VERJava: Vulnerable Version Identification for Java OSS with a Two-Stage Analysis

Qing Sun[1,2], Lili Xu[1,2,*], Yang Xiao[1,2], Feng Li[1,2], He Su[1,2], Yiming Liu[1,2], Hongyun Huang[3,*], Wei Huo[1,2]
[1]Institute of Information Engineering, Chinese Academy of Sciences, China
[2]School of Cyber Security,University of Chinese Academy of Sciences, China
[3]Zhejiang Sci-Tech University, China

***Abstract*—The software version information affected by the CVEs (Common Vulnerabilities and Exposures) provided by the National Vulnerability Database (NVD) is not always accurate. This could seriously mislead the repair priority for software users, and greatly hinder the work of security researchers. Bao et al. improved the well-known Sliwerski-Zimmermann-Zeller (SZZ) algorithm for vulnerabilities (called V-SZZ) to precisely refine vulnerable software versions. But V-SZZ only focuses on those CVEs of which patches only have deleted lines.**

In this study, we target Java Open Source Software (OSS) by virtue of its pervasiveness and ubiquitousness. Due to Java's object-oriented characteristic, a single security patch often involves modifications of multiple functions. Existing patch code similarity analysis does not consider patch existence from the point of view of an entire patch, which would generate too many false positives for Java CVEs. In this work, we address these limitations by introducing a two-stage approach named VERJava, to systematically assess vulnerable versions for a target vulnerability in Java OSS. Specifically, vulnerable versions are calculated respectively at a function level and an entire patch level, then the results are synthesized to decide the final vulnerable versions. For evaluation, we manually annotated the vulnerable versions of 167 real CVEs from seven popular Java open source projects. The result shows that VERJava achieves the precision of 90.7% on average, significantly outperforming the state-of-the-art work V-SZZ. Furthermore, our study reveals some interesting findings that have not yet been discussed.

***Index Terms*—patch analysis, vulnerability, Java OSS, vulnerable version identification, code similarity**

I. Introduction

Nowadays, it is very common for software developers to use Open Source Software (OSS for short). According to the 2021 GitHub statistical report [1], more than 61 million new repositories were created in the last year. Besides, a recent study [2] shows OSS usually contains a large number of vulnerabilities. To patch the vulnerabilities in OSS promptly, software maintainers often go to check the vulnerable versions of CVEs [3] given by NVD [4]. However, studies by Nguyen et al. [5], VIEM [6] and Bao et al. [7] show that CVE's version information in NVD is generally inaccurate. This could seriously mislead the repair priority for software maintainers, and greatly hinder the work of security researchers [8] [9] [10]. Therefore, it is of great importance to accurately identify the exact vulnerable versions affected by CVEs.

Identifying whether a software version is vulnerable to a known vulnerability in CVEs can be done by dynamically triggering the vulnerability. SemFuzz [11] and VULSCOPE [12] use fuzzing-based method for triggering. They first need a public PoC of CVE's reference version, and mutate certain data and control flows to generate triggerable PoCs for more versions. Dynamical approach introduces no false positives, however, it cannot ensure the vulnerability existence for the versions that are failed to construct a mutated PoC, which means that it may have false negatives. Moreover, this approach requires a runnable software and environment for each version under consideration, for the popular Java project Tomcat [13] which has hundreds of existing versions, the dynamic approach may suffer from scalability issues.

To address this problem in a relatively light-weight way, static analysis approach based on the blame feature of version-control systems also leads an active line of research (e.g., V-SZZ [7], SZZ [14], Alexopoulos et al. [15], OpenSZZ [16]). The Sliwerski-Zimmermann-Zeller (SZZ) algorithm is a well-known algorithm for identifying bug-inducing commits. A recent work V-SZZ [7] has improved it for identifying vulnerability-inducing commits. Essentially, V-SZZ uses the git blame command for the deleted lines in the patch to locate the earliest commit that introduce the deleted lines. The versions between the inducing commit and the fixing commit are considered to be the range of vulnerable versions. Such methods cannot handle the patches where the vulnerabilities are fixed by adding checks, namely, no deletion exists. Besides, this kind of algorithm heavily relies on the accuracy and completeness of repair commit IDs.

Another instinctive reaction for identifying vulnerable version is to use the static approach based on the code similarity analysis between a target version and the patch code. There are a wealth of work on patch code analysis, while, aiming at the search for similar vulnerable codes, e.g., MVP [17], VCCFinder [18], ReDeBug [19] and VUDDY [20]. They focus on C and C++ projects and mainly identify the similar vulnerable code at the function level or at the patch-fragment (i.e., lines of consecutive modifications) level. In this work, we target Java OSS by virtue of its pervasiveness and ubiquitousness. According to our empirical analysis, security patches of the CVEs for Java OSS often involve multiple functions, specifically, of which portion is 61.7% in our collected dataset. Whether a version is vulnerable should be decided by considering the overall status of all patch functions, namely, at the entire patch level. Thus, existing patch analysis works would introduce

* Lili Xu and Hongyun Huang are the corresponding authors.

many false positives by considering only fine-grained code unit. Moreover, the task of vulnerable version identification is different from the one of similar vulnerability searching. It is not clear whether some heavy static analysis methods like code normalization, forward/backward slicing often utilized in similar vulnerability searching are still needed in the task of vulnerable version identification.

To address the above questions, we first conduct a large-scale manual annotation and empirical analysis on Java patches and OSS versions, summarize the statistics with serveral observations. An important one is that thanks to Java's object-oriented characteristic, a single security patch often involves modifications of multiple functions. Under the guidance of the observations, we propose a light-weight and accurate approach "VERJava" for vulnerable version identification at the entire patch level. By "light-weight", we mean that no heavy compilation tool chain is involved to perform complex code slicing, thus VERJava can be easily deployed. By "accurate", we mean that the evaluation performed on the labelled dataset shows a rather satisfying precision of VERJava.

The key rationale behind our idea is that: since a Java patch often involve multiple functions, and software maintainers naturally apply an entire patch when fixing a vulnerability, if the majority of functions are similar to the post-patch code, the corresponding version is more likely to be patched. Specifically, we use a two-stage analysis. In the first stage, we perform a function level vulnerability existence analysis, namely, the versions where an involved function is un-patched are calculated using code similarity analysis. In the second stage, we perform an entire-patch level vulnerability existence analysis, namely, whether a target version is vulnerable is deduced from the proportion of patched functions.

To the best of our knowledge, this is the first work that considers the special security patch characteristic for the object-oriented language Java and identifies vulnerable versions at the entire patch level. The contributions of this paper are summarized as follows:

- For empirical analysis and evaluation, we manually annotated the vulnerable versions of 167 real CVEs from seven popular Java open source projects. A package of the detailed dataset is made publicly available[1].
- We propose a two-stage, i.e., function-level and patch-level, approach, named VERJava, to systematically assess vulnerable versions for a target vulnerability in Java OSS. Evaluation shows that the identification precision of VERJava achieves 90.7% on average, significantly outperforming the state-of-the-art work V-SZZ.
- Some interesting findings concerning the number of functions in a Java security patch and the object-oriented patch feature are summarized. They are helpful in inspiring the work in this paper, and hopefully be enlightening as well in works like similar vulnerability searching.

