V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities

Lingfeng Bao
Zhejiang University
China
lingfengbao@zju.edu.cn

Xin Xia*
Huawei
China
xin.xia@acm.org

Ahmed E. Hassan
Queen's University
Canada
ahmed@cs.queensu.ca

Xiaohu Yang
Zhejiang University
China
yangxh@zju.edu.cn

ABSTRACT

Vulnerabilities publicly disclosed in the National Vulnerability Database (NVD) are assigned with CVE (Common Vulnerabilities and Exposures) IDs and associated with specific software versions. Many organizations, including IT companies and government, heavily rely on the disclosed vulnerabilities in NVD to mitigate their security risks. Once a software is claimed as vulnerable by NVD, these organizations would examine the presence of the vulnerable versions of the software and assess the impact on themselves. However, the version information about vulnerable software in NVD is not always reliable. Nguyen et al. find that the version information of many CVE vulnerabilities is spurious and propose an approach based on the original SZZ algorithm (i.e., an approach to identify bug-introducing commits) to assess the software versions affected by CVE vulnerabilities.

However, SZZ algorithms are designed for common bugs, while vulnerabilities and bugs are different. Many bugs are introduced by a recent bug-fixing commit, but vulnerabilities are usually introduced in their initial versions. Thus, the current SZZ algorithms often fail to identify the inducing commits for vulnerabilities. Therefore, in this study, we propose an approach based on an improved SZZ algorithm to refine software versions affected by CVE vulnerabilities. Our proposed SZZ algorithm leverages the line mapping algorithms to identify the earliest commit that modified the vulnerable lines, and then considers these commits to be the vulnerability-inducing commits, as opposed to the previous SZZ algorithms that assume the commits that last modified the buggy lines as the inducing commits. To evaluate our proposed approach, we manually annotate the true inducing commits and verify the vulnerable versions for 172 CVE vulnerabilities with fixing commits from two publicly available datasets with five C/C++ and 41 Java projects, respectively. We find that 99 out of 172 vulnerabilities whose version information is spurious. The experiment results show that our proposed approach can identify more vulnerabilities with the true inducing commits and correct vulnerable versions than the previous SZZ algorithms. Our approach outperforms the previous SZZ algorithms in terms of F1-score for identifying vulnerability-inducing commits on both C/C++ and Java projects (0.736 and 0.630, respectively). For refining vulnerable versions, our approach also achieves the best performance on the two datasets in terms of F1-score (0.928 and 0.952).

KEYWORDS

SZZ, Vulnerability, CVE

ACM Reference Format:
Lingfeng Bao, Xin Xia, Ahmed E. Hassan, and Xiaohu Yang. 2022. V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510113

1 INTRODUCTION

Software vulnerabilities are software security bugs, posing a severe threat to software systems. They can be exploited by attackers and result in a security breach or a violation of the system's security policy; for example, attackers can control the system or acquire private data by exploiting a vulnerability. To share information pertaining to publicly disclosed software vulnerabilities, the US National Institute of Standards and Technology (NIST) builds the National Vulnerability Database (NVD). These vulnerabilities are identified by CVE (Common Vulnerabilities and Exposures) IDs, containing a description, an estimation of the severity of the vulnerability, the know affected software, and related references.

Recently, software supply chain security is increasingly important for industrial companies as they usually use many OSS software in their projects. Due to concern about vulnerabilities in project dependencies, industrial companies often use Software Composition Analysis (SCA) tools (e.g., Snyk[1] and Whitesource[2]) to learn about vulnerabilities in their dependencies. The output of these SCA tools is based on CVE/NVD information. However, the software versions of CVE vulnerabilities are not always reliable. For

*Xin Xia is the corresponding author.

[1] https://snyk.io/

[2] https://www.whitesourcesoftware.com/

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510113

example, for CVE-2018-6621 as shown in the motivating example (see Section 2.2), its version range was 'in FFmpeg through 3.4.1' when created on February 04, 2018, but was updated to 'through 3.2' after three years (January 05, 2021). Furthermore, Nguyen et al. reported an inconsistent claim between the NVD team and software vendors in terms of 'vulnerable version' [38], i.e., NVD claimed vulnerable versions were taken from software vendors; whereas, software vendors claimed they did not know about this information. Additionally, Nguyen et al. reported that NVD usually applies a conservative rule: "If version X is vulnerable, then so are all its previous versions" [38]. So, many earlier versions of software might be incorrectly marked as vulnerable due to this bias. These erroneous/-conservative vulnerable versions would cause additional work for software companies. Additionally, many proprietary software systems might use an older version of an OSS software due to many reasons, for example, updating a newer version may break functionality. Once a vulnerability in an OSS software is publicly disclosed in the NVD, software vendors need to remediate its risk within a specified time according to remediation service level agreements (e.g., fix a vulnerability within 14 days[3]). Thus, if a tool can confirm that the vulnerability does not affect the OSS software they use, software vendors do not need to upgrade it and save lots of resources and time.

Previous research has typically focused on vulnerability management [19, 42, 48], or patches for vulnerabilities [8, 24, 29, 30, 48]. Few studies have investigated how to refine the software versions affected by CVE vulnerabilities. In a previous study, Nguyen et al. [38] proposed an approach, built on the inducing commits generated by the original SZZ algorithm [51], to detect spurious claims on vulnerable versions in NVD. They revealed systematic spurious vulnerability claims, i.e., NVD reported vulnerable, but there is no code evidence for the presence of the vulnerability. However, the original SZZ algorithm is designed to detect inducing commits for common bugs. A previous study [8] found that the empirical connection between bugs and vulnerabilities is considerably weak by evaluating code review effectiveness and participant experience. Many bugs are introduced by a recent bug-fixing commit [44], while vulnerabilities are usually *foundational* [39], i.e., introduced in their initial versions. Additionally, many SZZ variants have been proposed to boost the accuracy of the original SZZ [11, 12, 27, 59]. There are many difficulties to identify the bug-inducing commits by SZZ algorithms accurately [44, 45]. Therefore, we want to investigate whether the previous SZZ algorithms can identify inducing commits for CVE vulnerabilities and refine vulnerable versions based on inducing commits effectively.

In this study, we propose an approach based on an improved SZZ algorithm (called V-SZZ) for vulnerabilities to refine software versions affected by CVE vulnerabilities. To annotate the true inducing commits for vulnerabilities, we follow the model proposed by Rodríguez-Pérez et al. [44], which is designed for bug-inducing commit annotation. We choose two two publicly available datasets with vulnerability-fixing commits, containing five C/C++ and 41 Java projects, respectively. For the C/C++ dataset, we randomly select 20 CVEs with small size fixing commits (i.e, have larger than zero and less than or equal to five deleted lines) for each project. For the Java project, we select all CVEs with small size fixing commits. Finally, we annotated 172 CVE vulnerabilities. We run V-SZZ on the annotated vulnerability fixing commits to identify the inducing commits. Then, we identify the vulnerable versions based on the inducing commits generated by V-SZZ. We also select the original SZZ algorithm and its three variants (AG-SZZ, MA-SZZ, and RA-SZZ) as the baselines. The experiment results show that V-SZZ can correctly identify inducing commits for 88 and 59 vulnerabilities from C/C++ and Java datasets, respectively. V-SZZ also has a higher F1-score on C/C++ and Java datasets (0.736 and 0.630) than the previous SZZ algorithms.

To verify the software versions affected by CVE vulnerabilities, we first generate vulnerable versions based on the inducing commits we annotated. Then, we manually check the presence of the vulnerable code in the boundary versions to investigate whether the generated vulnerable versions are correct. Finally, we compare them with the version information of CVE vulnerabilities. We also identify vulnerable versions based on the inducing commits generated by V-SZZ and the previous SZZ algorithms. We find that 99 out of 172 vulnerabilities' version information is spurious. The experiment results show that our approach based on V-SZZ has a higher F1-score on C/C++ (0.928) and Java dataset (0.952) than the previous SZZ algorithms.

Our contributions can be summarized as follows:

(1) We build a dataset linking 172 CVE vulnerabilities and their inducing commits. This dataset covers vulnerabilities from five C/C++ projects and 41 Java projects. We also manually verify the software versions affected by these vulnerabilities and find that the version information of many CVE vulnerabilities is spurious.
(2) We propose an approach based on an improved SZZ algorithm (V-SZZ) to refine software versions affected by CVE vulnerabilities. The experiment results show that our approach can effectively identify inducing commits for vulnerabilities and refine vulnerable versions based on the inducing commits generated by V-SZZ.

