Vision: Identifying Affected Library Versions for Open Source Software Vulnerabilities

Susheng Wu*
School of Computer Science
Fudan University
Shanghai, China

Ruisi Wang*
School of Computer Science
Fudan University
Shanghai, China

Kaifeng Huang†
School of Software Engineering
Tongji University
Shanghai, China

Yiheng Cao*
School of Computer Science
Fudan University
Shanghai, China

Wenyan Song*
School of Computer Science
Fudan University
Shanghai, China

Zhuotong Zhou*
School of Computer Science
Fudan University
Shanghai, China

Yiheng Huang*
School of Computer Science
Fudan University
Shanghai, China

Bihuan Chen*†
School of Computer Science
Fudan University
Shanghai, China

Xin Peng*
School of Computer Science
Fudan University
Shanghai, China

ABSTRACT

Vulnerability reports play a crucial role in mitigating open-source software risks. Typically, the vulnerability report contains affected versions of a software. However, despite the validation by security expert who discovers and vendors who review, the affected versions are not always accurate. Especially, the complexity of maintaining its accuracy increases significantly when dealing with multiple versions and their differences. Several advances have been made to identify affected versions. However, they still face limitations. First, some existing approaches identify affected versions based on repository-hosting platforms (*i.e.,* GitHub), but these versions are not always consistent with those in package registries (*i.e.,* Maven). Second, existing approaches fail to distinguish the importance of different vulnerable methods and patched statements in face of vulnerabilities with multiple methods and change hunks.

To address these problems, this paper proposes a novel approach, Vision, to accurately identify affected library versions (ALVs) for vulnerabilities. Vision uses library versions from the package registry as inputs. To distinguish the importance of vulnerable methods and patched statements, Vision performs critical method selection and critical statement selection to prioritize important changes and their context. Furthermore, the vulnerability signature is represented by weighted inter-procedural program dependency graphs that incorporate critical methods and statements. Vision determines ALVs based on the similarities between these weighted graphs. Our evaluation demonstrates that Vision outperforms state-of-the-art approaches, achieving a precision of 0.91 and a recall of 0.94. Additionally, our evaluation shows the practical usefulness of Vision in correcting affected versions in existing vulnerability databases.

*S. Wu, R. Wang, Y. Cao, W. Song, Z. Zhou, Y. Huang, B. Chen and X. Peng are also with Shanghai Key Laboratory of Data Science, Fudan University, China.
†K. Huang and B. Chen are the corresponding authors.

CCS CONCEPTS

• **Security and privacy → Vulnerability management**; • **Software and its engineering → Software safety**; • **Information systems → Open source software**;

KEYWORDS

open source software, vulnerability quality, affected versions

ACM Reference Format:
Susheng Wu, Ruisi Wang, Kaifeng Huang, Yiheng Cao, Wenyan Song, Zhuotong Zhou, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Vision: Identifying Affected Library Versions for Open Source Software Vulnerabilities. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695516

1 INTRODUCTION

Open-source software (OSS) promotes innovation sharing and accelerates software development, which has become a key infrastructure in modern industry. Despite its benefits, it also presents security risks. Vulnerabilities in OSS can be exploited by attackers to compromise downstream software. A recent report from Sonatype [52] highlights that approximately 12.5% of OSS downloads contain known vulnerabilities. Therefore, detecting vulnerabilities in OSS is crucial for software security. Fortunately, there are OSS vulnerability databases containing vulnerability reports which are crowd-sourced from security experts. They help downstream clients determine if their applications are affected by matching the vulnerable OSS versions in the database to the versions used in the applications.

Problem. However, the manually compiled vulnerability reports would inevitably introduce inaccuracies [8, 14, 15, 30]. Particularly, the field of affected versions which indicates the vulnerable and safe versions to the vulnerability, is prone to inaccuracies. The primary reason is that while security experts can actively identify and confirm vulnerabilities, checking across multiple versions is labor-intensive and less appealing to their objectives. These inaccuracies can significantly impact the consumers of vulnerability reports, e.g., OSS vulnerability management applications [28, 32, 61, 73].