The remainder of this paper is organized as follows. In Sec. II, we use two examples to illustrate the limitation of existing approaches and motivate the idea of VERJava. Sec. III describes the data collection and manual vulnerability annotation efforts. Sec. IV outlines the design of VERJava. Subsequently, Sec. V presents the evaluation and Sec. VI discusses the threat to validity and possible future improvement. Related work is discussed in Sec. VII and Sec. VIII concludes the paper.

[1]https://github.com/sunSUNQ/VERJava_Dataset

```
--- 7.0.x/.../authenticator/FormAuthenticator.java
    (revision 1408043)
@@ -404,6 +405,15 @@
...
+   Session session = request.getSessionInternal(false);
+   if (session != null) {
+     Manager manager =
    request.getContext().getManager();
+     manager.changeSessionId(session);
+     request.changeSessionId(session.getId());
+   }
...
```

Listing 1: The patch of CVE-2013-2067 in Tomcat.

```
// commit 2235894210c75f624a3d0cd60bfb0434a20a18bf
--- /src/.../impl/SubTypeValidator.java
@@ -0,0 +1,98 @@
...
+   if (full.startsWith(PREFIX_STRING)) {
+      for (Class<?> cls = raw; cls != Object.class;
    cls = cls.getSuperclass()) {
+          String name = cls.getSimpleName();
+          if ("AbstractPointcutAdvisor".equals(name)||
    "AbstractApplicationContext.equals".equals(name)){
+              break;
+          }
+      }
+   }
// SubTypeValidator.java in version 2.9.10
    if (raw.isInterface()) {
        ... ;
    } else if (full.startsWith(PREFIX_SPRING)) {
        for (Class<?> cls = raw; (cls != null) && (cls
              != Object.class); cls =cls.getSuperclass()){
            String name = cls.getSimpleName();
            if ("AbstractPointcutAdvisor".equals(name)||
                "AbstractApplicationContext".equals(name)){
                break main_check;
            }
        }
     } else if (full.startsWith(PREFIX_C3P0)) {
         if (full.endsWith("DataSource")) {
             break main_check;
         }
     }
```

Listing 2: CVE-2017-17485's patch in Jackson-databind.

II. Challenges And Insights

We choose the security patches of Tomcat's CVE-2013-2067 [21], Jackson-databind's CVE-2017-17485 [22] and Tomcat's CVE-2011-0013 [23] as examples to illustrate the limitations of existing methods, challenges, and our insights into addressing these challenges.

A. Limitations of Existing Work

Vulnerability patches can be fixed in a variety of ways, such as removing or correcting the code that caused the vulnerability, or adding a checking code. We studied the patch codes of 167 real CVEs and found that 61 of them are fixed by adding checks, like the example in Listing 1. It shows the

```
1  --- 6.0.x/.../manager/HTMLManagerServlet.java (revision
       1057269)
2  @@ -407,10 +407,11 @@
3                  args = new Object[7];
4                  args[0] = URL_ENCODER.encode(displayPath);
5  -               args[1] = displayPath;
6  -               args[2] = context.getDisplayName();
7  -               if (args[2] == null) {
8  +               args[1] = RequestUtil.filter(displayPath);
9  +               if (context.getDisplayName() == null) {
10                     args[2] = " ";
11 +               } else {
12 +                   args[2] =
       RequestUtil.filter(context.getDisplayName());
13                 }
14 --- 6.0.x/.../manager/StatusTransformer.java (revision
       1057269)
15 @@ -575,7 +575,7 @@
16 ...
17 -               writer.print(webModuleName);
18 +               writer.print(filter(webModuleName));
19 ...
20 @@ -650,7 +650,7 @@
21 ...
22 -           writer.print(name);
23 +           writer.print(filter(name));
24 ...
25 @@ -778,11 +778,11 @@
26 ...
27 -           writer.print(servletName);
28 +           writer.print(filter(servletName));
29 ...
30                 for (int i = 0; i < mappings.length; i++) {
31 -                   writer.print(mappings[i]);
32 +                   writer.print(filter(mappings[i]));
33 ...
```

Listing 3: The patch snippets of CVE-2011-0013 in Tomcat.

CVE-2013-2067 patch for Tomcat. V-SZZ and other methods that are based on the blame feature of version-control systems cannot handle such patches (marked as *Limitation 1*). This paper aims to deal with multi-kind patches, especially, cover the case of patches fixed by adding a checking code.

Secondly, due to the ubiquitous situation of multi-branch management in Java, different commit IDs are usually used when submitting commits. In order to accurately identify vulnerability versions on each branch, the method based on git blame needs to obtain the corresponding repair commit of the vulnerability on all branches, and the collection process of this information could be labor-intensive. In the analysis of 7 popular Java OSS, it is found that only Tomcat maintains individual patches for each branch when publishing CVE, other projects basically only provide one patch, and it takes extra human resources to collect all branches' security fixes from issue history information (marked as *Limitation* 2). While our method is essentially based on code similarity analysis. Specifically, target versions are analyzed according to the patch's content. Thus even the patch for only one branch is known, we can accurately identify the cases where similar fixes are applied on different branches. Listing 2 shows the patch of CVE-2017-17485 in *validateSubType()* and the same function code in version 2.9.10. In there, we only got the patch for Jackson-databind 2.9.x, but we can detect that the vulnerability exists in other early branches simultaneously (like branch 2.6.x, 2.7.x, 2.8.x, etc.), and V-SZZ will cause false negatives. For the case where multiple branch patches can be fetched, we distinguish the patches of each branch, analyze them separately, and conduct a comprehensive evaluation by merging the results of each branch.

B. Challenges

We knew that CVE-2017-17485 was fixed in version 2.9.4 for branch 2.9.x. According to experience, it should not exist in versions 2.9.4 and later. With the version evolution, the code may change. Take the code of *validateSubType()* in version 2.9.10 shown in Listing 2 as an example. The conditional and loop statement has been modified (Line 14-17), and a new conditional statement has been added (Line 23-26), which is different from the patch code (Line 5-12). Simply using added lines to consider the existence of the vulnerability will only match two lines in the patch code, i.e., Line 5 and 7, thus would misjudge version 2.9.10 as vulnerable. It is challenging to consider the difference between target code and post-patch accurately (marked as *Challenge-I: code evolution processing*). This paper aims to comprehensively consider whether the current function is repaired through the similarity analysis between pre-patch and post-patch functions for the samples with code evolutions (seen in IV-B).

Listing 3 shows the patch of CVE-2011-0013 for branch 6.0.x. It involves modifications not only in one function (Line 5-12) but also in four other functions. The idea of patching is to filter the print function's parameters to prevent sensitive information from being leaked. So the modifications in all the four functions are determined to be security fixes by manually annotating. After analyzing an extensive number of patches, we found that Java patches generally involve multiple function locations. How do we comprehensively estimate the version range of a vulnerability when it exists in multiple functions in multiple files (marked as *Challenge-II: multi-location fix handling*)? A patch-level vulnerable version calculation is proposed to handle the multi-location fix case (seen in IV-C).

