the borderline cases.

Furthermore, our search query was run on March 24, 2023. New techniques and datasets could have been released. So, we periodically re-run our search query to monitor new resources that introduce new techniques and datasets suitable for our study. We found a new technique called "Neural SZZ" [78], which relies on a heterogeneous graph attention network to filter out lines deleted in a bug fix commit that likely did not remove the bug root cause. Then, the standard SZZ [93] is run on the remaining lines. We could not include this technique in our experimentation due to errors encountered while replicating this approach without disrupting the released pipeline, risking introducing defects.

Even though we tried to employ the mining techniques as close to their original implementation as possible, some minor adjustments had to be made in some cases—for example, V-SZZ was integrated within the version of PySZZ we used for the other traditional techniques (B-SZZ, AG-SZZ, etc.). In this respect, we relied on ready-to-use tools or scripts like PySZZ [68] or SZZUnleashed [7] to minimize the errors we might introduce when reproducing the technique. Yet, we could not exclude possible drifts from their original definition, but we still opted to assess the validity of each reimplementation by examining their output manually to see if they are in line with their supposed behavior.

Project-KB [63] was used as the main source of both datasets we employed to compare the VCC mining techniques. Project-KB is a reliable source containing manually curated links of known vulnerabilities with their fixing commits. Before running the experimented techniques, we ensured Project-KB had no duplicate records or forked repositories, i.e., no vulnerabilities mapped to fixing commits of different repositories that stem from the same base project, which would inevitably add noise to the benchmark and affect the credibility of the performance observed.

Our dataset comprised 100 known vulnerabilities taken from Project-KB, which were manually inspected to retrieve their set of VCCs. We could not map the entire Project-KB dataset due to the effort required to manually retrieve the actual VCCs—indeed, more than 250 person-hours were required to map 100 vulnerabilities. To avoid any selection bias, we randomly sampled a reasonable number of vulnerabilities that the inspector could afford to analyze. Before starting the manual inspection, we checked their main characteristics—i.e., the involved projects, the fixing commit sizes, etc.—to ensure they were diversified enough. The analysis of the fixing commits to retrieve the actual VCCs was mainly driven by human effort; hence, we cannot exclude errors made in the process. We mitigated this risk by handing out a subset of 30 vulnerabilities to another security researcher to validate the findings, as explained in Section 4.2. Moreover, the VCC mapping was only restricted to the scope of the repository's change history, meaning that any modifications made before the project was ported into GIT from other versioning systems are lost.

The inspector who assessed the VCC in our dataset was a security-trained researcher with experience mining software repositories. Unfortunately, we could not ensure that the inspection found all the VCCs associated with a given vulnerability. Indeed, guaranteeing the retrieval of all VCCs is only possible if the entire repository's commit log is inspected, which would have been infeasible due to the presence of projects containing thousands of commits. Therefore, we accepted the fact that our dataset could be incomplete, though we are highly confident about the correctness of the labeled VCCs. Still concerning the dataset construction, the human-driven inspection process incorporated a degree of realism from the outset. Specifically, we applied structured guidelines that accounted not only for the textual content of the commit, but also for its timing, its linkage to the corresponding fixing commit, and the plausibility of its contribution to the disclosed vulnerability. In this way, aspects typically evaluated through separate "realism" assessments are already implicitly embedded in our benchmark. This design choice allowed us to rely on standard precision and recall metrics while preserving a strong alignment with real-world plausibility.

RA-SZZ [56] requires the refactoring instances to be excluded while running `git-blame`. The correctness of the found refactoring instances inevitably influences the performance of RA-SZZ. For this reason, RA-SZZ relies on RefactoringMiner [81], a state-of-the-art tool for automated refactoring detection. The tool was evaluated on 538 commits from 185 open-source Java projects, scoring a precision of 97% and recall of 87%, overcoming the capabilities of other similar detectors [81]. We deemed such performance adequate for the intended goal of RA-SZZ.

The syntactic analysis on the fix commit made during $\mathbf{RQ}_2$ is closely related to the process used by Canfora et al. [11]. We leveraged Coming [48], a well-known and reliable tool, with a reported true positive rate of 93%, used in mining software repository research. This allowed us to reduce the risk of improperly retrieving the fixed actions. Regarding the semantic analysis, the inspectors followed an inductive approach, expressing their judgments independently and then ruling out unlikely actions in case of disagreements. Then, both inspectors made a new pass to ensure all the samples had the right actions assigned according to the final agreed set of fixing actions.