Existing Approaches. Various methods have been proposed to identify affected library versions (ALVs). Dong et al. [14] proposed to use named entity recognition to extract ALVs from vulnerability descriptions, which is limited by the quality of the descriptions. Dai et al. [13] utilized trace-guided fuzzing to detect and validate ALVs. Although it provides convincing proof-of-concepts, the procedure is both computationally intensive and time-consuming. Shi et al. [48] leveraged taint analysis to identify ALVs. However, it requires manual curation of dangerous functions, which is restricted to certain types of vulnerabilities. Researchers have increasingly focused on statically analyzing source code by matching the vulnerability patches or vulnerable clones. Specifically, *Patch-based approaches* [2, 27, 38, 55] identify ALVs by tracing code changes in version history. *Clone-based approaches* [31, 65–67, 70] generate clone-based fingerprints from the modified methods in vulnerability, and ALVs are reported if fingerprints are matched in ALVs.

Limitations. The existing methods have several limitations. **(a)** They identify library versions from repository-hosting platforms (e.g., GitHub). They overlook the discrepancies of library versions between repository-hosting platforms and package registries (e.g., Maven repository). As a result, this leads to missed detections of library versions in package registries. Since a large portion of downstream consumers retrieves libraries from these registries, it significantly impacts the effectiveness of vulnerability detection tools. **(b)** *Patch-based approaches* depend on the change type (*i.e.,* added lines or deleted lines) in the patch, and cannot report ALVs when the patch does not contain deleted lines. Moreover, these approaches do not include the context of the vulnerability (*i.e.,* depended and depending statements), which inevitably leads to false alarms. **(c)** *Clone-based approaches* add the vulnerability context by incorporating program slicing [65, 66, 70]. However, they assign equal importance for all fixing methods (*i.e.,* methods used to patch vulnerabilities) within a vulnerability. They neither differentiate the importance of fixing methods nor assign different priorities to the changed statements within a fixing method.

Our Approach. We propose a novel approach, Vision, to identify affected library versions of OSS vulnerabilities. Unlike prior approaches, Vision analyzes library versions from the package registry (*i.e.,* Maven) to overcome limitation **(a)**. The Maven repository serves as the default repository for the Maven package manager, offering a comprehensive list of library versions available for both manual downloads and automatic requests. The key insight of Vision is that different methods or statements contain unequal semantic knowledge of the vulnerability. Therefore, different from existing approaches which place equal importance on the methods and statements to generate vulnerability signatures, Vision distinguishes critical methods and critical statements from the rest of the methods and statements. Specifically, Vision conducts *vulnerability and patch signature generation* for each vulnerability, and generates weighted inter-procedural program dependency graphs (weighted IPDGs). The weighted IPDGs encode the criticalness of methods and statements into a graph representation. Simultaneously, Vision performs *vulnerability-potential and patch-potential version signature generation* for candidate library versions, generating signatures specific to vulnerable and patched versions which are also weighted IPDGs. Subsequently, Vision performs *affected library version detection* by comparing similarities between IPDGs.

Specifically, Vision not only generates a vulnerability signature for the vulnerable library version and a vulnerability-potential signature for the candidate library version using deleted lines, but also generates patch signature for patched library version and patch-potential signature for candidate library version using added lines. It compares the similarity of a candidate library version to the vulnerable code while ensuring dissimilarity to the patched code. Vision can leverage the original context to generate vulnerability signatures even when there are no deleted lines, which effectively addresses limitation **(b)**. To overcome limitation **(c)**, Vision employs the Hyperlink-Induced Topic Search algorithm (HITS) [63] to select critical methods on the Method Reference Graph (MRGs). Furthermore, it highlights critical statements and paths in weighted IPDGs by identifying the critical variables from the patch.

Evaluation. We evaluate Vision's effectiveness by comparing it against two state-of-the-art patch-based approaches, and three advanced clone-based approaches across 102 CVEs involving 79 libraries and 12,073 version pairs. Our results indicate that Vision achieves a precision of 0.91 and a recall of 0.94, significantly outperforming by at least 12.3% and 154.1% to the state-of-the-art approaches, respectively. Specifically, Vision reports 357 false positives and 184 false negatives in the overlapping library versions, and 418 false positives and 258 false negatives in the complete ground truth. Comparatively, patch-based approaches averagely report 1,031 false positives and 1,654 false negatives in the overlapping library versions, 1,720 false positives and 2,961 false negatives in the complete ground truth. Clone-based approaches averagely report 286 false positives and 3,826 false negatives in the complete ground truth. Additionally, ablation study and threshold sensitivity analyses confirm the contributions of components in Vision to its overall effectiveness. Besides, we demonstrate the generalizability of Vision by applying it to the original datasets of V-SZZ and Ver-Java, where it achieves comparable effectiveness with precision of 0.90 and recall of 0.92. To demonstrate the practical usefulness of Vision, we use Vision to analyze vulnerabilities that are labeled with incorrect affected library versions in five vulnerability databases and report them to the five vendors. Three vendors have replied and fixed 39, 42 and 8 vulnerabilities, respectively.