III. Dataset Construction and Manual Annotation

To investigate the vulnerability fixing patterns of Java OSS, and propose an effective vulnerable version identification method, we first need a ground truth containing vulnerabilities and their affected software version lists. In this section, we describe the selection rules of target projects and CVEs (Sec. III-A), and then we manually annotate the vulnerable versions for each project and each CVE, the annotating principle is explained in Sec. III-B. Finally, two findings are given based on the statistics of labeled data (Sec. III-C).

A. Dataset Creation

The selection rules of target projects and known vulnerabilities to be manually annotated are presented below.

*1) **Target project selection criteria**:* To explore the characteristics of Java vulnerability patches, we first referred to the dataset of Java projects collected by Ponta et al. [24], which is also referred to in V-SZZ. In total, Ponta's dataset

TABLE I: Number of CVEs, Commits, Functions and Versions we collected per project.

Target	Stars	CVEs	Commits	Functions	Versions
Tomcat	6k	45	125	430	144
Jackson-databind	3k	62	68	104	46
Struts	1.1k	14	18	114	41
Jenkins	18.7k	16	21	61	72
Liferay-portal	1.9k	20	99	200	13
Spring-security	6.7k	3	3	44	15
Spring-framework	47.1k	7	10	107	42
Total	-	167	344	1060	373

has 624 CVEs from more than 200 Java OSS. The TOP-6 projects w.r.t. the number of CVEs of which patches can be obtained are selected. Moreover, we searched on NVD details and selected Liferay-portal which has the most CVEs excluding the pre-selected TOP-6 projects. Due to the manual annotation overhead, we decided to first analyze the 7 projects, which are Tomcat [13], Jenkins [25], Struts [26], Jackson-databind [27], Spring-framework [28], Spring-security [29], Liferay portal [30].

2) ***CVE selection criteria:*** After the target projects have been determined, the selected CVEs are preferably to cover the following conditions as much as possible: (1) The most important thing is that the patch file for the selected CVE can be directly obtained through NVD's CVE description page, or indirectly obtained through cross-searching between CVE description page and the project's issue list; (2) The selected CVEs cover a wide period of a project's lifetime; (3) They have different types of vulnerabilities (like path traversal, XSS, information disclosure, etc.); (4) They contain various numbers of patch lines, including cases where more than one hundred lines of code are modified, and also cases where only one line is modified in a vulnerability fixing patch(e.g., CVE-2008-0128's patch modifies only one line).

Eventually 167 CVEs are collected, and the details of the dataset are shown in Table I. The first and the second columns give the target OSS name and its number of github stars. The third column gives the total number of CVEs returned from a search in NVD. The fourth column gives the corresponding sum of the commit numbers for those CVEs. Note that for the project Tomcat, the total commit number (i.e., 125) is nearly three times of the CVE number (i.e., 45), the reason is that Tomcat separately manages vulnerability fixing commits for each individual branch; for the project Liferay-portal, the total commit number (i.e., 99) is nearly 5 times of the CVE number (i.e., 20), the reason is that this project tends to repeatedly submit patches for the same CVE. The fifth column shows the number of functions involved in those commits. And the last column displays the number of versions selected from these projects, which cover mainstream versions and also a wide range of the project's lifetime.

B. Manual Annotation Principles

To link each vulnerability with its affected versions, we perform static and manual auditing, without using dynamic ways to trigger it. Intuitively speaking, we think a software version is vulnerable if it contains the vulnerable code and does not contain the fixing code. Undoubtedly, objective annotation principles are essential for ensuring the accuracy of the subsequent experiment. We explain our manual annotation principles by considering the following three situations.

1) ***Situation I: The function or file where the patch fragment is located does not exist in the target version's code:*** In that case, we think that the absence of patch function or file tends to show the absence of the vulnerability, thus annotate the target version as non-vulnerable. For example, all the functions involved in the fixing commit of CVE-2017-9805 do not exist before the version 2.1.0 of Struts. So we think before 2.1.0, Struts is not affected by CVE-2017-9805.

2) ***Situation II: The target code is the same as the pre-patch or the post-patch code:*** If the target code's counterpart is identical to the pre-patch code, it is annotated as vulnerable, and if the snippet is identical to the post-patch code, it is marked as fixed. For example, the pre-patch code for CVE-2020-35490 strictly matches the code for versions 2.9.5 to 2.10.1 of Jackson-databind. So we label the versions 2.9.5-2.10.1 of Jackson-databind as vulnerable.

3) ***Situation III: The target code is different from the pre-patch and the post-patch code:*** Differences include all kinds of changes in the target code, such as line missing, variable or method renaming, operator and control flow changes, etc. The example in Listing 2 displays that CVE-2017-17485's patch code is different from the counterpart in 2.9.10.

For this situation, we use the manual auditing expertise to determine whether the vulnerability is fixed in the current version through a comprehensive analysis of the vulnerability description, the patch code and the target code. The annotating principle in this situation is clarified in the following. We divide fixing patches into two categories, of which the first has both added and deleted lines, and the second has only one type of line modification. The added and deleted lines mentioned below are referred to as those critical code modifications identified by auditing expertise.

For the cases where a vulnerability fixing patch contains both added and deleted lines. We use two code-reviewing rules: (1) The existence of the added line in a target code takes precedence over the deleted lines. The reason is that empirical experience formed during manual labeling process shows that deleted lines may often appear in multiple places in the same function, while added lines recur less frequently. Thus, if some key added lines identified by auditing expertise exist in the target code, the vulnerability is considered to be fixed in the target version, regardless of whether the deleted lines exist or not. (2) When the added lines do not exist in the target code but the deleted lines exist, it is assumed that there is a vulnerability in the target code.

For the cases where a vulnerability fixing patch contains only one type of line modification. They can be further divided into two categories: (1) There are no deleted lines in the patch (the vulnerability is fixed by adding checks, functions, classes, files, etc.). These cases can be analyzed only through the information given by the added lines. Essentially,

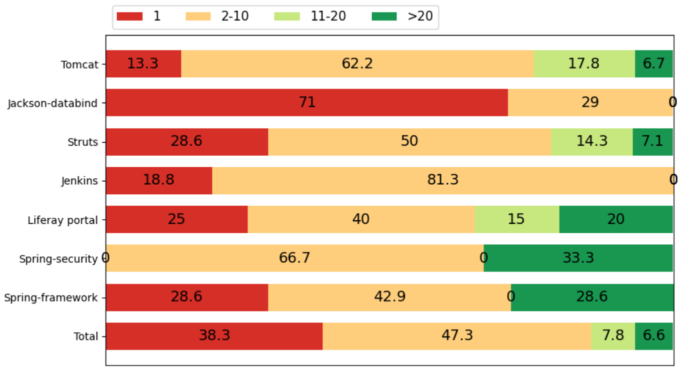

Fig. 1: Statistics of the proportion of the number of repaired functions in patches.

TABLE II: The data set we selected to analyze the actual proportion of annotation situations.