Paper Structure: Section 2 describes the background of the SZZ algorithms and demonstrates two motivating examples. Section 3 presents our approach based on an improved SZZ to refine vulnerable versions for vulnerabilities in NVD. Section 4 describes the manual annotation for inducing commits for vulnerabilities and verification of vulnerable versions. Section 5 shows the results of our approach on identifying inducing commits, and refining vulnerable versions for vulnerabilities. Section 6 discusses the impact of duplicated changes on detecting vulnerable versions, the difference between vulnerabilities and common bugs, and the threats to validity. Section 7 reviews related work. Section 8 concludes the paper and discusses future directions.

2 BACKGROUND

In this section, we first introduce the original SZZ and its variants, then we show two motivating examples.

[3] https://www.ivanti.com/blog/industry-driving-toward-14-day-sla-vulnerability-remediation

2.1 SZZ algorithms

The original SZZ and its variants have been widely used in many previous studies to identify bug-inducing commits based on bug-fixing commits [2, 3, 5, 14, 25, 62]. SZZ algorithms assume that the lines of code that are modified by the bug-fixing commits are the same as or evolved from the lines of code that are modified by the bug-inducing commits [57]. In our study, we will use the following SZZ algorithms to investigate whether they can identify VICs for vulnerabilities, which is the same as a prior study [16]. In this paper, the implementation of these SZZ algorithms used is from the replication package of the study of Rosa et al. [45].

The original SZZ approach (B-SZZ) was proposed by Sliwerski et al. [51]. First, B-SZZ identifies bug-fixing changes in issue tracking systems based on the description of bug reports and commit messages. Then, it leverages the `diff` command provided by the version control systems (VCS) to identify the lines of code modified by the bug-fixing changes. Finally, B-SZZ traces back through the code history to identify the changes that introduce one or more of the buggy lines by using the `annotate` command in the VCS.

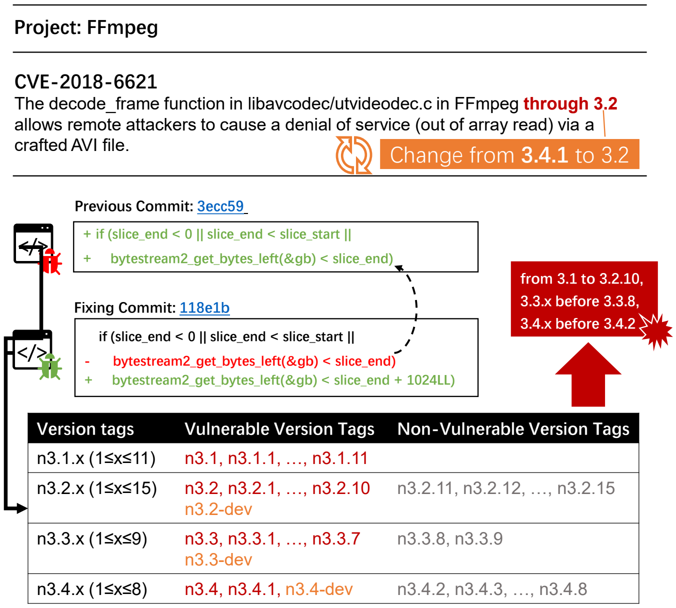
AG-SZZ was proposed by Kim et al. [27] since they observed that B-SZZ might consider non-semantic lines, including blank/comment lines and those involving format modifications (e.g., modifications to the code indentation) to be buggy lines. Kim et al. achieved a more precise result by using the annotation graph that provides more comprehensive information about line moves and modifications within a file than the `annotate` command.

MA-SZZ was proposed by Da Costa et al. [11] because they noticed that AG-SZZ flags meta-changes as potential bug-inducing changes due to the limitation of the annotation graph. Meta-changes are the commits that do not change source code, including branch changes (i.e., changes copying code from one branch to a new branch), merge changes (i.e., changes applying code modifications from one branch to another), and property changes (i.e., changes impacting file properties)

RA-SZZ was proposed by Neto et al. [35] since they observed that prior SZZ algorithms identify incorrect bug-inducing changes due to the impact of refactoring lines. Refactoring lines refer to those involving refactoring modifications (e.g., modification to a function name). There are two versions of RA-SZZ implemented by Neto et al. [35, 36], which are based on RefDiff [50] and Refactoring Miner [54], respectively. In our study, we select the RA-SZZ based on Refactoring Miner because it is more effective [36]. But RA-SZZ can only work on Java projects since the two refactoring-detection tools are only for Java code.

2.2 Motivating Examples

Nguyen et al. demonstrated that the versions affected by a CVE vulnerability might be spurious and could be refined by its fixing commits and inducing commits [38]. Figure 1 depicts a motivating example from CVE-2018-6621. According to its change history, we find that the version information in the CVE description was changed from *'3.4.1'* to *'3.2'*. This indicates that sometimes it is difficult for developers to determine the correct vulnerable versions. We apply the original SZZ approach to identify the inducing commit based on the fixing commit of this vulnerability. We assume that if a version contains the vulnerable code, this version is vulnerable; otherwise, the version is not vulnerable. Thus, the versions between the inducing commit and the fixing commit are vulnerable. Since a version in a git repository is usually assigned to a tag, we first identify the tags of vulnerable versions (see Figure 1). Then, we find that the vulnerable versions can be summarized into *'from 3.1 to 3.2.10, 3.3.x before 3.3.8, and 3.4.x before 3.4.2'*, which is different from the version information in the CVE description. Additionally, we find that several development versions (e.g., *3.2-dev*) are also vulnerable.



Project: FFmpeg

CVE-2018-6621

The decode_frame function in libavcodec/utvideodec.c in FFmpeg **through 3.2** allows remote attackers to cause a denial of service (out of array read) via a crafted AVI file.

Previous Commit: 3ecc59

```
+ if (slice_end < 0 || slice_end < slice_start ||
+     bytestream2_get_bytes_left(&gb) < slice_end)
```

Fixing Commit: 118e1b

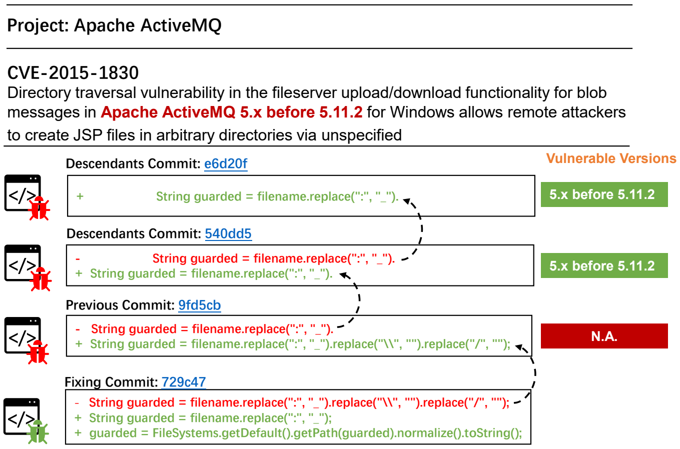
```
    if (slice_end < 0 || slice_end < slice_start ||
-     bytestream2_get_bytes_left(&gb) < slice_end)
+     bytestream2_get_bytes_left(&gb) < slice_end + 1024LL)
```

Version tags	Vulnerable Version Tags	Non-Vulnerable Version Tags
n3.1.x (1≤x≤11)	n3.1, n3.1.1, …, n3.1.11	
n3.2.x (1≤x≤15)	n3.2, n3.2.1, …, n3.2.10 n3.2-dev	n3.2.11, n3.2.12, …, n3.2.15
n3.3.x (1≤x≤9)	n3.3, n3.3.1, …, n3.3.7 n3.3-dev	n3.3.8, n3.3.9
n3.4.x (1≤x≤8)	n3.4, n3.4.1, n3.4-dev	n3.4.2, n3.4.3, …, n3.4.8

Figure 1: The first motivating example from CVE-2018-6621

However, the current SZZ algorithms cannot identify the true inducing commits for some vulnerabilities. Figure 2 present another motivating example from CVE-2015-1830. For the fixing commit of this vulnerability, the SZZ algorithms identify the commit that last modified the vulnerable line as the inducing commit (called 'Previous Commit' in Figure 2). But if we use this previous commit to infer the versions affected by this vulnerability, no vulnerable version would be detected. Nevertheless, this vulnerability still exists in the version with respect to the previous commit based on the CVE description and the fixing commit. Thus, we continue to identify the commits that modified the vulnerable line (called 'Descendants Commits'). As the change in the first descendants commit is just a format change, we regard the second descendants commit as the true inducing commit in which the vulnerable line of code was firstly added[4]. This vulnerability can be considered to be foundational [39] since it was present in an initial version. Based on this inducing commit, we infer the vulnerable versions are *'5.x before 5.11.2'*, which is consistent with the version information in the CVE description.