8.2 Conclusion Validity

The selected techniques were evaluated using precision, recall, and F1-score. Such metrics can evaluate the extent to which an information retrieval algorithm can achieve its two fundamental goals, i.e., how many retrieved elements were correct (precision) and how many correct elements were retrieved (recall). Such metrics were widely employed in similar studies comparing multiple techniques to retrieve bug-inducing commits [57, 68].

The machine learning classifiers were trained and tested following a 10-fold cross-validation schema, which allows the evaluation of the models under different compositions of the datasets. Such a practice is meant to identify the effect of randomness, such as having a fortunate test dataset made of instances that are easy to classify. Not only do we report and discuss the average performance scored by the cross-validated models, but we also plot the distribution of the metrics scored in the 10 iterations.

Another factor that could influence the results obtained is the development model of each project in our benchmarks, i.e., how developers distribute their work across commits. Indeed, projects having commits making cohesive and limited changes [31] can make the retrieval algorithms match the right set of commits more easily, reducing the number of false positives. Despite having made many efforts to handle the irrelevant changes [21], SZZ is still weak to the style adopted by developers when making changes.

In numerous instances (nine in our dataset and four in the V-SZZ dataset), we observed that the surveyed techniques flagged the respective projects' initial commits as VCC. With a closer inspection, such commits contain many kinds of seemingly unrelated changes, forming what is known in the literature as "large commits" [32]. We found that in most cases, the commit corresponds to a migration from a different version control system, e.g., Subversion, flattening the full content of the repository into a single git commit. This phenomenon does not allow any SZZ-like techniques to go back from the moment the repository was migrated. In any case, such commits did not hinder the technique's precision as they correctly contain the contribution to the vulnerability, though mixed with many other changes.

8.3 External Validity

Although most SZZ-like techniques are, in principle, applicable across multiple programming languages, our experiments focused exclusively on Java projects. This decision was influenced by the greater availability and maturity of datasets containing mapped vulnerability-fixing commits for Java, which remain limited or less well-established for other languages. As a result, the performance and behavior of the assessed techniques may vary when applied to projects written in different programming languages. In this respect, we plan to conduct additional cross-language analyses as part of our future research agenda.

As Section 4.2 described, the V-SZZ dataset is limited to vulnerabilities with specific properties. In addition, the dataset size makes it difficult to generalize our findings to a larger set of vulnerabilities. The dataset that the inspector manually labeled bypassed these restrictions by removing several exclusion criteria for vulnerabilities, hopefully expanding the representativeness of our dataset. The use of two datasets allowed us to cover a large variety of different vulnerabilities. Nevertheless, the total number of examples was still limited, not allowing us to derive statistically sound conclusions about the findings observed in $\mathbf{RQ}_2$.

9 Related Work

Our multivocal literature review revealed that most techniques for retrieving bug- or vulnerability-inducing commits leverage the basic idea behind the SZZ algorithm, extensively described in Section 3. Here we report the key works that compared and evaluated the SZZ variants to retrieve bug-introducing commits.

Rosa et al. proposed a method to build a *developer-informed* oracle for evaluating SZZ variants [68]. They employed Natural Language Processing (NLP) techniques to identify bug-fixing commits in which developers explicitly refer to the commit(s) that introduced the bug (e.g., *'Fixed bug caused by 2f7806a9"*'). This method allowed the building of a sound ground truth that directly stems from the original developers, who are supposed to have maximum knowledge of the code they modify. Then, the authors used such an oracle to evaluate nine SZZ variants in terms of Precision, Recall, and F-measure. They found that R-SZZ [19], which returns the most recent bug-inducing commit among the set of commits found with `git blame` from the fixing commit, performs best on their dataset reaching ~64% in precision and recall. The same authors later extended the study adding two new heuristics to handle the main limitations seen for all SZZ variants [69]. Such heuristics were meant to handle (1) the fixing commits made of only added lines and (2) bypass rebase and revert commits. The two new heuristics can be equipped to any SZZ variant and were seen to improve their general performance, though not always with a noticeable difference.

da Costa et al. [18] evaluated five SZZ variants on ten open-source projects. Differently from other studies, the goal was not to determine the technique with the best retrieval performance (i.e., higher precision or recall) but rather to analyze some characteristics connected to the retrieved bug-inducing commits. Specifically, they measured the degree of (1) agreement with what developers believe is the first appearance of the bug, (2) the distance between the moment the bugs are introduced and their discovery, and (3) the distance between the retrieved bug-inducing commits for a given bug. The analyses showed that in more than 50% of the cases, the retrieved commits come after the moment the developers believe the bug was really inserted. More interestingly, almost 50% of the bugs are caused by changes spanning several years. The authors believe such findings are not in line with how developers discover and fix bugs, questioning the reliability of SZZ-like techniques for retrieving the real bug-inducing commits.