Target	CVE Num	Version Num
Tomcat	40	30
Jackson-databind	62	17
Struts	13	13
Jenkins	16	20
Liferay portal	19	13
Spring-security	3	10
Spring-framework	5	15
Total	**158**	**118**

if the target code do not have the added lines, it is considered as vulnerable, otherwise, as secure. (2) There are no added lines in the patch. Whether the target code is vulnerable or not can only be determined by the existence of the deleted lines. Basically, if the target code contains the deleted lines, it is annotated as vulnerable, otherwise, as secure.

C. Statistics and Findings

Finding 1: Vulnerability patches usually involve multiple functions in Java CVEs, of which percentage is about 61.7% in our dataset.

Fig. 1 shows the percentage distribution of the number of functions involved in patches in each project. The red, yellow, light green and dark green colors represent the percentages of the CVEs where one function, 2-10, 11-20 and more than 20 functions being modified in their patches, respectively. The last row shows the overall distribution of the number of the fixing functions in the seven projects. The statistics indicate that Java vulnerability patches usually involve multiple function locations. 103 CVEs are multi-function repaired (61.7%), which means considering multi-function changes is necessary for an accurate vulnerable version identification. In particular, there are 10 CVEs with more than 20 repaired functions (6.6%). This could increase the difficulty of identification.

Finding 2: The situation in which the code of each version is different from the pre-patch and post-patch code occupies a small portion in Java OSS, of which percentage is only 8% in our labelled dataset.

To investigate the actual proportion of each annotation situation introduced in Sec. III-B, we perform a statistic analysis

TABLE III: The statistics of annotation situations.

Situation ID	Num	Percent	Merged Num	Percent
Situation I	1895	26.54%	1895	26.54%
Situation II (pre-patch)	2282	31.97%	4673	65.46%
Situation II (post-patch)	2391	33.49%		
Situation III (pre-patch)	60	1.84%	571	8.00%
Situation III (post-patch)	511	7.16%		

on a sub-dataset of which details are shown in Table II. The first column gives the OSS name. The second column gives the total number of CVEs we selected (the CVEs whose numbers of patched functions are not more than 20). The third column gives the number of versions randomly selected at various stages of the target OSS's lifetime. In total, 158 CVEs and 118 versions are selected for labelling.

We compare each function involved in each patch against the counterpart function in each target version. For example, only one function is modified in the patch for CVE-2020-36184 of the project Jackson-databind. We will conduct seventeen comparisons for the seventeen versions we selected, to analyze which type of annotation situation each version belongs to. In total, we performed 7139 comparisons, and the detailed data are shown in Table III. The second column represents the statistics of the number of comparisons, and the third column represents the corresponding percentage. The fourth and fifth columns are the merged pre-patch and post-patch statistics. Through data analysis, we found that there are only 8% of the comparisons indicates that there is a discrepancy between the target version and the patch code (Situation III). The data show that for most Java OSS versions, the vulnerability existence could be determined without complex semantic extraction.

The above two findings are helpful in guiding our algorithm design for vulnerable version identification. Since a Java patch often involves multiple-function modification (Finding 1), the designed algorithm is supposed to be aware of the fact that whether a version is vulnerable should be inferred by comprehensively considering the patching status of all involved functions. Besides, since a majority of versions are identical to the pre/post-patch code (Finding 2), we tend to not introduce immediately some heavy methods like slicing or normalization when designing the code matching algorithm.

IV. APPROACH AND DESIGN

Fig. 2 shows the overview of VERJava, which mainly contains three steps. The information collection (Sec. IV-A) takes a target project's name and a CVE ID as input, uses a customized crawler to fetch the CVE's patch and collect multiple versions' source code for the target project. The function-level vulnerable version calculation step (Sec. IV-B) takes patch functions and multiple versions' source code as input and generates a list of vulnerable versions for each patch function. The patch-level vulnerable version calculation step (Sec. IV-C) takes all patch functions' vulnerable version lists as input and generates the vulnerable version list for the entire patch. Finally, we simply merge the results of all patches to get the final vulnerable versions of the CVE.

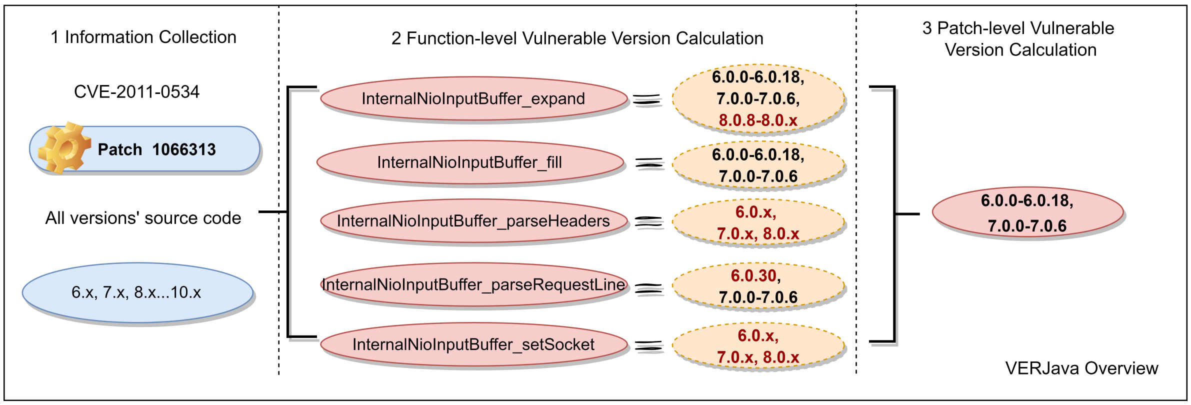

Fig. 2: Overview of VERJava's Approach.

In Fig. 2, we illustrate the workflow of VERJava by using Tomcat's CVE-2011-0534 as an example. This CVE has a fixing patch with an issue management ID 1066313, containing five patch functions. At the information collection step, the patch functions and all versions' source code are collected. At the function-level vulnerable version calculation stage, a list of versions where each involved function is not patched are calculated respectively. For example, the involved function *InternalNioInputBuffer_expand()* is analyzed to be not patched in Tomcat versions 6.0.0-6.0.18, 7.0.0-7.0.6 and 8.0.8-8.0.x. Note that the vulnerable function searching is performed over the versions of all branches, that is why the result contains versions from different branches. At the patch-level vulnerable version calculation stage, basically, since almost all the five functions are un-patched in versions 6.0.0-6.0.18 and 7.0.0-7.0.6, these versions are considered vulnerable to CVE-2011-0534. Each step is further presented in details below.

A. Information Collection

We use different customized crawlers to collect CVE patches and multi-version source code of the target project. The CVE patch files (i.e., diff files) are obtained mainly in three ways: (1) There is a reference link on the CVE description page of NVD, which can directly bring us to the GitHub or SVN patch page. (2) According to the links on the CVE description page, we can find an internal management number of the project's issue list. Then we go to cross-search the project issue list according to the number to obtain the fixing commit. (3) We search for the CVE ID in the description of issue list of the target project and locate the corresponding issue and the fixing patch.

Furthermore, we split each obtained patch into functions, extract function information, such as file location, file name, function name, line numbers and line contents for added and deleted lines, etc, then store them in a JSON format.