[4] This CVE is a directory traversal vulnerability, also known as the ../ (dot dot slash) attack. The exploit information in a reference mentions that the vulnerability can be exploited via a traversal path '/fileserver/../admin/'. The added API 'FileSystem.getDefault().getPath(String).normalize()' in the fixing commit removes redundant elements in a path including any "." or "directory/.." occurrences.



Directory traversal vulnerability in the fileserver upload/download functionality for blob messages in Apache ActiveMQ 5.x before 5.11.2 for Windows allows remote attackers to create JSP files in arbitrary directories via unspecified

Descendants Commit: e6d20f

```
+                    String guarded = filename.replace(":", "_").
```

Descendants Commit: 540dd5

```
-                    String guarded = filename.replace(":", "_").
+ String guarded = filename.replace(":", "_").
```

Previous Commit: 9fd5cb

```
-  String guarded = filename.replace(":", "_").
+ String guarded = filename.replace(":", "_").replace("\\", "").replace("/", "");
```

Fixing Commit: 729c47

```
-  String guarded = filename.replace(":", "_").replace("\\", "").replace("/", "");
+ String guarded = filename.replace(":", "_");
+ guarded = FileSystems.getDefault().getPath(guarded).normalize().toString();
```

Figure 2: The second motivating example from CVE-2015-1830

Furthermore, there are a few CVE vulnerabilities that have no vulnerable versions. For example, the version information of CVE-2017-5024 is *'FFmpeg in Google Chrome prior to 56.0.2924.76'*. This vulnerability indeed affects FFmpeg but the version information only mentions the version of Google Chrome. We identify the vulnerable version using SZZ algorithm based on and find the version information is *'from 2.0 to 3.2.15'*

From the first example, we show that the version information in some CVE vulnerabilities is spurious, and the vulnerable versions can be refined based on the fixing commits and the inducing commits generated by SZZ algorithms. However, the second example reveals that SZZ algorithms might not identify the true inducing commit. Because SZZ algorithms assume the commit that last modified the buggy line introduced the bugs. However, unlike common bugs, many vulnerabilities are foundational and introduced in an earlier version. Therefore, in this study, we want to investigate whether SZZ algorithms can effectively identify inducing commits for vulnerabilities and help refine the spurious versions of CVE vulnerabilities. We also want to improve the current SZZ algorithms to identify inducing commits for vulnerabilities and refine the vulnerable versions effectively.

3 THE PROPOSED APPROACH

Figure 3 presents the process of our approach. The input of our approach is a vulnerability with its fixing commits in a software project. First, we apply an improved SZZ algorithm to identify inducing commits for this vulnerability. Then, we scan the project repository to identify the commits that contain the same changes in the fixing commits and inducing commits. Finally, we infer the versions affected by this vulnerability based on the fixing commits and inducing commits.

SZZ for vulnerabilities. As shown in the second motivating example in Section 2.2, vulnerabilities can be introduced by a earlier change on the vulnerable code in the fixing commit. Moreover, Ozment and Schechter found that more than 50% vulnerabilities are foundational, i.e., they were present in their initial versions [39]. However, one assumption of SZZ algorithms is that *for each line that is modified by the commits for fixing a bug, the commit that last modified the line introduced this bug*, which is not suitable for the identification of many vulnerability-inducing commits. So, the current SZZ algorithms might not identify inducing commits for vulnerabilities effectively.

Therefore, we propose a modified SZZ algorithm called V-SZZ, aiming to identify inducing commits for vulnerabilities by going further back to get earlier changes on the vulnerable code. Similar to the study of Rodríguez-Pérez et al. [44], we call the immediately previous commits to the lines changed in the fixing commit as the **previous commits** and all the commits that previously modified the lines changed in the fixing commit as the **descendants commits**.

Given each modified line in a fixing commit, V-SZZ first identifies the previous commit using the `git blame` command. Similar to these improved SZZ algorithms [11, 27, 35], we ignore the non-semantic lines of code, including blank/comment lines and those involving format modifications. Then, to automatically identify descendants commits, we first leverage the line mapping algorithm to map the modified line to the line in the previous version. Then, we continue to use the `git blame` command to go back to identify the descendants commits.

For Java projects, we leverage the abstract syntax tree (AST) mapping algorithms [13, 15, 20] to map the modified lines between two versions of the source code files. We implement the line mapping algorithm provided by Fan et al. [17].

However, most previous studies for AST mapping algorithms focused on Java projects and did not provide the implementation for other languages. Thus, for C/C++ projects, we map the modified lines considering the string similarity and localization between the lines. We compute the line similarity based on the edit distance and set the similarity threshold is equal to 0.75. But this might result in some incorrect mappings, which would be discussed in the evaluation (see Section 5). If the line mapping algorithm can identify a corresponding line in the previous commit or descendants commits, we continue going back by running the `git blame` command for the line. This process stops until the line mapping algorithm identifies the line as a new line, the source code file of the line is firstly added, or the commit is the initial commit of the repository.

We consider the earliest commit as the inducing commit for each modified line of code since vulnerabilities are more likely to be foundational. Finally, V-SZZ can generate a set of inducing commits for a vulnerability.

Duplicated change detection. When using Git for software development, developers often create multiple branches to manage workflows of different versions in a project. To apply the change in a commit among multiple branches, developers often use the `git cherry-pick` command to pick it individually or the `git merge` command to integrate several changes. We find that the fixing commits or inducing commits of vulnerabilities are often applied to other branches. For example, in the first motivating example, the fixing commit *118e1b* has been cherry-picked three times.

To infer vulnerable versions accurately, we need to identify all the commits that contain changes in the fixing commits and inducing commits. If missing a fixing commit, some non-vulnerable versions would be considered vulnerable. On the other hand, if missing an

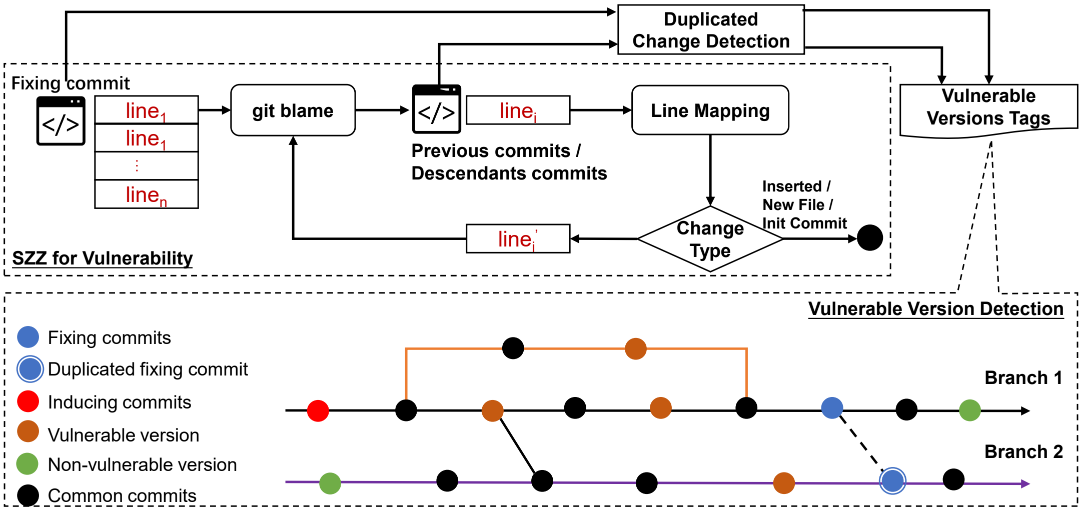
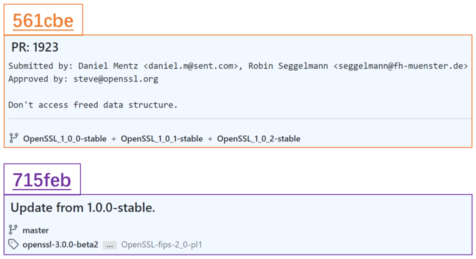


Figure 3: The proposed approach



561cbe

PR: 1923

```
Submitted by: Daniel Mentz <daniel.m@sent.com>, Robin Seggelmann <seggelmann@fh-muenster.de>
Approved by: steve@openssl.org

Don't access freed data structure.
```

OpenSSL_1_0_0-stable + OpenSSL_1_0_1-stable + OpenSSL_1_0_2-stable

715feb

Update from 1.0.0-stable.

master
openssl-3.0.0-beta2 ... OpenSSL-fips-2_0-pl1

Figure 4: Two commits in the project OpenSSL with the same change

inducing commit, some vulnerable versions would be considered as not vulnerable.