Contribution. Our paper makes the following contributions.

- We propose Vision to identify affected library versions for OSS vulnerabilities. It accepts library versions from the Maven and encodes the criticality of methods and statements of vulnerabilities into weighted IPDGs for accurate identification.
- Our experiment has demonstrated the effectiveness and practical usefulness of Vision, outperforming the state-of-the-art in precision and recall by at least 12.3% and 154.1%, respectively.

2 MOTIVATION

We first introduce inaccurate affected library versions and three limitations in existing approaches that motivate Vision's design.

Table 1: Versions of `Armeria` Affected by `CVE-2021-43795` in Vulnerability Databases and SOTA Approaches (v_0 is the earliest version; red text denotes the falsely identified versions)

(a) Vulnerability Databases

Databases	ALVs
NVD	[0.4.0, 1.13.3]
GitHub Advisory	[1.12.0,1.13.3]
GitLab Advisory	[1.12.0,1.13.3]
Snyk	[v_0, 1.12.0]
Veracode	[0.71.0,1.13.3]
Ground Truth	[1.12.0,1.13.3]

(b) SOTA Approaches

Tools	ALVs
V-SZZ	[0.69.0, 1.11.0], [1.12.0, 1.13.3]
VerJava	[0.87.0, 1.11.0], [1.12.0, 1.13.3]
V0Finder	0.81.1, [0.87.0, 1.11.0], [1.12.0, 1.13.3]
VUDDY	∅
MVP	∅
Ground Truth	[1.12.0,1.13.3]

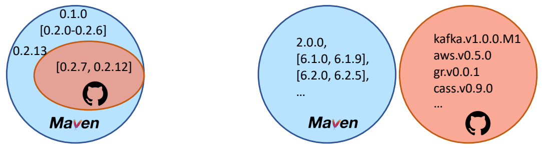
2.1 Inaccurate Affected Library Versions in Vulnerability Databases

The inaccurate Affected Library Versions (ALVs) in vulnerability databases mean that the affected library versions of a vulnerability either contain wrong versions or miss versions. It has been widely recognized and remains a significant challenge [8, 14, 15, 30]. Table 1(a) illustrates the inconsistency in reporting affected library versions related to `Armeria` and `CVE-2021-43795` [42] across various vulnerability databases. The ground truth was determined by manually inspecting the source code on each version. Specifically, only two of the five databases (*i.e.,* NVD [41], GitHub Advisory [19], GitLab Advisory [22], Snyk [51], and Veracode [59]) provide the correct range of affected library versions. v_0 indicates that no left affected version bound is provided, defaulting to the earliest version. This discrepancy underscores the challenge of relying solely on vulnerability databases for precise affected library versions.

2.2 Limitations of Existing Works

Library Versions from Repository-Hosting Platforms. The library versions on the repository-hosting platforms and package registries can be discrepant. For example, we compare library versions of *spring-boot-actuator-logview* and *spring-integration-zip* on Maven and GitHub, as shown in Fig. 1. For *spring-boot-actuator-logview* (see Fig. 1(a)), 15 versions are released on Maven [36], whereas only 7 versions are available on GitHub [20]. For *spring-integration-zip* (see Fig. 1(b)), 21 versions are released on Maven [37], while 47 versions are released on GitHub [21]. Their overlap is zero.

To observe the prevalence of discrepancies between the repository-hosting platform and package registry, we conducted an empirical analysis. Specifically, we identified 539 libraries on Maven and their corresponding 434 GitHub repositories from the 1,083 CVEs related to Java libraries collected during the first step of ground truth construction in Sec 4.1. Note that one repository may contain multiple libraries. Then, we extract and compare versions from Maven and GitHub repositories. We normalized the original versions by converting them into semantic versioning, removing library-related prefixes/suffixes or other meaningless elements. For example, the version `rel/v5.4-beta1` of the Apache HttpComponents Client in its GitHub repository was normalized to `5.4-beta1`. As a result, 511 (94.8%) Maven libraries and 407 (93.8%) GitHub repositories contain discrepant versions compared with their counterparts. The discrepant versions account for 28.0% (15,341/54,713) of the total versions. Furthermore, we compared newer versions that were released over the past three years (*i.e.,* 07-2021 to 07-2024). The discrepancies persist, with 471 (87.4%) Maven libraries and 375 (86.4%) GitHub repositories containing discrepant versions. They make up 25.2% (12,067/49,356) of the total versions.