Fan et al. [21] investigated the impact of *mislabeled changes*—i.e., when bug-inducing commits are deemed as safe or when safe commits are improperly flagged as bug-inducing—made by different SZZ variants on the performance and interpretation of JIT defect prediction models. They analyzed four SZZ variants employed in prior studies—i.e., B-SZZ (another name for the original SZZ [93]), AG-SZZ [43], MA-SZZ [18], and RA-SZZ [56]. Through a large-scale empirical investigation on a total of 126,526 changes mined from ten Apache projects, the authors found that the mislabeled changes by B-SZZ and MA-SZZ are not likely to cause a considerable performance reduction (in terms of AUC-ROC, F1-score, G-mean, and Recall), while AG-SZZ can cause a statistically significant performance reduction. In other words, AG-SZZ seems to be the least recommended technique to employ for building the ground truth of a JIT defect prediction model.

Petrulio et al. [61] conducted an in-depth investigation on the reliability and performance of SZZ in the multi-commit model, i.e., when developers complete tasks, e.g., implement a new feature by splitting their work into several commits. They compared B-SZZ, AG-SZZ, L-SZZ, and R-SZZ in the context of a *multi-commit model*, i.e., when bugs are fixed with several commits, based on the dataset by Rosa et al. [68] and another created by developers and Quality Assurance (QA) engineers in Mozilla. The results showed that the second dataset had significantly lower performance (~20% less) than the Rosa et al. dataset [68]. The authors assumed this is caused by different datasets or project-specific aspects like the programming language used.

Vandehei et al. proposed a method for retrieving the affected versions (AVs) of a defect, i.e., labeling defective classes in releases, and compare its accuracy with three SZZ variants, B-SZZ, DJ-SZZ and MA-SZZ[82]. The proposed method resulted in being significantly more precise (more than 14%) than all three SZZ variants in (i) retrieving AVs, (ii) labeling classes as defective, and (iii) developing defects repositories to perform feature selection.

To the best of our knowledge, no previous research to date has conducted an empirical evaluation as exhaustive as ours, unifying all currently acknowledged methods appropriate for vulnerability-contributing commits. This makes our study distinctly comprehensive and groundbreaking in both scope and depth of study.

10 Conclusion

The analyses carried out in this paper revealed that the available techniques to mine VCCs cannot precisely identify vulnerabilities over the change history of software systems. Combinations can relieve this limitation, but this comes at a significant expense of recall: The trade-off between the two performance metrics is too strong. Besides, the qualitative investigation revealed that techniques with higher recall perform equally well, irrespective of the fixing actions found in the input fixing commits. Yet, the specific trends change according to the benchmark dataset adopted. Such results help software security researchers comprehend the existing limits and challenges in automated VCC mining.

Our research agenda foresees the design of better retrieval techniques that can leverage additional instruments to triangulate the real VCCs, such as using the content in bug and vulnerability reports, employing automated security analysis tools, or running known test cases that reproduce the vulnerability [10]. Then, we envision a more solid modeling of the changes happening in vulnerability-contributing and fixing commits. The former allows the definition of "vulnerability contribution patterns", which can support an analysis of the *root causes* leading to the introduction of vulnerabilities. The latter, instead, is meant to refine the currently-known vulnerability fix patterns [9] with additional software metrics that capture the essence of developers' patches and allow a more thorough analysis of the causes leading the mining techniques to retrieve the wrong VCCs. Furthermore, we identify the assessment of how different VCC mining techniques influence downstream tasks such as vulnerability prediction, change classification, or CVE range refinement, as a valuable direction for future work. Such an investigation would help quantify the practical implications of choosing one technique over another and shed light on how trade-offs between precision and recall propagate into real-world software analytics pipelines. We also acknowledge the importance of realism-aware assessments and deeper investigations into the nature of false positives. This stems from the consideration that not all incorrect classifications are equally harmful or implausible (some may be borderline cases that require contextual interpretation). While the manually validated dataset employed in this study already incorporates a level of realism—based on observable cues such as commit timing, linkage to fixing commits, and plausibility of changes in relation to the disclosed vulnerability, a deeper, formal realism-based evaluation would be a valuable addition. Such an analysis would require a more precise understanding of development phases, internal timelines, release planning decisions, or contributor intent, which are typically accessible only through direct contact with original developers or through ethnographic methods. Lastly, we plan to observe how VCC mining behaves in other popular programming languages characterized by different families of vulnerabilities, such as C/C++ and Python.