To detect the cherry-picked commits for a commit, we can search its commit id in the commit message because a sentence with the following pattern is automatically generated in the commit message: *"cherry picked from commit xxx"*. However, some commits can be copied to another branch without explicit information in the commit message. For example, Figure 4 present two commits with the same change from the project OpenSSL. The below commit in the figure is copied from the above commit for an upgrade. We use a hash-based duplicated change detection method to identify the commits copied to other branches without explicit messages. We first hash the hunks of all commits in a project repository. Given a commit, we search for the commits that contain all the hashes in it.

Vulnerable version detection. In this step, we want to identify the versions affected by a vulnerability based on a set of fixing commits and their corresponding inducing commits. In a Git workflow, project maintainers usually assign a tag to a commit for a release version, such as '*'n3.1'* in FFmpeg and *'OpenSSL_1_0_0-stable'* in OpenSSL. Hence, our goal is to identify these commits with version tags that are affected by a vulnerability. The idea is intuitive, i.e., the vulnerability introduced by the inducing commits will be transferred to the future commits until it gets fixed. Thus, the vulnerable versions are between the inducing commits and the fixing commits. We first identify the commits with version tags that are *reachable* to the inducing commits (denoted as C_i). A commit A is reachable to another commit B, indicating that A belongs to the parents or the ancestor commits of B. We also identify the commits with version tags that are reachable to the fixing commits (denoted as C_f). Then, we can get the set of the vulnerable versions C_v, i.e., the set difference of C_i and C_f (i.e., $C_i - C_f$).

As shown in Figure 3, the two commits in green color are not vulnerable because the one in the first branch is after a fixing commit and the other one in the second branch is not reachable to the inducing commit. The vulnerable versions in the figure are marked as brown. Finally, the output of this step is a set of version tags that are affected by a vulnerability, see an example in Figure 1.

4 MANUAL ANNOTATION

To investigate the effectiveness of the SZZ algorithms, we need a ground truth linking vulnerabilities and the true inducing commits. However, to the best of our knowledge, no such ground truth exists. Nguyen et al. manually verified the results generated by their approach for 80 vulnerabilities from two projects, but they did not make the dataset public [38]. Thus, in this section, we aim to build a ground truth linking vulnerabilities and their corresponding inducing commits. We first select two datasets with vulnerability-fixing commits. Then, we annotate the inducing commits for vulnerabilities by following the model proposed by Rodríguez-Pérez et al. [44], and manually verify the vulnerable versions based on the annotated inducing commits. Finally, we present the results of manual annotation.

4.1 Dataset

To investigate whether our proposed approach can detect inducing commits for vulnerabilities and refine vulnerable versions effectively, we select two datasets of vulnerability-fixing commits containing projects written in two programming languages, i.e., C/C++ [30] and Java [43], respectively. Table 1 presents the statistics of the vulnerabilities and the corresponding fixing commits in the two datasets. In this table, #Vul and #VFC are the number of the vulnerabilities and the fixing commits, respectively. Notice that some vulnerabilities have multiple fixing commits. We also count the vulnerability fixing commits that have no deleted lines

Table 1: The statistics of the vulnerabilities and the corresponding fixing commits in two dataset.

Language	Project	#Vul	#VFC	#ZERO	#SMALL (NO_CVE_ID)	#LARGE
C/C++ [30]	FFmpeg	908	908	234	583 (435)	91
	ImageMagick	756	748	207	423 (262)	118
	linux-kernel	1,716	1,841	461	887 (1)	493
	OpenSSL	132	126	30	53 (0)	43
	php-src	426	414	98	223 (42)	93
	Total	3,938	4,037	1,030	2,169 (740)	838
Java [43]	apache/tomcat	69	98	16	32 (0)	50
	apache/struts	44	76	6	39 (0)	21
	apache/tomcat70[1]	35	59	n.a.	n.a.	n.a.
	jenkinsci/jenkins	34	43	7	27 (0)	9
	spring-projects/ spring-framework	24	43	4	17 (1)	32
	...(more 199 projects)					
	Total	624	1,282	172	387 (9)	557

[1] Since the repository of Apache Tomcat70 has been removed on GitHub, we cannot get the number of deleted lines in VFCs and set the values to *n.a.*.

(#ZERO), larger than zero and less than or equal to five deleted lines (#SMALL), and large than five deleted lines (#LARGE), respectively.

The dataset with C/C++ projects is collected by Liu et al. [30]. There are five popular C/C++ projects, i.e., FFmpeg (a multimedia framework), ImageMagick (a representative raster/vector image file processing software suite), OpenSSL (an implementation of secure communication protocols), PHP-SRC (the official interpreter for PHP language), and Linux kernel (one of the most widely used operating systems). They collect a large number of vulnerabilities with fixing commits by utilizing automated crawlers and spending months of manual effort. There are 3,938 vulnerabilities with 4,037 fixing commits in this dataset.

The dataset with Java projects is collected by Ponta et al. [43]. They construct this dataset by collecting publicly disclosed vulnerabilities affecting more than 200 Java open source projects used in SAP products or internal tools, including Apache Tomcat and Struts, Jenkins (an automation server facilitating continuous integration and continuous delivery), and Spring framework. In total, there are 624 vulnerabilities with 1,282 fixing commits in this dataset.

We have many popular projects with two programming languages that span a wide range of functionalities based on the above two datasets. Thus, we believe that various types of vulnerabilities and the corresponding fixing commits have been covered.

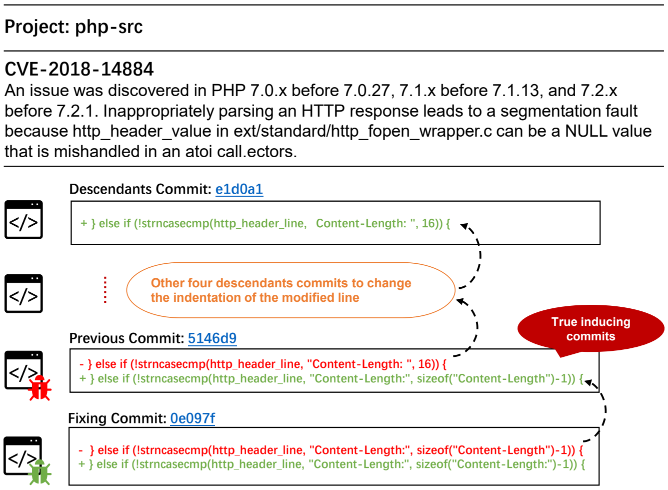
4.2 Manual Annotation for Vulnerability-Inducing Commits

To annotate inducing commits for vulnerabilities, we follow the model proposed by Rodríguez-Pérez et al. [44], which is designed to identify how bugs are introduced in software components. Given a bug-fixing commit, Rodríguez-Pérez et al. check the previous commits and the descendants commits of each line in the bug-fixing commit until identifying the first-failing change. They check whether a bug is present in a certain snapshot using a "perfect test". Since there is no such a perfect test in practice, they use a mentally designed test as a proxy of the perfect test based on bug reports, comments, and discussions from developers.

We use a similar approach to annotate the inducing commits for vulnerabilities. Following Rodríguez-Pérez et al.'s idea of "perfect test", we leverage the related information of a vulnerability (e.g., the CVE description, fixing commits, and commit messages) to determine whether it exists on a snapshot (referred to as a commit on a software system). To help annotators go further back and quickly check previous commits and descendants commits, we develop a web application that integrates the `git blame` tool. For each modified line in a vulnerability fixing commit, we first identify its previous commit to check whether the vulnerability exists in the snapshot, then go further back recursively to analyze descendants commits.