(a) Versions of *spring-boot-actuator-logview* (b) Versions of *spring-integration-zip*

Figure 1: Discrepant Library Versions on Maven and GitHub (the blue circles denote versions from Maven, while the red circles denote versions from GitHub)

The discrepancies between the two platforms pose challenges in identifying the ALVs, especially for *patch-based* approaches [2, 27, 38, 55]. These approaches depend on the source code repository history and their annotated versions in commits to pinpoint library versions. However, the identified ALVs may not represent all the library versions that developers may use from package registries. In fact, due to the complexity of the software supply chain, many libraries, both direct and transitive, are retrieved from package registries by package managers' automatic requests.

Equal Importance to Changed Methods. We compared the capabilities of two *patch-based* approaches (*i.e.,* V-SZZ [2] and VerJava [55]) and three *clone-based* approaches (*i.e.,* V0Finder [67], Vuddy [31] and MVP [70]) in identifying ALVs for the same vulnerability used in Sec. 2.1. We provided two *patch-based* approaches with library versions tagged in the GitHub repository, as they rely on source code repositories. Additionally, we provided three *clone-based* approaches with candidate library versions retrieved from Maven. As presented in Table 1(b), V-SZZ, VerJava, and V0Finder reported 49, 68 and 69 vulnerable versions, respectively. Meanwhile, Vuddy and MVP did not report any affected library versions. The falsely identified library versions are denoted in red.

We examined the root cause of this vulnerability. Fig. 2 illustrates the evolution history of `Armeria`. Specifically, the root cause was introduced in commit `a380cf` and fixed in commit `e2697a` [43, 44]. There are multiple changed methods in commit `e2697a`. However, if equal importance is given to changed methods, detection tools may erroneously trace method changes back to commits prior to commit `a380cf`. For instance, V-SZZ pinpoints the deleted statement s_1 in the `appendHexNibble()` method as vulnerable and traces it to commit `bf1ee5`. However, this method does not contribute to the root cause of the vulnerability. Consequently, V-SZZ falsely labels versions from 0.69.0 to 1.12.0 as vulnerable. Similarly, VerJava and V0Finder also include falsely identified versions in their assessments, as they consider insignificant methods as indicative of the vulnerability. Vuddy and MVP did not report any vulnerable versions because of failed identification of any method in this case. Besides, our ablation evaluation results (see Section 4.3) also indicate the necessity of selecting critical methods.

Equal Importance to Changed Statements. We use one of the changed methods (*i.e.,* `crypt_raw`) in the patch of `CVE-2022-22976` [45] in `spring-security` as an illustrative example, which is presented in Fig. 3. We highlight the modified lines, collapse the unchanged ones, and remove blank lines for improved clarity. The line numbers remain consistent with the original numbers. We analyze the cause of the vulnerability and the reasoning behind the

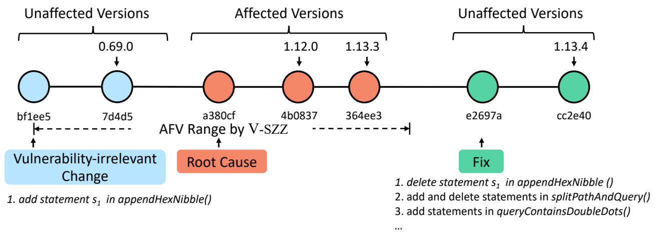
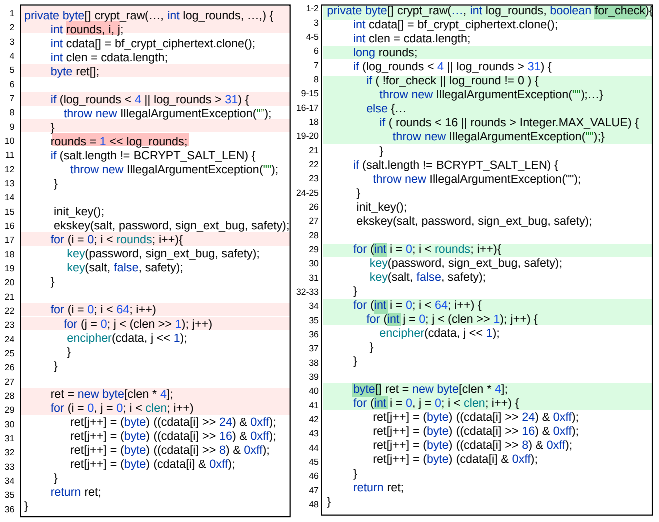