B. Function-level Vulnerable Version Calculation

In this stage, we will decide the versions which are vulnerable at a function level, namely, the versions where an involved function is not patched. Preferably, if all deleted

Algorithm 1 Vulnerable Function Calculation

1: Input: $TargetFunc$: function f's source code in a given version V, $Patchfunc$: function f 's patch information in a given CVE
2: Output: whether $TargetFunc$ is vulnerable in V
3: $deleteLineSet \leftarrow$ the set of $Patchfunc$'s deleted lines
4: $addLineSet \leftarrow$ the set of $Patchfunc$'s added lines
5: $delSim$ and $addSim$ are defined in Eq. 1 and 2, resp.
6: **if** $addLineSet \neq \emptyset$ and $deleteLineSet \neq \emptyset$ **then**
7:     **if** $delSim \geq tDel$ and $addSim \leq tAdd$ **then**
8:         **return** vulnerable
9: **else if** $addLineSet == \emptyset$ **then**
10:     **if** $delSim \geq tDel$ **then**
11:         **return** vulnerable
12: **else if** $deleteLineSet == \emptyset$ **then**
13:     **if** $addSim \leq tAdd$ **then**
14:         **return** vulnerable
15: **return** safe

lines in a patch exist and all added lines in a patch do not exist in a function of a given version, then the function is vulnerable. However, patched functions may get changed as a project evolves. To increase the robustness, whether a function f of a given version V is vulnerable should be decided by synthesizing the degree of similarity between the target function and the patch code (to solve *Challenge-I: code evolution processing*). The decision algorithm is presented in Algorithm 1. We calculate the similarity between the target function and the pre/post-patch functions, and decide the vulnerability existence for the function under three different patch types: there are both added and deleted lines, only deleted lines and only added lines in the patch, respectively (Lines 6, 9 and 12 in Algorithm 1).

We denote the similarity between the target function and the pre-patch function by $delSim$, the similarity between the target function and the post-patch function by $addSim$. We use $TargetFunc$ to represent the function f body's lines in the version V's source code. The symbols $delLineSet$ and $addLineSet$ represent the sets of the deleted lines and the

Algorithm 2 Patch-level Vulnerable Version Calculation

```
1: Input: AllPFun: the set of functions in a given patch P, VersionList: the list of target software versions
2: Output: vulnerable versions w.r.t. P
3: TotalNum ← the number of functions in AllPFun
4: VulNum ← the number of vulnerable functions w.r.t. P in version V
5: Initialize vulnerableVersions to be empty.
6: for each V ∈ VersionList do
7:     for each f ∈ AllPFun do
8:         if f does not exist in V then
9:             TotalNum -= 1
10:    if (TotalNum > 3 ⋀ VulNum / TotalNum ≥ T) ∨ (TotalNum ≤ 3 ⋀ VulNum / TotalNum == 1) then
11:        vulnerableVersions.insert(V)
12: return vulnerableVersions
```

added lines of the patch function, respectively. For each line of source code, the tabs and spaces before and after it are deleted, and if a source code sentence spans multiple lines, we merge them into one line. The notation # represents the number of items in a set. The formulas for calculating the function similarities are given below.

$$delSim \leftarrow \frac{\#(delLineSet \bigcap TargetFunc)}{\#delLineSet} \quad (1)$$

$$addSim \leftarrow \frac{\#(addLineSet \bigcap TargetFunc)}{\#addLineSet} \quad (2)$$

To compare the similarities in a robust way, we use two thresholds: the added line existence threshold (called $tAdd$) and the deleted line existence threshold (called $tDel$). When $delSim$ is higher than $tDel$, and $addSim$ is lower than $tAdd$, we consider $TargetFunc$ as vulnerable (Line 6-8). For patches with only one type of modification, we only need to consider the similarity with the pre-patch (Line 9-11) or post-patch (Line 12-14) function. Experiments on the threshold sensitivity have been conducted in Sec. V and the results show that the best effect is achieved when $tdel$ is 1 and $tAdd$ is 0.9. Therefore, $tDel$ and $tAdd$ are empirically set to be 1 and 0.9.

C. *Patch-level Vulnerable Version Calculation*

Theoretically, when a patch is applied, all involved functions are fixed, so the vulnerable versions of all patch functions should be the same. However, due to code modifications in the version evolution process, the vulnerable versions calculated by the previous stage are different for different functions in the same patch, as already seen in Fig. 2. Considering the prevalence of multi-function fixing in Java patches (to solve *Challenge-II: multi-location fix handling*), we need to deal with all patch functions' results in a comprehensive way. The decision algorithm is stated in Algorithm 2.

$TotalNum$ represents the number of functions involved in a given patch P. First we count how many patch functions are still present in a given version V. There are code refactoring in some versions as a project evolves. If a patch function does not exist in V any more, we subtract 1 from $TotalNum$ (Lines 7-9 in Algorithm 2).

Ideally, the version V is considered vulnerable with respect to (w.r.t.) P when the vulnerability version lists of all functions in P contain V. However, this condition is so strict that it may introduce false positives, as there are cases where patched lines are modified as the code evolves. Therefore, here we use a threshold T (Line 10) to control false positives. It is empirically to be 0.8 based on threshold sensitivity analysis in Sec. V. But for the case where the number of patch functions, i.e., $TotalNum$, is less than or equal to three, the ratio $VulNum/TotalNum$ varies greatly and the threshold is strictly required to be 1 (Line 10-11).

Software maintainers usually manage Java projects in multiple branches (e.g., Tomcat maintains multiple branches such as 10.0.x, 9.0.x, 8.0.x, 7.0.x, etc.). Therefore, when fixing a CVE, they also submit patches separately for different branches. Thus we separately analyze the vulnerable versions w.r.t. each patch from different branches, merge the results of all patches and obtain a final list of versions vulnerable to the CVE.

V. Evaluation

In this section, we evaluate VERJava on 167 Java real-world CVEs (detail shows in Table I), and compare it with the state-of-the-art work V-SZZ, regarding the accuracy and time overhead in identifying vulnerable version ranges.

A. *Effectiveness of VERJava*

As shown in Table I, our dataset contains seven real-world open-source Java projects, including 167 CVEs, and extracts 344 patches from them. Our method takes patch function as the unit and collects 1060 patch functions.

Overall Results. Table IV shows the results of VERJava in identifying vulnerable version ranges. Among the 167 CVEs, VERJava can correctly identify the versions of 147 CVEs. It has a precision of 90.7% and a recall of 96.7%. Mention that in Table IV, only if all the affected versions of a CVE are reported correctly, the report is considered as a true positive (TP). For example, the versions annotated for CVE-2017-1000391 contain 61 versions, and VERJava accurately reports the 61 versions that exactly match the annotated version list so that the identification result of CVE-2017-100391 is marked as a TP. Similarly, VERJava reported one more version for CVE-2009-3742, then we will mark it as a false positive (FP). When the result misses versions and reports extra versions at the same time, we count it as a false negative (FN). For example, if the ground truth (GT) of vulnerable versions for a CVE is [6.0.0-6.0.18] but the tool reports [6.0.10-6.0.30], this is marked as a FN. The calculation method for TP/FP/FN used in Table IV is rigorous, while VERJava can still reach a good precision and recall.