Acknowledgments

This work was partially supported by the following projects:

- European Union's Horizon 2020 programme under grant agreement No. 952647 (AssureMOSS).
- EMELIOT national research project, funded by the MUR under the PRIN 2020 program (Contract 2020W3A5FY).
- Project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.
- EU-funded project Sec4AI4Sec (grant no. 101120393).

Conflict of Interest

The authors declare that they have no conflict of interest.

Data Availability Statement

The datasets built during the current study, plus the scripts used to analyze and generate the data, are available in a FigShare repository [33].

References

[1] Tamás Aladics, Péter Hegedűs, and Rudolf Ferenc. 2022. A Vulnerability Introducing Commit Dataset for Java: An Improved SZZ based Approach. In *Proceedings of the 17th International Conference on Software Technologies - ICSOFT*. INSTICC, SciTePress, 68–78. https://doi.org/10.5220/0011275200003266

[2] Ashish Arora, Ramayya Krishnan, Rahul Telang, and Yubao Yang. 2010. An empirical analysis of software vendors' patch release behavior: impact of vulnerability disclosure. *Information Systems Research* 21, 1 (2010), 115–132.

[3] Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. 2352–2364. https://doi.org/10.1145/3510003.3510113

[4] Yoav Benjamini and Yosef Hochberg. 1995. Controlling the False Discovery Rate: A Practical and Powerful Approach to Multiple Testing. *Journal of the Royal Statistical Society: Series B (Methodological)* 57, 1 (1995), 289–300. https://doi.org/10.1111/j.2517-6161.1995.tb02031.x arXiv:https://rss.onlinelibrary.wiley.com/doi/pdf/10.1111/j.2517-6161.1995.tb02031.x

[5] Karen M Benzies, Shahirose Premji, K Alix Hayden, and Karen Serrett. 2006. State-of-the-evidence reviews: advantages and challenges of including grey literature. *Worldviews on Evidence-Based Nursing* 3, 2 (2006), 55–61.

[6] Peter Bludau and Alexander Pretschner. 2022. 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)*. 1–12. https://doi.org/10.1109/SANER53432.2022.00012

[7] Markus Borg, Oscar Svensson, Kristian Berg, and Daniel Hansson. 2019. SZZ Unleashed: An Open Implementation of the SZZ Algorithm - Featuring Example Usage in a Study of Just-in-Time Bug Prediction for the Jenkins Project. In *Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation* (Tallinn, Estonia) *(MaLTeSQuE 2019)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3340482.3342742

[8] L. Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[9] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. 2023. APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities. *Empirical Software Engineering* 29, 1 (06 Dec 2023), 18. https://doi.org/10.1007/s10664-023-10415-7

[10] Quang-Cuong Bui, Riccardo Scandariato, and Nicolás E. Díaz Ferreyra. 2022. Vul4J: a dataset of reproducible Java vulnerabilities geared towards the study of program repair techniques. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 464–468. https://doi.org/10.1145/3524842.3528482

[11] Gerardo Canfora, Andrea Di Sorbo, Sara Forootani, Matias Martinez, and Corrado A. Visaggio. 2022. Patchworking: Exploring the code changes induced by vulnerability fixing activities. *Information and Software Technology* 142 (2022), 106745. https://doi.org/10.1016/j.infsof.2021.106745

[12] Gerardo Canfora, Andrea Di Sorbo, Sara Forootani, Antonio Pirozzi, and Corrado Aaron Visaggio. 2020. Investigating the vulnerability fixing process in OSS projects: Peculiarities and challenges. *Computers & Security* 99 (2020), 102067. https://doi.org/10.1016/j.cose.2020.102067

[13] Bruno Cartaxo, Gustavo Pinto, and Sergio Soares. 2020. *Rapid Reviews in Software Engineering*. Springer International Publishing, Cham, 357–384. https://doi.org/10.1007/978-3-030-32489-6_13

[14] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104 arXiv:https://doi.org/10.1177/001316446002000104

[15] T. Cover and P. Hart. 1967. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory* 13, 1 (1967), 21–27. https://doi.org/10.1109/TIT.1967.1053964

[16] John W Creswell. 1999. Mixed-method research: Introduction and application. In *Handbook of educational policy*. Elsevier, 455–472.