Thus, we might get multiple inducing commits when there are more than one modified line of code. However, according to the model of Rodríguez-Pérez et al. [44], there is just one change that introduced a vulnerability. Thus, we consider the earliest one in the annotated inducing commits as the true inducing commits because we assume all the modified lines of code in the fixing commits are vulnerable and the earliest inducing commits that contain the vulnerable lines of code are more likely to be vulnerable.

To show the annotation process, we can refer to the second motivating example (as shown in Figure 2). We also find that only four vulnerabilities (see Table 2) are introduced by the previous commits but not the descendants commits. For example, CVE-2018-14884 in Fig 5 is not foundational. For example, the commit message of the fixing commit for CVE-2018-14884 in Figure 5 mentions that '*The sizeof()s for Content-Length and Transfer-Encoding were missing the trailing ":"*'. So, we think this vulnerability is introduced by the previous commit because the *sizeof* function in the vulnerable line of code is firstly added in the previous commit.



Project: php-src

CVE-2018-14884

An issue was discovered in PHP 7.0.x before 7.0.27, 7.1.x before 7.1.13, and 7.2.x before 7.2.1. Inappropriately parsing an HTTP response leads to a segmentation fault because http_header_value in ext/standard/http_fopen_wrapper.c can be a NULL value that is mishandled in an atoi call.ectors.

Descendants Commit: e1d0a1

```
+ } else if (!strncasecmp(http_header_line,   Content-Length: ", 16)) {
```

Other four descendants commits to change the indentation of the modified line
True inducing commits

Previous Commit: 5146d9

```
- } else if (!strncasecmp(http_header_line, "Content-Length: ", 16)) {
+ } else if (!strncasecmp(http_header_line, "Content-Length:", sizeof("Content-Length")-1)) {
```

Fixing Commit: 0e097f

```
-  } else if (!strncasecmp(http_header_line, "Content-Length:", sizeof("Content-Length")-1)) {
+ } else if (!strncasecmp(http_header_line, "Content-Length:", sizeof("Content-Length:")-1)) {
```

Figure 5: An example for annotating inducing commit from CVE-2018-14884

The first author and a postdoc who both have more than seven years of programming experience annotated the inducing commits for the selected vulnerabilities (see Section 4.4). We use Fleiss Kappa [18] to measure the agreement between the two annotators. The Kappa value between the two annotators is 0.76, indicating a substantial agreement. The two annotators resolved the disagreements by discussing and reviewing the information of the vulnerabilities.

4.3 Manual Verification for Vulnerable Versions

Our proposed approach can generate a set of version tags affected by a vulnerability. However, we cannot compare them with the textual description about vulnerable versions in CVE automatically because the format of the version tags and the textual description about vulnerable versions vary across projects, see the two motivating examples in Section 2.2. Hence, we want to verify whether the set of versions tags generated by our approach is truly vulnerable versions and consistent with the version description in CVE.

Based on the fixing commits and the inducing commits we annotated, we apply our approach to generate the set of version tags. Then, we manually check whether the boundary versions are vulnerable or not. With the same assumption in the study of Nguyen et al. [38], we think a software version is vulnerable if it contains the vulnerable code; otherwise, it is not vulnerable. Considering the version tags in the first motivating example (see Figure 1), we manually check the boundary versions (e.g., *3.1, 3.2, 3.2.10*, etc.) and find that the vulnerable code exists in these versions. We also check the non-vulnerable boundary versions (e.g., *3.0.12 (the previous version of 3.1), n3.2.11*, etc.) and find that the vulnerable code does not exist in these versions. Finally, we summarize the set of version tags into a textual description and verify whether the vulnerable versions we summarize are consistent with the version information of the CVE vulnerability.

The same two annotators who annotated the inducing commits verified vulnerable versions generated by our approach. For each vulnerability-fixing commit, our approach also generate the vulnerable versions for the previous commit and all the descendants commits. Then, the two annotators checked all the boundary versions manually in the results.

4.4 Vulnerability Selection

Since the manual annotation is time-consuming and labor-intensive, we select a subset of vulnerabilities from the two datasets described in Section 4.1. First, we exclude the vulnerabilities using the following criteria:

- We ignore the vulnerabilities that are not listed in CVE/NVD because we need to leverage the information of CVEs as a proxy of the "perfect test" to annotate the inducing commits. Many vulnerabilities in the two datasets are not listed in CVE/NVD (see Table 1). Liu et al. identified 1,462 security bugs associated with fixing commits and put them into their dataset [30]. Also, in the Java dataset, 29 vulnerabilities do not have a CVE identifier, and 46 vulnerabilities have been assigned a CVE identifier by a CVE numbering authority but are not yet published on NVD [43].
- The vulnerabilities whose fixing commits have no deleted lines of code are excluded. As shown in Table 1, there are 25.5% (1,030/4,037) and 13.4% (172/1,282) of fixing commits that have no deleted lines in the two datasets. One reason is that SZZ algorithms cannot identify inducing commits for those fixing commits since they assume the deleted lines of code in fixing commits are the same as or evolved from the lines of code in the inducing commits. Another reason is that such vulnerabilities are usually foundational (i.e., they exist in their initial versions). All lines of code in the modified file of the versions before the fixing commits can be considered as vulnerable [38]. So, the commit in which the modified file was initialized can be considered as the inducing commit.
- We also exclude the vulnerability whose fixing commits contain large changes, i.e., having more than five lines of code. Changed lines of code in the fixing commits are not necessarily vulnerable, and it is difficult to identify such lines of code in a large change. Besides, the manual annotation for inducing commits with large changes is complicated and time-consuming.
- If the versions affected by a CVE vulnerability cannot be mapped to the version tags in a software project, it would be excluded since we cannot verify whether its vulnerable versions are right or not. For example, all the version tags in the project Apache/-Tomcat are above *7.0.0* but the version information in some CVE vulnerabilities might be lower versions, e.g., *5.x.x* or *6.x.x*.

Among the remaining vulnerabilities, we randomly sample 20 vulnerabilities for each project from the C/C++ dataset (i.e., 100 vulnerabilities in total). For the Java dataset, we label all the remaining vulnerabilities (i.e., 72) covering 41 projects. In total, we annotate 172 CVE vulnerabilities with 188 fixing commits.

4.5 Annotation Results

Table 2 presents the annotation results. In this table, #Descendant-Commit is the number of the vulnerability-fixing commits that have descendants commits, #IntroducedByPC is the number of the

Table 2: The annotation results.

	C/++	Java	Total
#Vulnerability	100	72	172
#FixingCommit	100	88	188
#DescendantCommit	40	52	92
#IntroducedByPC	3	1	4
#VersionInconsistent	56	43	99

vulnerabilities induced by the previous commits if descendants commits exist, and #VersionInconsistent is the number of vulnerabilities whose version information is inconsistent with our verified vulnerable versions. We find that approximately half of vulnerability-fixing commits (92/188) have descendants commits. Among the vulnerabilities that have descendants commits, most of them are introduced by the descendants commits except for four cases. We also find that all the vulnerabilities except for these four cases are foundational, i.e., they exist in their initial version. Additionally, out of 172 vulnerabilities, 99 vulnerabilities' version information in NVD/CVE is inconsistent with the manually verified vulnerable versions. Especially, among the 99 vulnerabilities we find that there are 11 cases in which the vulnerability-free versions are also not affected by other vulnerabilities. But these corresponding projects have a very small number of vulnerabilities. For example, the project Facebook/Buck only has one CVE in total. Many inconsistent cases are that the earlier versions of a software are wrongly marked as vulnerable. This is because NVD applies a conservative rule: "If version X is vulnerable, then so are all its previous versions" (a folk knowledge from [38]).

5 EXPERIMENT RESULTS

In this study, we want to answer the following research questions:

RQ1: Can our approach effectively identify inducing commits for vulnerabilities?

RQ2: Can our approach effectively refine software versions affected by CVE vulnerabilities?

5.1 Evaluation Metrics

Given all the inducing commits we annotated and the set of vulnerability inducing commits detected by SZZ algorithms, we employ two widely metrics, i.e., recall and precision, to measure the accuracy of SZZ algorithms. These two metrics are also used in previous studies [36, 45] to evaluate the performance of SZZ algorithms, which are calculated as follows:

$$recall_c = \frac{|correct_c \cap identified_c|}{|correct_c|}$$

$$precision_c = \frac{|correct_c \cap identified_c|}{|identified_c|}$$

where, $correct_c$ and $identified_c$ are the set of true positive inducing commits and the set of inducing commits detected by SZZ algorithms, respectively. We also compute the F1-score, which is the harmonic mean of precision and recall.