Furthermore, we use a set-based metric and calculate another form of precision and recall as shown in Table V. In there, the TP/FP/FN are interpreted as follows: Suppose GT for one CVE is [v1, v2, v3] but the tool reports [v2, v3, v4],

TABLE IV: Accuracy (i.e., True Positive, False Positive and False Negative) of V-SZZ, ΔV-SZZ and VERJava. GT stands for ground truth, Precision equals TP/(TP+FP) and Recall equals TP/(TP+FN).

Target	GT	V-SZZ					ΔV-SZZ					VERJava				
		TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall	TP	FP	FN	Precision	Recall
Tomcat	45	9	0	36	100%	20.0%	33	4	8	89.2%	80.5%	39	6	0	86.7%	100%
Jackson-databind	62	3	1	58	75.0%	4.9%	3	1	58	75.0%	4.9%	57	5	0	91.9%	100%
Struts	14	5	2	7	71.4%	41.7%	6	4	4	60.0%	60.0%	12	0	2	100%	85.7%
Jenkins	16	7	0	9	100%	43.8%	7	0	9	100%	43.8%	12	1	3	92.3%	80.0%
Liferay portal	20	-	-	-	-	-	-	-	-	-	-	17	3	0	85%	100%
Spring-security	3	0	2	1	0%	0%	0	2	1	0%	0%	3	0	0	100%	100%
Spring-framework	7	0	0	7	0%	0%	1	0	6	100%	14.3%	7	0	0	100%	100%
Total	167	24	5	118	82.8%	16.9%	50	11	86	82.0%	36.8%	147	15	5	90.7%	96.7%

TABLE V: Set-based precisions and recalls of V-SZZ, ΔV-SZZ and VERJava. Jackson-d.(Jackson-databind), Spring-f.(Spring-framework), Spring-s.(Spring-security). Precision is TP/(TP+FP) and Recall is TP/(TP+FN).

Target	V-SZZ		ΔV-SZZ		VERJava	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
Tomcat	22.2%	21.5%	80.5%	83.7%	94.4%	100%
Jackson-d.	7.7%	8.6%	7.7%	8.6%	96.0%	100%
Struts	57.0%	54.2%	82.8%	77.7%	100%	88.1%
Jenkins	74.0%	66.3%	74.0%	66.3%	98.0%	93.3%
Liferay portal	-	-	-	-	88.9%	100%
Spring-s.	35.7%	50.0%	69.0%	62.5%	100%	100%
Spring-f.	28.6%	14.3%	63.5%	37.0%	100%	100%
Total	22.5%	21.4%	42.5%	41.3%	95.5%	98.4%

then TP = 2, FP = 1, FN = 1. A precision is computed for each CVE and the precision of each project is obtained by averaging the precisions of all its CVEs. As can be seen in Table V, VERJava achieves an even better precision (95.5%) and recall (98.4%) in this way.

False Positive and False Negetive Analysis for VERJava. VERJava has 9.3% FP in Table IV. In the example CVE-2013-2067 in Listing 1, there are only added lines in the patch for Version 7.0.33. But in 7.0.100, the added lines have class and method modifications. We confirmed through manual auditing that 7.0.100 has fixed the vulnerability. While VERJava cannot sense that because of the significant semantic and code changes, which leads to FPs. There are five FNs in Table IV, all of which are since some versions only exist in one or a few patch functions' results, which account for a small proportion of the total functions. Nonetheless, the versions that were found to be FN are early versions, and more attention is paid to the version with a relatively recent release time in the real-world vulnerability analysis, so the impact of under-reporting in practical applications should be small.

B. Comparison with V-SZZ

To the best of our knowledge, only tools in the SZZ series directly target vulnerable OSS version identification and use also static analysis. V-SZZ is the state-of-the-art one among them, so only it was selected for comparison. We found that most CVEs could not be successfully run on V-SZZ (close to 70%) for the following reasons (we counted these cases as V-SZZ's FN): (1) V-SZZ cannot handle those patches fixed only by adding lines because of the inherent limitation of its design based on the git blame feature. In our dataset, such cases account for over 30% (61 out of 167 CVEs). While, VERJava can handle such patch types. (2) V-SZZ only processes patches with less than five lines of deletion due to time consumption. According to our statistics, patches involving multiple lines are prevalent. Here we improve V-SZZ (asΔV-SZZ) to remove this 5-line limit and evaluate them separately. The results are shown in Table IV and V. It can be seen that VERJava achieves the best precision and recall.

TABLE VI: Time overhead of the tools in comparison.

Tool	CVE	Max Time	Min Time	Average Time
VERJava	167	0.31min	10ms	0.71s
V-SZZ	42	2.65min	30ms	7.60s
ΔV-SZZ	80	28.69min	30ms	29.39s

We further analyze why VERJava significantly outperforms V-SZZ as follows. Specifically, for multi-branch management, different commit IDs would be assigned to each submission. V-SZZ highly depends on the commit IDs. For example, CVE-2020-36183 submitted two fixing commits on the branches 2.0 and 2.9, respectively. V-SZZ needs the both commit IDs to determine the version range accurately, missing commit IDs could cause FN. Whereas, VERJava uses patch content for analysis, therefore it accurately identifies the vulnerable version range for this CVE based on only one commit. Moreover, the inaccuracy of several reports given by V-SZZ is due to the defects in the official commit management. There are situations such as wrong-submission of some tags or multiple submissions when fixing a CVE (e.g., when fixing CVE-2020-11111, the software maintainer first submitted an incorrect patch, and then submitted one more patch for revision). The git blame command cannot be used on Liferay portal because its file path is too long, so we did not apply V-SZZ on this project which therefore does not have results in Table IV and V.

Time Overhead. Both VERJava and V-SZZ require a manual or semi-automatic part of the patch collection. We collect as many source code versions as possible in advance, and V-SZZ requires git checkout operations for each analysis, so our time overhead is shorter in terms of analysis time. In total, VERJava only needs 118.5s to analyze 167 CVEs, and the average analysis time of each CVE is 0.71s as shown in Table VI. The analysis time is positively correlated with the number of patch functions of CVEs. There is one patch function with the least time consumption which is only 10ms. The most time-consuming one is CVE-2012-0022, which has