[17] R. Croft, Y. Xie, and M. Babar. 2023. Data Preparation for Software Vulnerability Prediction: A Systematic Literature Review. *IEEE Transactions on Software Engineering* 49, 03 (mar 2023), 1044–1063. https://doi.org/10.1109/TSE.2022.3171202

[18] Daniel Alencar da Costa, Shane McIntosh, Weiyi Shang, Uira Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering* 43 (7 2017), 641–657. Issue 7. https://doi.org/10.1109/TSE.2016.2616306

[19] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing text-based and dependence-based approaches for determining the origins of bugs. *Journal of Software: Evolution and Process* 26 (1 2014), 107–139. Issue 1. https://doi.org/10.1002/smr.1619

[20] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*. 313–324. https://doi.org/10.1145/2642937.2642982

[21] Yuanrui Fan, Xin Xia, Daniel Alencar da Costa, David Lo, Ahmed E. Hassan, and Shanping Li. 2021. The Impact of Mislabeled Changes by SZZ on Just-in-Time Defect Prediction. *IEEE Transactions on Software Engineering* 47, 8 (2021), 1559–1586. https://doi.org/10.1109/TSE.2019.2929761

[22] Yuanrui Fan, Xin Xia, David Lo, Ahmed E. Hassan, Yuan Wang, and Shanping Li. 2021. A Differential Testing Approach for Evaluating Abstract Syntax Tree Mapping Algorithms. In *Proceedings of the 43rd International Conference on Software Engineering* (Madrid, Spain) *(ICSE '21)*. IEEE Press, 1174–1185. https://doi.org/10.1109/ICSE43902.2021.00108

[23] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. 2006. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*. 131–138.

[24] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. 2014. An Empirical Study of Just-in-Time Defect Prediction Using Cross-Project Models. In *Proceedings of the 11th Working Conference on Mining Software Repositories* (Hyderabad, India) *(MSR 2014)*. Association for Computing Machinery, New York, NY, USA, 172–181. https://doi.org/10.1145/2597073.2597075

[25] Vahid Garousi, Michael Felderer, and Mika V Mäntylä. 2019. Guidelines for including grey literature and conducting multivocal literature reviews in software engineering. *Information and Software Technology* 106 (2019), 101–121.

[26] Inc. GitHub. 2025. CodeQL. https://codeql.github.com/docs/. [Online; accessed 08-05-2025].

[27] Sigi Goode, Chinho Lin, Jacob C Tsai, and James J Jiang. 2015. Rethinking the role of security in client satisfaction with Software-as-a-Service (SaaS) providers. *Decision Support Systems* 70 (2015), 73–85.

[28] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162

[29] Hazim Hanif, Mohd Hairul Nizam Md Nasir, Mohd Faizal Ab Razak, Ahmad Firdaus, and Nor Badrul Anuar. 2021. The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches. *Journal of Network and Computer Applications* 179 (2021), 103009.

[30] Yongzhong He, Yiming Wang, Sencun Zhu, Wei Wang, Yunjia Zhang, Qiang Li, and Aimin Yu. 2024. Automatically Identifying CVE Affected Versions With Patches and Developer Logs. *IEEE Transactions on Dependable and Secure Computing* 21, 2 (2024), 905–919. https://doi.org/10.1109/TDSC.2023.3264567

[31] Kim Herzig and Andreas Zeller. 2013. The impact of tangled code changes. In *2013 10th Working Conference on Mining Software Repositories (MSR)*. 121–130. https://doi.org/10.1109/MSR.2013.6624018

[32] Abram Hindle, Daniel M. German, and Ric Holt. 2008. What Do Large Commits Tell Us? A Taxonomical Study of Large Commits. In *Proceedings of the 2008 International Working Conference on Mining Software Repositories* (Leipzig, Germany) *(MSR '08)*. Association for Computing Machinery, New York, NY, USA, 99–108. https://doi.org/10.1145/1370750.1370773

[33] Torge Hinrichs, Emanuele Iannone, Tamás Aladics, Péter Hegedűs, Andrea De Lucia, Fabio Palomba, and Riccardo Scandariato. 2024. Back to the Roots: Assessing Mining Techniques for Java Vulnerability-Contributing Commits - Online Appendix. https://doi.org/10.6084/m9.figshare.24305659.