We use similar metrics to evaluate the effectiveness of our approach on refining vulnerable versions affected by CVE vulnerabilities. Given a vulnerability vi, a set of tags of vulnerable versions are detected based on the annotated inducing commits and fixing commits, which are denoted as $correct_{vi}$. For each SZZ algorithm, we also have a set of tags of vulnerable versions based on the inducing commits generated by it, which is denoted as $identified_{vi}$. Then, we compute the recall and precision of an SZZ algorithm on refining vulnerable versions. Here, recall indicates the ratio of the vulnerable versions detected by an SZZ algorithm to the whole true vulnerable versions, while precision indicates the accuracy of an SZZ algorithm on detecting vulnerable versions. Finally, we report the average of the recalls and precisions on the annotated vulnerabilities, see the following formulas:

Table 3: The results of SZZ algorithms on identifying inducing commits for vulnerabilities in C/C++ projects.

	#Identified	Recall	Precision	F1-score
B-SZZ	63	0.676	0.568	0.617
AG-SZZ	67	0.730	0.509	0.600
MA-SZZ	65	0.689	0.481	0.567
V-SZZ	**86**	**0.851**	**0.649**	**0.736**

Table 4: The results of SZZ algorithms on identifying inducing commits for vulnerabilities in Java projects.

	#Identified	Recall	Precision	F1-score
B-SZZ	47	0.687	0.359	0.472
AG-SZZ	51	0.731	**0.521**	0.608
MA-SZZ	54	0.761	0.418	0.540
RA-SZZ	30	0.591	0.433	0.499
V-SZZ	**59**	**0.836**	0.505	**0.630**

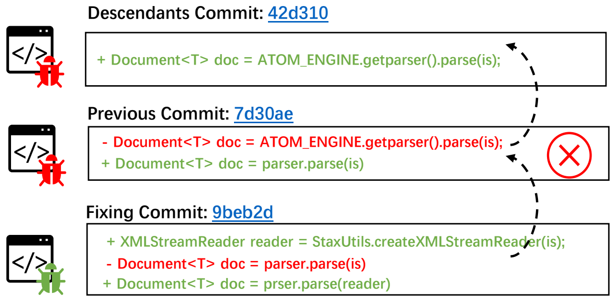
$$average\_recall = \sum_{vi=1}^{N} \frac{|correct_{vi} \cap identified_{vi}|}{|correct_{vi}|} / N$$

$$average\_precision = \sum_{vi=1}^{N} \frac{|correct_{vi} \cap identified_{vi}|}{|identified_{vi}|} / N$$

Where N is the number of vulnerabilities.

5.2 Results

5.2.1 RQ1. Table 3 and 4 present the results of SZZ algorithms on identifying inducing commits for vulnerabilities in C/C++ and Java projects, respectively. We also count the number of cases in which the inducing commits detected by an SZZ algorithm contain the true inducing commit; see the column #Identified in the two tables. Note that RA-SZZ cannot work on many vulnerabilities (i.e., 21), so we exclude these cases in Table 4. Out of 100 and 72 vulnerabilities in C/C++ and Java projects, V-SZZ can identify the true inducing commits for 86 and 59 vulnerabilities, respectively. V-SZZ detects 19 and 6 more inducing commits that the best performing baseline and its F1-score improves the best performing baseline by 19.3% and 3.6% for C/C++ and Java datasheet, respectively. The improvement is considerably good. In terms of recall, precision, and F1-score, V-SZZ achieves the best performance except for the precision on the vulnerabilities in Java projects. And the difference between the precisions of AG-SZZ and V-SZZ is small (0.521 *vs.* 0.505). Thus, we believe that V-SZZ is more effective in identifying inducing commits for vulnerabilities than the previous SZZ algorithms.



```
+ Document<T> doc = ATOM_ENGINE.getparser().parse(is);
```

```
- Document<T> doc = ATOM_ENGINE.getparser().parse(is);
+ Document<T> doc = parser.parse(is)
```

```
+ XMLStreamReader reader = StaxUtils.createXMLStreamReader(is);
- Document<T> doc = parser.parse(is)
+ Document<T> doc = prser.parse(reader)
```

Figure 6: An example of a failing line mapping.

However, V-SZZ still failed to identify inducing commits for a small number of vulnerabilities. One reason is that a few vulnerabilities' true inducing commits are not in the earliest descendants commits, e.g., CVE-2018-14884 in Figure 5. Another reason is that the mapping algorithms in our proposed approach fail to identify the modified lines. For C/C++ projects, the line mapping algorithm based on string similarity does not consider the semantic of the source code. For example, in CVE-2016-2549 from linux-kernel, the modified line `hrtimer_cancel(&stime->hrt);` in the previous commit is wrongly mapped to a deleted line with the same content in another function. For Java projects, the mapping algorithms based on AST [17] also failed in some cases. Figure 6 presents an example from CVE-2016-8739 in the project Apache/CXF. We find that a vulnerable line in the fixing commit cannot be mapped in the previous commit so that our approach cannot identify the descendants commit, which is the true inducing commit.

> Compared to the previous SZZ algorithms, V-SZZ can detect more true inducing commits for the vulnerabilities and outperform them in terms of F1-score.

5.2.2 RQ2. Table 5 and 6 present the results of SZZ algorithms on refining vulnerable versions affected by the vulnerabilities in C/C++ and Java projects, respectively. We also count the number of cases in which the vulnerable versions are correctly inferred based on the inducing commits detected by an SZZ algorithm, denoted as #Correct in the two tables. We find that the number of vulnerabilities with correct vulnerable versions detected by SZZ algorithms increases comparing to the number of vulnerabilities with true inducing commits. This is because the inducing commits wrongly detected by SZZ algorithms might belong to the same version of the true inducing commit. As shown in the table, our approach based on V-SZZ has a much higher recall than the other SZZ algorithms. This indicates that our approach has fewer false negatives (i.e., a version is identified as not vulnerable, but it is vulnerable). Compared to false positives (i.e., a version is identified as vulnerable, but it is not vulnerable), false negatives might result in more potential security risk. Because a conservative action is to upgrade the newest version without the vulnerability for false positives, while if a version is considered not vulnerable, no action might be taken. On the other hand, our approach based on V-SZZ has similar precisions to the other SZZ algorithms. Overall, in terms of F1-score, our approach based on V-SZZ achieves the best performance on the vulnerabilities from both C/C++ and Java projects (0.928 and 0.952, respectively). Furthermore, to measure whether the improvement of our approach based on V-SZZ over the previous SZZ algorithms is statistically significant, we apply Wilcoxon signed-rank test [58] at 95% significance level on F1-scores. We find that the p-values are smaller than 0.05, which indicates the improvement is statistically significant at the confidence level of 95%.

Table 5: The results of SZZ algorithms on identifying vulnerable versions in C/C++ projects.

	#Correct	AvgRecall	AvgPrecision	AvgF1
B-SZZ	69	0.789	**0.960**	0.866
AG-SZZ	73	0.821	0.950	0.881
MA-SZZ	71	0.829	0.930	0.877
V-SZZ	**87**	**0.930**	0.927	**0.928**

Table 6: The results of SZZ algorithms on identifying vulnerable versions in Java projects.

	#Correct	AvgRecall	AvgPrecision	AvgF1
B-SZZ	53	0.875	0.948	0.910
AG-SZZ	55	0.871	0.957	0.912
MA-SZZ	54	0.894	0.934	0.914
RA-SZZ	30	0.640	0.672	0.656
V-SZZ	**59**	**0.941**	**0.963**	**0.952**

> In terms of F1-score, our approach based on V-SZZ can effectively refine version ranges affected by CVE vulnerabilities and achieve higher performance than those based on the previous SZZ algorithms.