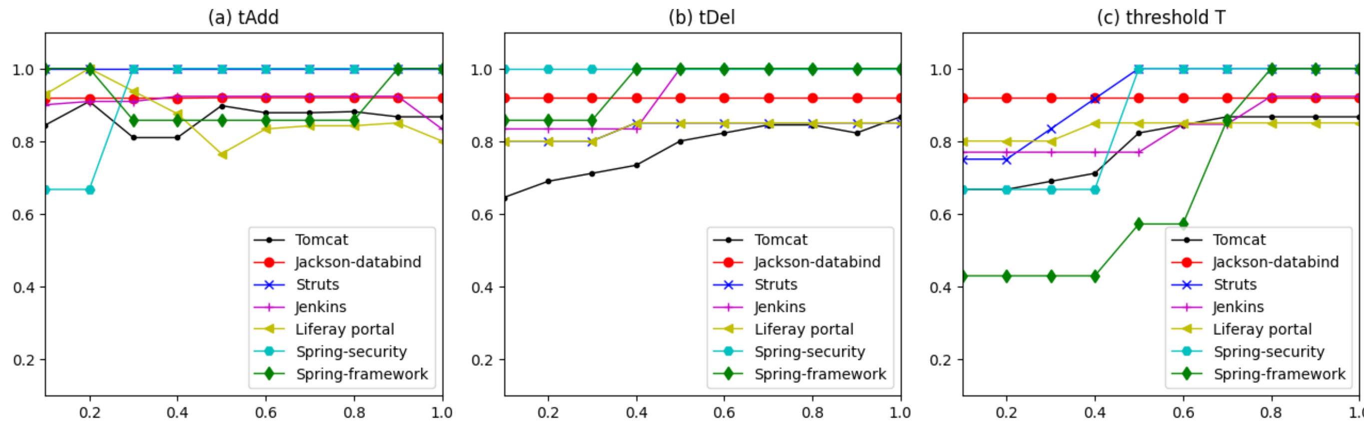

Fig. 3: Precision vs. Three Thresholds (x-axis denotes the value of threshold, and y-axis denotes precision).

46 patch functions costing 0.31min. Moreover, the average time of V-SZZ is 7.6s which is worse than that of VERJava. The time consumption of ΔV-SZZ is worse than V-SZZ. After removing the 5-line limit of V-SZZ, although more CVEs can be processed, the time consumption is greatly increased.

C. *Threshold Sensitivity Analysis*

Three thresholds (i.e., tAdd, tDel, T) are configurable in the steps of VERJava. The default configuration is 0.9, 1, 0.8, which is used in the experiment in Algorithm 1 and Algorithm 2. To evaluate the sensitivity of these thresholds to the accuracy, we reconfigured one threshold and fixed the other two, and ran VERJava against the seven targets. As $tAdd$ and $tDel$ are used to determine whether a function is vulnerable, they were reconfigured from 0.1 to 1 by a step of 0.1. As T is adopted to determine whether CVE exists in version V, it was reconfigured from 0.1 to 1 by a step of 0.1. In total, 29 (i.e., 3 × 10 - 1) configurations of VERJava were run.

Fig. 3 presents the impact of three thresholds on precision, respectively. Overall, before $tAdd$ increased to 0.9, the precision was almost stable in most target systems. As $tAdd$ increased from 0.9 to 1.0, the precision decreased, since many TP were missed due to patch function modification in version evolution. As the precision kept increasing when $tDel$ increased from 0.1 to 1. We believe that 0.9 and 1 are good values for $tAdd$ and $tDel$, respectively. On the other hand, as T increased from 0.0 to 0.8, the precision was increasing. Thus, we believe 0.8 to 1 is a good value for T.

In Fig. 3, some low values of $tAdd$ and $tDel$ still show good precision for some projects such as Jackson-databind. The reason is that for some patches, the matched added or deleted lines barely change during the version evolution (i.e., Finding 2), resulting in $delSim/addSim$ being unchanged (either close to 0 or 1). Thus though $tAdd$ and $tDel$ change, the precision is unchanged for some patches. Taking Jackson-databind as an example, 71% of its patches have only one function and only added lines, so changes in $tDel$ and T have little effect.

D. *Interesting Findings*

Some more interesting findings are given below.

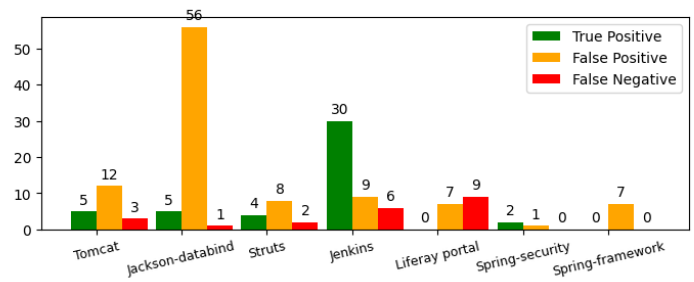

Fig. 4: Software labelling accuracy statistics of NVD.

Accuracy of NVD version labeling in seven target projects. We manually annotated 167 selected CVEs. The annotation results show that NVD has incorrect version annotations for 121 of the 167 CVEs (72%). The accuracy of NVD versions in seven projects is shown in Fig. 4. The NVD ranges seem overly conservative (i.e., FPs are more than FNs).

Sibling classes and interface class fixes. During the manual analysis of the patch, we found repairs to sibling classes and interface classes. For example, when CVE-2013-4322 was fixed, the same correction was applied in four classes. *AbstractHttp11Processor* and *Http11Processor* are interface classes, while *Http11AprProcessor* and *Http11NioProcessor* are siblings. According to statistics, 34 of the 167 CVEs have sibling repairs, and 9 have interface repairs. We think this observation would give some clues in better recurring vulnerability detection for OO language. Specifically, by giving higher weights to similar code snippets in sibling or interface classes, more recurring vulnerabilities may hopefully be found.

VI. Discussion

In this section, we discuss the threat to validity and possible future improvement of VERJava.

VERJava does not differentiate between secure and non-secure patches. We think that all the obtained patches are security fixes. But patches submitted may contain security fixes and non-security fixes. When doing manual annotation and tool analysis, we assumed that all codes modified in the patch are security-related. In the future, it is desirable to distinguish between security and non-security fixes and set different weights for them so that the analysis results obtained will be more credible.

VERJava does not consider the vulnerable function/file absence imposed by code refactoring for Situation I in Sec. III-B. In Situation I, we assume that the absence of a patched function/file tends to show the absence of vulnerability. However, due to refactoring, a vulnerability may be present, but its location can be scattered across different functions (or part of a larger function) in early versions of the project. To mitigate the problem, a function migration mapping method could be designed and built during the code refactoring, and the search scope of vulnerable functions is expanded based on this mapping.

The vulnerable versions found by VERJava have not been verified. Like other static analysis methods, VERJava does not contain dynamic vulnerability triggering, which requires a considerable amount of extra work. For instance, the triggering of some vulnerabilities requires a particular configuration to be set. In future, performing dynamic vulnerability triggering based on the results of static analysis would definitely give a more accurate vulnerable version range.

VII. Related Work

We review the most closely related work in vulnerable version identification and patch analysis.

A. Vulnerable Version Identification

Sliwerski et al. [14] proposed a technique called Sliwerski-Zimmermann-Zeller (SZZ for short) that automatically locates bug-inducing changes. Basically, SZZ leverages the *diff* command provided by the version control systems (VCS) to identify the lines of code modified by the bug-fixing changes, and traces back through the code history, to identify the bug-inducing changes by using the *annotate* command in the VCS. Many researchers have improved SZZ either for increasing the accuracy in identifying bug-inducing commits (OpenSZZ [16], AG-SZZ [31], MA-SZZ [32], etc.) or for identifying vulnerability-inducing commits (Nguyen et al. [5], V-SZZ [7]). However, as been discussed in Sec. II-A, SZZ-based methods highly rely on the blame feature of VCS, thus can not handle patches which were fixed by only adding lines. Moreover, the current works do not consider the characteristics of multi-branch management, multi-function repair and multi-line repair of the object-oriented language's patch, which have been handled in this paper.