[34] Kevin Hogan, Noel Warford, Robert Morrison, David Miller, Sean Malone, and James Purtilo. 2019. The challenges of labeling vulnerability-contributing commits. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 270–275.

[35] Emanuele Iannone, Zadia Codabux, Valentina Lenarduzzi, Andrea De Lucia, and Fabio Palomba. 2023. Rubbing salt in the wound? A large-scale investigation into the effects of refactoring on security. *Empirical Software Engineering* 28, 4 (24 May 2023), 89. https://doi.org/10.1007/s10664-023-10287-x

[36] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, and Fabio Palomba. 2023. The Secret Life of Software Vulnerabilities: A Large-Scale Empirical Study. *IEEE Transactions on Software Engineering* 49, 1 (2023), 44–63. https://doi.org/10.1109/TSE.2022.3140868

[37] Julian Jang-Jaccard and Surya Nepal. 2014. A survey of emerging threats in cybersecurity. *J. Comput. System Sci.* 80, 5 (2014), 973–993.

[38] Tian Jiang, Lin Tan, and Sunghun Kim. 2013. Personalized defect prediction. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 279–289. https://doi.org/10.1109/ASE.2013.6693087

[39] Matthieu Jimenez, Mike Papadakis, and Yves Le Traon. 2016. Vulnerability prediction models: A case study on the linux kernel. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 1–10.

[40] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M German, and Daniela Damian. 2016. An in-depth study of the promises and perils of mining GitHub. *Empirical Software Engineering* 21 (2016), 2035–2071.

[41] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E. Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2013. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2013), 757–773. https://doi.org/10.1109/TSE.2012.70

[42] Sunghun Kim, E. James Whitehead, and Yi Zhang. 2008. Classifying Software Changes: Clean or Buggy? *IEEE Transactions on Software Engineering* 34, 2 (2008), 181–196. https://doi.org/10.1109/TSE.2007.70773

[43] Sunghun Kim, Thomas Zimmermann, Kai Pan, and E. James Jr. Whitehead. 2006. Automatic Identification of Bug-Introducing Changes. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. 81–90. https://doi.org/10.1109/ASE.2006.23

[44] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. 2021. The do's and don'ts of infrastructure code: A systematic gray literature review. *Information and Software Technology* 137 (2021), 106593.

[45] Tien-Duy B. Le, Mario Linares-Vasquez, David Lo, and Denys Poshyvanyk. 2015. RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information. In *2015 IEEE 23rd International Conference on Program Comprehension*. 36–47. https://doi.org/10.1109/ICPC.2015.13

[46] Francesco Lomio, Emanuele Iannone, Andrea De Lucia, Fabio Palomba, and Valentina Lenarduzzi. 2022. Just-in-time software vulnerability detection: Are we there yet? *Journal of Systems and Software* 188 (2022), 111283. https://doi.org/10.1016/j.jss.2022.111283

[47] Yunbo Lyu, Hong Jin Kang, Ratnadira Widyasari, Julia Lawall, and David Lo. 2024. Evaluating SZZ Implementations: An Empirical Study on the Linux Kernel. *IEEE Transactions on Software Engineering* 50, 9 (2024), 2219–2239. https://doi.org/10.1109/TSE.2024.3406718

[48] Matias Martinez and Martin Monperrus. 2019. Coming: A Tool for Mining Change Pattern Instances from Git Commits. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 79–82. https://doi.org/10.1109/ICSE-Companion.2019.00043

[49] Dean Richard McKinnel, Tooska Dargahi, Ali Dehghantanha, and Kim-Kwang Raymond Choo. 2019. A systematic literature review and meta-analysis on artificial intelligence in penetration testing and vulnerability assessment. *Computers & Electrical Engineering* 75 (2019), 175–188.

