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

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

\(^1\)https://snyk.io/
\(^2\)https://www.whitesourcesoftware.com/
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 days³). 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 Rodriguez-Pérez et al. [44], which is designed for bug-inducing commit annotation. We choose 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.
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 Sliewerski 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 \texttt{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 ReDiff [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.

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\(^1\). 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.

\(^1\)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 ’/fileservlet/./admin/’. The added API ’FileSystemService.getDefault().getPath(String, normalize)’ in the fixing commit removes redundant elements in a path including any ’/’ or ’directory/’ occurrences.
CVE-2015-1830
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 workflows of different versions in a project. To apply the change in the previous commit or descendants commits, we continue going back by running 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 to refine the version information.

Given each modified line in a fixing commit, V-SZZ first identifies thelining algorithm identifies the line as a new line, the source code file of the previous commits.

Given each modified line in a fixing commit, V-SZZ first identifies thecommit 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 Rodriguez-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 theprevious 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 729b47 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

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 `n.3.1` 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.

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

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 larger 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.
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 atoll call. 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, #DescendantCommit is the number of the vulnerability-fixing commits that have descendants commits, #IntroducedByPC is the number of the
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:

\[
\text{recall}_v = \frac{|\text{correct}_v \cap \text{identified}_v|}{|\text{correct}_v|}
\]

\[
\text{precision}_v = \frac{|\text{correct}_v \cap \text{identified}_v|}{|\text{identified}_v|}
\]

where, \(\text{correct}_v\) and \(\text{identified}_v\) 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 \(v_i\), a set of tags of vulnerable versions are detected based on the annotated inducing commits and fixing commits, which are denoted as \(\text{correct}_v\). 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 \(\text{identified}_v\). 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:

\[
\text{average}\_\text{recall} = \frac{\sum_{i=1}^{N} |\text{correct}_v \cap \text{identified}_v|}{|\text{correct}_v|}/N
\]

\[
\text{average}\_\text{precision} = \frac{\sum_{i=1}^{N} |\text{correct}_v \cap \text{identified}_v|}{|\text{identified}_v|}/N
\]

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 datasets, 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.

<table>
<thead>
<tr>
<th>#Identified</th>
<th>Recall</th>
<th>Precision</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-SZZ</td>
<td>63</td>
<td>0.676</td>
<td>0.568</td>
</tr>
<tr>
<td>AG-SZZ</td>
<td>67</td>
<td>0.730</td>
<td>0.509</td>
</tr>
<tr>
<td>MA-SZZ</td>
<td>65</td>
<td>0.689</td>
<td>0.481</td>
</tr>
<tr>
<td>V-SZZ</td>
<td>86</td>
<td>0.851</td>
<td>0.649</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>#Identified</th>
<th>Recall</th>
<th>Precision</th>
<th>F1-score</th>
</tr>
</thead>
<tbody>
<tr>
<td>B-SZZ</td>
<td>47</td>
<td>0.687</td>
<td>0.359</td>
</tr>
<tr>
<td>AG-SZZ</td>
<td>51</td>
<td>0.731</td>
<td>0.521</td>
</tr>
<tr>
<td>MA-SZZ</td>
<td>54</td>
<td>0.761</td>
<td>0.418</td>
</tr>
<tr>
<td>RA-SZZ</td>
<td>30</td>
<td>0.591</td>
<td>0.433</td>
</tr>
<tr>
<td>V-SZZ</td>
<td>59</td>
<td>0.836</td>
<td>0.505</td>
</tr>
</tbody>
</table>
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(&time->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%.

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
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-inducing commit for a bug-fixing commit dbdbda7 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/exists). 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.

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

<table>
<thead>
<tr>
<th>B-SZZ</th>
<th>AG-SZZ</th>
<th>MA-SZZ</th>
<th>RA-SZZ</th>
<th>V-SZZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td>53</td>
<td>43</td>
<td>47</td>
<td>37</td>
</tr>
<tr>
<td>FP</td>
<td>26</td>
<td>36</td>
<td>32</td>
<td>42</td>
</tr>
</tbody>
</table>
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 Rodríguez-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-introducing 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