B. Patch Analysis

The target software of existing works can be divided into closed source software(CSS) and open source software(OSS).

Patch analysis work for CSS is mainly used for patch presence testing. FIBER [33] generates binary signatures that can reflect the representative syntax and semantic changes introduced by the patch. It insufficiently considers the code variance incurred by third-party customization and non-standard building configurations. PDIFF [34] and BSCOUT [35] have made improvements for this weakness. But for similarity computation between binaries, PDIFF generates relatively heavy-weight code feature, i.e., sets of patch-related paths, which is less influenced by compilation and building configurations. While in this paper, we uses lightweight but effective code features for OSS. After mapping source codes to Java bytecodes, BSCOUT adopts line-level similarity analysis but it does not consider from function-level perspective and neither Challenge I nor II which are proposed in the paper.

Patch analysis work for OSS is mainly used for vulnerable code clone detection. ReDeBug [19] establishes a sliding window and judges whether the target code has a similar vulnerability according to the code in the window. VUDDY [20] unifies all identifiers in code into abstract representations (e.g., every called function becomes 'FUNCCALL'). MVP [17] designs a code slicing method based on the patch code for vulnerablity-related feature extraction. They all target C/C++ and neglect the characteristcs in patches for object-oriented language. Besides traditional static analysis, some researchers integrated machine learning methods. VCCFinder [18] and Vulpecker [36] both use an SVM classifier to flag suspicious target code. On basis of Vulpecker, deep-learning algorithms are studied to boost accuracy [37] [38]. However, for the task of vulnerable version identification, since the code differences between different versions are small (i.e., Finding 2), an accurate identification can be achieved by using lightweight code features, without introducing the code abstraction and slicing as done in the above works.

VIII. Conclusions

In this paper, we collected 167 CVEs across seven popular Java open-source software. Through manual analysis, we found that the version labels of 121 CVEs of NVD were inaccurate, and Java patches have the characteristics of multi-function repair, multi-line repair and multi-branch management. To handle these characteristics, we have proposed and implemented a novel approach named VERJava to accurately identify the range of vulnerable versions. Our evaluation results have demonstrated that VERJava can significantly outperform the state-of-the-art approach V-SZZ.

Acknowledgment

This research was partly supported by the Chinese National Natural Science Foundation(U1836209, 62032010, 61802394), 2020YFB1005704, 2021-JCJQ-JJ-0921, Strategic Priority Research Program of the CAS (XDC02040100), the Key Laboratory of Network Assessment Technology, Chinese Academy of Sciences and Beijing Key Laboratory of Network Security and Protection Technology.

References

[1] GitHub, "GitHub 2021 Report," https://octoverse.github.com/.
[2] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 356–367.
[3] CVE, "Common vulnerabilities and exposures," https://cve.mitre.org/.
[4] NVD, "National vulnerability database," https://nvd.nist.gov/.

[5] V. H. Nguyen, S. Dashevskyi, and F. Massacci, "An automatic method for assessing the versions affected by a vulnerability," *Empirical Software Engineering*, vol. 21, no. 6, pp. 2268–2297, 2016.
[6] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 869–885.
[7] L. Bao, X. Xia, A. E.Hassan, and X. Yang, "SZZ for vulnerability: Automatic identification of version ranges affected by cve vulnerabilities." ICSE, 2022.
[8] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 396–407.
[9] Y. Dong, W. Guo, Y. Chen, X. Xing, Y. Zhang, and G. Wang, "Towards the detection of inconsistencies in public security vulnerability reports," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 869–885.
[10] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3282–3299.
[11] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "SemFuzz: Semantics-Based Automatic Generation of Proof-of-Concept Exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2139–2154.
[12] J. Dai, Y. Zhang, H. Xu, H. Lyu, Z. Wu, X. Xing, and M. Yang, "Facilitating vulnerability assessment through poc migration," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 3300–3317.
[13] Apache, "Tomcat," https://tomcat.apache.org/index.html.
[14] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" *ACM sigsoft software engineering notes*, vol. 30, no. 4, pp. 1–5, 2005.
[15] N. Alexopoulos, M. Brack, J. P. Wagner, T. Grube, and M. Mühlhäuser, "How long do vulnerabilities live in the code? a Large-Scale empirical measurement study on FOSS vulnerability lifetimes," in *31st USENIX Security Symposium (USENIX Security 22)*, Aug. 2022.
[16] V. Lenarduzzi, F. Palomba, D. Taibi, and D. A. Tamburri, "OpenSZZ: A free, open-source, web-accessible implementation of the szz algorithm," in *Proceedings of the 28th international conference on program comprehension*, 2020, pp. 446–450.
[17] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou *et al.*, "MVP: Detecting vulnerabilities using Patch-Enhanced vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1165–1182.
[18] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "VCCFinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.
[19] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: finding unpatched code clones in entire os distributions," in *2012 IEEE Symposium on Security and Privacy*. IEEE, 2012, pp. 48–62.
[20] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *2017 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017, pp. 595–614.
[21] Apache, "CVE-2013-2067 7.x branch commit," https://svn.apache.org/viewvc?view=revision&revision=1408044.
[22] FasterXML, "CVE-2017-17485 commit," https://github.com/FasterXML/jackson-databind/commit/2235894210c75f624a3d0cd60bfb0434a20a18bf.
[23] Apache, "CVE-2011-0013 6.x branch commit," https://svn.apache.org/viewvc?view=revision&revision=1057270.
[24] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *Proceedings of the 16th International Conference on Mining Software Repositories*, ser. MSR '19. IEEE Press, 2019, p. 383–387.
[25] Jenkins, "Jenkins," https://www.jenkins.io/.
[26] Apache, "Struts," https://struts.apache.org/.
[27] FastXML, "Jackson-databind," https://github.com/FasterXML/jackson-databind/wiki.
[28] Spring, "Spring-framework," https://spring.io/projects/spring-framework.
[29] ——, "Spring-security," https://spring.io/projects/spring-security.
[30] Liferay, "Liferay-portal," https://www.liferay.com/zh/locations.
[31] S. Kim, T. Zimmermann, K. Pan, and E. J. Jr. Whitehead, "Automatic identification of bug-introducing changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 81–90.
[32] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, "A framework for evaluating the results of the szz approach for identifying bug-introducing changes," *IEEE Transactions on Software Engineering*, vol. 43, no. 7, pp. 641–657, 2017.
[33] H. Zhang and Z. Qian, "Precise and accurate patch presence test for binaries," in *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, Aug. 2018, pp. 887–902.
[34] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "PDIFF: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1149–1163.
[35] J. Dai, Y. Zhang, Z. Jiang, Y. Zhou, J. Chen, X. Xing, X. Zhang, X. Tan, M. Yang, and Z. Yang, "BScout: Direct whole patch presence test for java executables," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1147–1164.
[36] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 201–213.
[37] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," *arXiv preprint arXiv:1801.01681*, 2018.
[38] D. Zou, S. Wang, S. Xu, Z. Li, and H. Jin, "μVulDeePecker: A deep learning-based system for multiclass vulnerability detection," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 5, pp. 2224–2236, 2019.