6 DISCUSSION

6.1 The Impact of Duplicated Changes

The second step of our approach is to detect commits that have the duplicated changes of the fixing commits and inducing commits. The novelty contribution of this step is minor, but it has a significant impact on the results of our approach. We find that the patches for approximately half of the vulnerabilities we annotated in the study are copied among different fixing commits (i.e., 48/100 and 37/72 for the C/C++ and Java dataset, respectively). Similarly, we also find that 8 and 11 vulnerabilities whose inducing commits we annotated are duplicated in the other commits for the C/C++ and Java dataset, respectively. To verify the impact of duplicated changes, we apply our approach without detecting duplicated changes on our annotated dataset. We find that the detected vulnerable versions of all vulnerabilities are different from those considering duplicated changes. If missing commits with duplicated changes in the fixing commits, the number of false positives increases. If missing commits with duplicated changes in the inducing commits, the number of false negatives increases. For example, the vulnerability-fixing commit in the first motivating example (see Figure 1) has three cherry-picked commits. If we do not consider these three cherry-picked commits, some non-vulnerable versions (e.g. versions above *3.2.10*, *3.3.7*, and *3.4.2*) are detected as vulnerable. Therefore, we

Table 7: The results of the SZZ algorithms on the common bugs

	B-SZZ	AG-SZZ	MA-SZZ	RA-SZZ	V-SZZ
TP	53	43	47	37	29
FP	26	36	32	42	50

think it is necessary to detect the duplicated changes to get more accurate vulnerable versions.

6.2 V-SZZ for Common Bugs

Previous SZZ algorithms assume that the previous commits introduced the bugs, but they fail to identify inducing commits for many common bugs [45]. Thus, we want to investigate whether some common bugs can be identified by going back to look at the descendants commits. We run V-SZZ and the previous SZZ algorithms on the bug-fixing commits collected by Rosa et al. [45]. They built a "developer-informed" oracle for the evaluation of the SZZ algorithms by identifying bug-fixing commits in which developers explicitly reference the commit(s) that introduced a fixed bug. We select the bug-fixing commits from Java projects in their dataset, which contains 79 bugs.

Table 7 presents the results of the SZZ algorithms. We count a true positive if the set of bug-inducing commits generated by SZZ algorithms contains the true inducing commits; otherwise, we count a false positive. We find that B-SZZ detects the most true positives than the other SZZ algorithms. This might be because some bug-inducing commits are introduced by non-semantic lines. For example, the bug-induing commit for a bug-fixing commit ddbda7 from the project DigitalPebble/storm-crawler is associated with a comment line. On the other hand, V-SZZ detects the least true positives. Besides, we check the difference between the set of inducing commits generated by V-SZZ and the other SZZ algorithms. We find that V-SZZ cannot detect additional bug-inducing commits. But if considering all the previous commits and descendants commits as the bug-inducing commit set, V-SZZ can detect one additional bug-inducing commit (the bug-fixing commit 78d23a in the project adamretter/exist). Therefore, we find that most common bugs are not introduced by the descendants commits, which is different from vulnerabilities.

6.3 V-SZZ for Large Commits

In this study, we only apply V-SZZ on the vulnerability-fixing commits with small size since manual verification is time-consuming and difficult. But we think V-SZZ can work on large commits. Nguyen et al. show that small commits help reduce bias in the approach. This is because large commits might contain noisy (nonessential or trivial) changes. But as V-SZZ detects the earliest commits that modify the vulnerable code, these noisy changes usually introduce false positives (i.e., a version is claimed to be vulnerable while it is not). In practice, false positives can be accepted since it requires developers to upgrade to a new version. For false negatives, developers might ignore it, and the vulnerability is not removed. Additionally, we can identify the root changes to reduce these noisy changes by using some existing approaches [47, 55].

6.4 Threats to Validity

Construct Validity. We follow the model proposed by Rodríguez-Pérez et al. [44], which is based on the idea of "perfect test", to annotate the inducing commits for vulnerabilities. However, we have to create such "perfect test" mentally based on the information provided by CVEs, which might result in threats in the results. To mitigate this threat, two annotators discussed the cases in which we were not sure the true inducing commits. Also, we verified the vulnerable versions generated by our approach manually. To mitigate this threat, we manually checked whether the vulnerable code exists in the boundary versions. Moreover, to decrease the difficulty of the manual annotation, we only focus on the vulnerability-fixing commits with a small number of modifications.

Internal Validity. There might be errors when building the dataset since the two annotators are not the developers of the software in the annotated dataset. Both annotators have seven years programming experience in Java and C/C++ programming. Additionally, two annotators check all the information about a vulnerability independently and discuss if they cannot reach an agreement.

External Validity. One of the threats is the number of vulnerabilities we analyzed in the study. We analyzed 172 vulnerabilities in total. But this number is similar to the number of bugs manually analyzed in the previous studies [11, 23, 44]. Another threat is that there is only two programming languages (C/C++ and Java). A previous survey shows that C/C++ and Java covered approximately 65% vulnerabilities [1]. In the future, we plan to investigate more vulnerabilities with different programming languages.

7 RELATED WORK

In this section, we first present some related work on vulnerabilities analysis, then review the studies about the evaluation and application of SZZ algorithms.

7.1 Vulnerabilities Analysis

There are many studies that analyze different aspects of vulnerabilities [8, 24, 29, 30, 34, 39, 48]. Ozment et al. studied the evolution of vulnerabilities in the OpenBSD operating system, and found that remedying half of the known vulnerabilities cost average 2.6 years for a release [39]. Shahzad et al. conducted an exploratory study of vulnerabilities life cycles and found that the vendors have been becoming more agile in patching the vulnerabilities and the access complexity of vulnerabilities has been increasing [48]. Nappa et al. investigated the patch deployment process in ten client applications, and analyzed when security updates were available to clients and how quickly clients patched [34]. Huang et al. analyzed 131 patches from five open-source project, and showed that there exist cases where patch development was lengthy and error-prone [24]. Li and Paxson conducted a large scale empirical study with more than 3,000 CVE entries and the related fixing commits [29]. They analyzed the duration of the impact of vulnerabilities, the reliability of the fix, and the difference with non-security fixes. Camilo et al. conducted an in-depth analysis of the Chromium project to examine the relationship between bugs and vulnerabilities [8]. They demonstrated that bugs and vulnerabilities are empirically dissimilar groups, prompting us to investigate the inducing commits for vulnerabilities. Liu et al.

analyzed the vulnerability distribution based on a large vulnerability dataset, consisting of all known vulnerabilities associated with five representative open source projects [30].

Many researchers also have investigated how to characterize and predict vulnerable code based on different attributes, such as code [22, 37, 41, 46, 49], commit [32, 33], and human factors [33, 49]. For example, Shin and Williams used code complexity metrics to predict vulnerable files and found that the defect prediction model and the vulnerability prediction model achieve similar performance [49]; Scandariato et al. predicted vulnerable software components by applying text mining on source code [46].

7.2 SZZ Evaluation and Application

To evaluate SZZ and its variants, some previous studies rely on manual analysis of a small sample of SZZ results [12, 27, 51, 59]. But manual analysis by researchers is of high cost and might identify bug introducing commits incorrectly. To build a more accurate ground truth, the "developer-informed" information can be leveraged. For example, Rosa et al. identify bug introducing commits by associating the commit id in the commit messages [45]; Wen et al. collect the bug introducing commits in the bug reports [57]. However, it is difficult to build such dataset that contain enough pairs of fixing commits and introducing commits for vulnerabilities. Therefore, we follow the approach of odríguez-Pérez et al. [44], based on the perfect test idea, to build a dataset linking vulnerabilities fixing commits and introducing commits.

SZZ algorithms have been used in many software engineering researches. For example, in many defect prediction studies [9, 10, 21, 26, 31, 52, 53, 60–62], an important research topic in software engineering, researchers use SZZ algorithms to identify the bug-inducing commits and build a dataset to evaluate their proposed defect prediction models. Some researchers use SZZ algorithms to conduct different kinds of empirical studies, such as code review [4, 28], code smells [40], developer activities [6, 7], technical debt [56], etc. In our study, we evaluate the effectiveness of SZZ algorithms on identifying inducing commits for vulnerabilities and proposed an approach based on an improved SZZ algorithm to refine the software versions affected by CVE vulnerabilities.

8 CONCLUSION & FUTURE WORK

In this paper, we proposed an approach based on an improved SZZ algorithm (V-SZZ) to refine vulnerable versions for vulnerabilities. V-SZZ assumes that the vulnerabilities are often introduced by the commit that modified the vulnerable code in earlier versions. Our approach uses the fixing commits and the inducing commits for a vulnerability to determine its vulnerable versions. Additionally, our approach also considers the duplicated fixing commits and inducing commits, making the results of vulnerable versions more accurate. To evaluate our proposed approach, we manually annotate the true inducing commits and verify the vulnerable versions for 172 vulnerabilities with fixing commits, which are from two publicly available datasets with C/C++ and Java projects. The experiment results show that our proposed approach can identify more vulnerabilities with the true inducing commits and correct vulnerable versions than the previous SZZ algorithms. Our approach has a higher F1-score for identifying inducing commits on both C/C++ and Java projects (0.736 and 0.630) than the previous SZZ algorithms. For refining vulnerable versions, our approach also achieves the best performance on the two datasets in terms of F1-score (0.928 and 0.952). In the future, we plan to analyze more vulnerabilities from projects with different programming languages. We also want to develop a tool based on our approach to help developers refine the versions affected by CVE vulnerabilities.