[50] Andy Meneely, Aryan Jha, Ryan Borger, sxm7571, Matt Thyng, Edward Wong, 17mgeffert, Kyle Mirley, Nicholas Valletta, Austin, Mark Vittozzi, Vincent Li, NBrockman, Matthew Vogt, olucomedy, Christopher Savan, austinsimmons1, Jesse Chen, Adam Del Rosso, Eric Tiano, aldenreed, Benjamin Meyers, kzalbak, Eric Lin, Elijah Cantella, dxc7790, Calvin Do, akn4743, Jeremy De La Cruz, and Marianna Sternefeld. 2023. Vulnerability-HistoryProject/vulnerabilities: First monorepo release! https://doi.org/10.5281/zenodo.7558232

[51] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodríguez Tejeda, Matthew Mokary, and Brian Spates. 2013. When a Patch Goes Bad: Exploring the Properties of Vulnerability-Contributing Commits. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 65–74. https://doi.org/10.1109/ESEM.2013.19

[52] Ayse Tosun Misirli, Emad Shihab, and Yasukata Kamei. 2016. Studying high impact fix-inducing changes. *Empirical Software Engineering* 21 (2016), 605–641.

[53] Patrick J. Morrison, Rahul Pandita, Xusheng Xiao, Ram Chillarege, and Laurie Williams. 2018. Are vulnerabilities discovered and resolved like other defects? *Empirical Software Engineering* 23, 3 (01 Jun 2018), 1383–1421. https://doi.org/10.1007/s10664-017-9541-1

[54] Nuthan Munaiah, Felivel Camilo, Wesley Wigham, Andrew Meneely, and Meiyappan Nagappan. 2017. Do bugs foreshadow vulnerabilities? An in-depth study of the chromium project. *Empirical Software Engineering* 22, 3 (June 2017), 1305–1347.

[55] Kartik Nayak, Daniel Marino, Petros Efstathopoulos, and Tudor Dumitraş. 2014. Some vulnerabilities are different than others: Studying vulnerabilities and attack surfaces in the wild. In *Research in Attacks, Intrusions and Defenses: 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings 17*. Springer, 426–446.

[56] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2018. The impact of refactoring changes on the SZZ algorithm: An empirical study. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 380–390. https://doi.org/10.1109/SANER.2018.8330225

[57] Edmilson Campos Neto, Daniel Alencar da Costa, and Uirá Kulesza. 2019. Revisiting and Improving SZZ Implementations. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 1–12. https://doi.org/10.1109/ESEM.2019.8870178

[58] Son Nguyen, Thu-Trang Nguyen, Thanh Trong Vu, Thanh-Dat Do, Kien-Tuan Ngo, and Hieu Dinh Vo. 2024. Code-centric learning-based just-in-time vulnerability detection. *Journal of Systems and Software* 214 (2024), 112014. https://doi.org/10.1016/j.jss.2024.112014

[59] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering* 21, 6 (01 Dec 2016), 2268–2297. https://doi.org/10.1007/s10664-015-9408-2

[60] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 426–437. https://doi.org/10.1145/2810103.2813604

[61] Fernando Petrulio, David Ackermann, Enrico Fregnan, Gül Calikli, Marco Castelluccio, Sylvestre Ledru, Calixte Denizet, Emma Humphries, and Alberto Bacchelli. 2022. SZZ in the time of Pull Requests. arXiv:2209.03311 [cs.SE]

[62] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. 2015. Impact assessment for vulnerabilities in open-source software libraries. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 411–420.

[63] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 383–387. https://doi.org/10.1109/MSR.2019.00064

[64] Christophe Rezk, Yasutaka Kamei, and Shane McIntosh. 2022. The Ghost Commit Problem When Identifying Fix-Inducing Changes: An Empirical Study of Apache Projects. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3297–3309. https://doi.org/10.1109/TSE.2021.3087419

[65] Timothé Riom, Arthur Sawadogo, Kevin Allix, Tegawendé F. Bissyandé, Naouel Moha, and Jacques Klein. 2021. Revisiting the VCCFinder approach for the identification of vulnerability-contributing commits. *Empirical Software Engineering* 26, 3 (29 Mar 2021), 46. https://doi.org/10.1007/s10664-021-09944-w

[66] Gema Rodríguez-Pérez, Gregorio Robles, Alexander Serebrenik, Andy Zaidman, Daniel M. Germán, and Jesus M. Gonzalez-Barahona. 2020. How bugs are born: a model to identify how bugs are introduced in software components. *Empirical Software Engineering* 25, 2 (01 Mar 2020), 1294–1340. https://doi.org/10.1007/s10664-019-09781-y

[67] Giovanni Rosa. 2024. PySZZ. https://github.com/grosa1/pyszz. Accessed: 2024-07-24.

[68] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating szz implementations through a developer-informed oracle. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 436–447.