DATA AVAILABILITY

We provide a replication package of our dataset and proposed approach, which is available at https://github.com/baolingfeng/V-SZZ.

ACKNOWLEDGEMENT

This research/project is supported by the National Science Foundation of China (No. 62141222, No. U20A20173 and No. 6190234).

REFERENCES

[1] 2017. How do the top programming languages measure up when it comes to security? https://www.whitesourcesoftware.com/most-secure-programming-languages. Accessed: 2021-08-23.

[2] Muhammad Asaduzzaman, Michael C Bullock, Chanchal K Roy, and Kevin A Schneider. 2012. Bug introducing changes: A case study with android. In *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. IEEE, 116–119.

[3] Gabriele Bavota, Bernardino De Carluccio, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, and Orazio Strollo. 2012. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 104–113.

[4] Gabriele Bavota and Barbara Russo. 2015. Four eyes are better than two: On the impact of code reviews on software quality. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 81–90.

[5] Mario Luca Bernardi, Gerardo Canfora, Giuseppe A Di Lucca, Massimiliano Di Penta, and Damiano Distante. 2012. Do developers introduce bugs when they do not communicate? the case of eclipse and mozilla. In *2012 16th European Conference on Software Maintenance and Reengineering*. IEEE, 139–148.

[6] Mario Luca Bernardi, Gerardo Canfora, Giuseppe A Di Lucca, Massimiliano Di Penta, and Damiano Distante. 2018. The relation between developersâĂŹ communication and fix-Inducing changes: An empirical study. *Journal of Systems and Software* 140 (2018), 111–125.

[7] Bora Çaglayan and Ayşe Başar Bener. 2016. Effect of developer collaboration activity on software quality in two large scale projects. *Journal of Systems and Software* 118 (2016), 288–296.

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

[9] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2013. Multi-objective cross-project defect prediction. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 252–261.

[10] Gerardo Canfora, Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. 2015. Defect prediction as a multiobjective optimization problem. *Software Testing, Verification and Reliability* 25, 4 (2015), 426–459.

[11] Daniel Alencar Da Costa, Shane McIntosh, Weiyi Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E Hassan. 2016. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering* 43, 7 (2016), 641–657.

[12] 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.

[13] Georg Dotzler and Michael Philippsen. 2016. Move-optimized source code tree differencing. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 660–671.

[14] Jordan Ell. 2013. Identifying failure inducing developer pairs within developer networks. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 1471–1473.

[15] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 313–324.

[16] Yuanrui Fan, D Alencar da Costa, D Lo, AE Hassan, and L Shanping. 2020. The impact of mislabeled changes by szz on just-in-time defect prediction. *IEEE Transactions on Software Engineering* (2020).

[17] 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 *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1174–1185.

[18] Joseph L Fleiss. 1971. Measuring nominal scale agreement among many raters. *Psychological bulletin* 76, 5 (1971), 378.

[19] 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.

[20] Veit Frick, Thomas Grassauer, Fabian Beck, and Martin Pinzger. 2018. Generating accurate and compact edit scripts using tree differencing. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 264–274.

[21] 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*. 172–181.

[22] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. 2008. Prioritizing software security fortification throughcode-level metrics. In *Proceedings of the 4th ACM workshop on Quality of protection*. 31–38.

[23] 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*. 99–108.

[24] Zhen Huang, Mariana DAngelo, Dhaval Miyani, and David Lie. 2016. Talos: Neutralizing vulnerabilities with security workarounds for rapid response. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 618–635.

[25] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. 2012. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering* 39, 6 (2012), 757–773.

[26] Sunghun Kim, Hongyu Zhang, Rongxin Wu, and Liang Gong. 2011. Dealing with noise in defect prediction. In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 481–490.

[27] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. 2006. Automatic identification of bug-introducing changes. In *21st IEEE/ACM international conference on automated software engineering (ASE'06)*. IEEE, 81–90.

[28] Oleksii Kononenko, Olga Baysal, Latifa Guerrouj, Yaxin Cao, and Michael W Godfrey. 2015. Investigating code review quality: Do people and participation matter?. In *2015 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 111–120.

[29] Frank Li and Vern Paxson. 2017. A large-scale empirical study of security patches. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2201–2215.

[30] Bingchang Liu, Guozhu Meng, Wei Zou, Qi Gong, Feng Li, Min Lin, Dandan Sun, Wei Huo, and Chao Zhang. 2020. A large-scale empirical study on vulnerability distribution within projects and the lessons learned. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1547–1559.

[31] Shane McIntosh and Yasutaka Kamei. 2017. Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction. *IEEE Transactions on Software Engineering* 44, 5 (2017), 412–428.

[32] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodriguez 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*. IEEE, 65–74.

[33] Andrew Meneely and Laurie Williams. 2009. Secure open source collaboration: an empirical study of linus' law. In *Proceedings of the 16th ACM conference on Computer and communications security*. 453–462.

[34] Antonio Nappa, Richard Johnson, Leyla Bilge, Juan Caballero, and Tudor Dumitras. 2015. The attack of the clones: A study of the impact of shared code on vulnerability patching. In *2015 IEEE symposium on security and privacy*. IEEE, 692–708.

[35] 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)*. IEEE, 380–390.

[36] 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)*. IEEE, 1–12.

[37] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting vulnerable software components. In *Proceedings of the 14th ACM conference on Computer and communications security*. 529–540.

[38] 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 (2016), 2268–2297.

[39] Andy Ozment and Stuart E Schechter. 2006. Milk or wine: does software security improve with age?. In *USENIX Security Symposium*, Vol. 6.

[40] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. 2018. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering* 23, 3 (2018), 1188–1221.

[41] 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*. 426–437.

[42] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 449–460.

[43] Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 383–387.

[44] 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 (2020), 1294–1340.

[45] Giovanni Rosa, Luca Pascarella, Simone Scalabrino, Rosalia Tufano, Gabriele Bavota, Michele Lanza, and Rocco Oliveto. 2021. Evaluating SZZ Implementations Through a Developer-informed Oracle. (2021), 436–447.

[46] 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.

[47] Adriana Sejfia, Yixue Zhao, and Nenad Medvidović. 2021. Identifying casualty changes in software patches. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 304–315.

[48] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X Liu. 2012. A large scale exploratory analysis of software vulnerability life cycles. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 771–781.

[49] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. 2010. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE transactions on software engineering* 37, 6 (2010), 772–787.

[50] 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)*. IEEE, 269–279.

[51] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes? *ACM sigsoft software engineering notes* 30, 4 (2005), 1–5.

[52] Qinbao Song, Yuchen Guo, and Martin Shepperd. 2018. A comprehensive investigation of the role of imbalanced learning for software defect prediction. *IEEE Transactions on Software Engineering* 45, 12 (2018), 1253–1269.

[53] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 2. IEEE, 99–108.

[54] 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)*. IEEE, 483–494.

[55] Min Wang, Zeqi Lin, Yanzhen Zou, and Bing Xie. 2019. Cora: Decomposing and describing tangled code changes for reviewer. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1050–1061.

[56] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the impact of self-admitted technical debt on software quality. In *2016 IEEE 23Rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 179–188.

[57] Ming Wen, Rongxin Wu, Yepang Liu, Yongqiang Tian, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. 2019. Exploring and exploiting the correlations between bug-inducing and bug-fixing commits. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 326–337.

[58] Frank Wilcoxon. 1992. Individual comparisons by ranking methods. In *Breakthroughs in statistics*. Springer, 196–202.

[59] 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*. 32–36.

[60] Xin Xia, David Lo, Sinno Jialin Pan, Nachiappan Nagappan, and Xinyu Wang. 2016. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering* 42, 10 (2016), 977–998.

[61] Xinli Yang, David Lo, Xin Xia, and Jianling Sun. 2017. TLEL: A two-layer ensemble learning approach for just-in-time defect prediction. *Information and Software*

Technology 87 (2017), 206–220.

[62] Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *2015 IEEE International Conference on Software Quality, Reliability and Security*. IEEE, 17–26.