[69] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2023. A comprehensive evaluation of SZZ Variants through a developer-informed oracle. *Journal of Systems and Software* 202 (2023), 111729. https://doi.org/10.1016/j.jss.2023.111729

[70] Hang Ruan, Bihuan Chen, Xin Peng, and Wenyun Zhao. 2019. DeepLink: Recovering issue-commit links based on deep learning. *Journal of Systems and Software* 158 (2019), 110406. https://doi.org/10.1016/j.jss.2019.110406

[71] Stuart Russell and Peter Norvig. 2010. *Artificial Intelligence: A Modern Approach* (3 ed.). Prentice Hall.

[72] Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Transactions on Software Engineering* 40, 10 (2014), 993–1006.

[73] Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering* 18 (2013), 25–59.

[74] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting Refactorings in Version Histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 269–279. https://doi.org/10.1109/MSR.2017.14

[75] Sonar. 2025. SonarQube. https://www.sonarsource.com/products/sonarqube/. [Online; accessed 08-05-2025].

[76] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*. ACM Press, New York, New York, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[77] Yan Sun, Qing Wang, and Ye Yang. 2017. FRLink: Improving the recovery of missing issue-commit links by revisiting file relevance. *Information and Software Technology* 84 (2017), 33–47. https://doi.org/10.1016/j.infsof.2016.11.010

[78] L. Tang, L. Bao, X. Xia, and Z. Huang. 2023. Neural SZZ Algorithm. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, USA, 1024–1035. https://doi.org/10.1109/ASE56229.2023.00037

[79] Lingxiao Tang, Chao Ni, Qiao Huang, and Lingfeng Bao. 2024. Enhancing Bug-Inducing Commit Identification: A Fine-Grained Semantic Analysis Approach. *IEEE Transactions on Software Engineering* 50, 11 (2024), 3037–3052. https://doi.org/10.1109/TSE.2024.3468296

[80] Christopher Theisen and Laurie Williams. 2020. Better together: Comparing vulnerability prediction models. *Information and Software Technology* 119 (2020), 106204.

[81] Nikolaos Tsantalis, Matin Mansouri, Laleh Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and Efficient Refactoring Detection in Commit History. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 483–494. https://doi.org/10.1145/3180155.3180206

[82] Bailey Vandehei, Daniel Alencar da Costa, and Davide Falessi. 2021. Leveraging the Defects Life Cycle to Label Affected Versions and Defective Classes. *ACM Trans. Softw. Eng. Methodol.* 30, 2, Article 24 (feb 2021), 35 pages. https://doi.org/10.1145/3433928

[83] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 262–273.

[84] Frank Wilcoxon. 1945. Individual Comparisons by Ranking Methods. *Biometrics Bulletin* 1, 6 (1945), 80–83. http://www.jstor.org/stable/3001968

[85] Chadd Williams and Jaime Spacco. 2008. SZZ Revisited: Verifying When Changes Induce Fixes. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems* (Seattle, Washington) *(DEFECTS '08)*. Association for Computing Machinery, New York, NY, USA, 32–36. https://doi.org/10.1145/1390817.1390826

[86] Ratsameetip Wita, Nattanatch Jiamnapanon, and Yunyong Teng-Amnuay. 2010. An ontology for vulnerability lifecycle. In *2010 Third International Symposium on Intelligent Information Technology and Security Informatics*. IEEE, 553–557.

[87] C. Wohlin, P. Runeson, M. Höst, M. C Ohlsson, B. Regnell, and A. Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[88] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3041–3058. https://www.usenix.org/conference/usenixsecurity21/presentation/woo

[89] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. ReLink: Recovering Links between Bugs and Changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 15–25. https://doi.org/10.1145/2025113.2025120

[90] Limin Yang, Xiangxue Li, and Yu Yu. 2017. VulDigger: A Just-in-Time and Cost-Aware Tool for Digging Vulnerability-Contributing Changes. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*. 1–7. https://doi.org/10.1109/GLOCOM.2017.8254428

[91] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. 2022. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering* 27, 5 (2022), 107.

[92] He Zhang, Xin Zhou, Xin Huang, Huang Huang, and Muhammad Ali Babar. 2020. An evidence-based inquiry into the use of grey literature in software engineering. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1422–1434.

[93] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes* 30 (7 2005), 1–5. Issue 4. https://doi.org/10.1145/1082983.1083147

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009