Unaffected Versions
Affected Versions
Unaffected Versions
0.69.0
1.12.0
1.13.3
1.13.4
bf1ee5
7d4d5
a380cf
4b0837
364ee3
e2697a
cc2e40
AFV Range by V-SZZ
Vulnerability-irrelevant Change
Root Cause
Fix
1. add statement s_1 in appendHexNibble()
1. delete statement s_1 in appendHexNibble ()
2. add and delete statements in splitPathAndQuery()
3. add statements in queryContainsDoubleDots()
...

Figure 2: Evolution History of `CVE-2021-43795` in `Armeria`.



```
1   private byte[] crypt_raw(…, int log_rounds, …,) {
2           int rounds, i, j;
3           int cdata[] = bf_crypt_ciphertext.clone();
4           int clen = cdata.length;
5           byte ret[];
6   
7           if (log_rounds < 4 || log_rounds > 31) {
8              throw new IllegalArgumentException("");
9           }
10          rounds = 1 << log_rounds;
11          if (salt.length != BCRYPT_SALT_LEN) {
12                throw new IllegalArgumentException("");
13           }
14  
15           init_key();
16           ekskey(salt, password, sign_ext_bug, safety);
17          for (i = 0; i < rounds; i++){
18              key(password, sign_ext_bug, safety);
19              key(salt, false, safety);
20          }
21  
22          for (i = 0; i < 64; i++)
23             for (j = 0; j < (clen >> 1); j++)
24              encipher(cdata, j << 1);
25              }
26           }
27  
28          ret = new byte[clen * 4];
29          for (i = 0, j = 0; i < clen; i++)
30                ret[j++] = (byte) ((cdata[i] >> 24) & 0xff);
31                ret[j++] = (byte) ((cdata[i] >> 16) & 0xff);
32                ret[j++] = (byte) ((cdata[i] >> 8) & 0xff);
33                ret[j++] = (byte) (cdata[i] & 0xff);
34           }
35          return ret;
36  }
```

(a) crypt_raw before change

```
1-2   private byte[] crypt_raw(…, int log_rounds, boolean for_check){
3             int cdata[] = bf_crypt_ciphertext.clone();
4-5           int clen = cdata.length;
6             long rounds;
7             if (log_rounds < 4 || log_rounds > 31) {
8                if ( !for_check || log_round != 0 ) {
9-15                throw new IllegalArgumentException("");…}
16-17          else {…
18                 if ( rounds < 16 || rounds > Integer.MAX_VALUE) {
19-20                 throw new IllegalArgumentException("");}
21                }
22            if (salt.length != BCRYPT_SALT_LEN) {
23                  throw new IllegalArgumentException("");
24-25          }
26             init_key();
27             ekskey(salt, password, sign_ext_bug, safety);
28    
29            for (int i = 0; i < rounds; i++){
30                key(password, sign_ext_bug, safety);
31                key(salt, false, safety);
32-33         }
34            for (int i = 0; i < 64; i++) {
35               for (int j = 0; j < (clen >> 1); j++) {
36                  encipher(cdata, j << 1);
37                }
38            }
39    
40            byte[] ret = new byte[clen * 4];
41            for (int i = 0, j = 0; i < clen; i++) {
42                 ret[j++] = (byte) ((cdata[i] >> 24) & 0xff);
43                 ret[j++] = (byte) ((cdata[i] >> 16) & 0xff);
44                 ret[j++] = (byte) ((cdata[i] >> 8) & 0xff);
45                 ret[j++] = (byte) (cdata[i] & 0xff);
46            }
47            return ret;
48    }
```

(b) crypt_raw after change

Figure 3: Patch in the `crypt_raw` Method in the Commit `a40f735` of `spring-security` (the red and green background denote the deleted and added lines respectively)

patch. An integer overflow error with the maximum work factor is the major cause. In the original method, the integer `rounds` overflows when `log_rounds` is set to 31 (Line 10 in Fig. 3(a)). The patch changes `rounds` from int to long (Lne 6 in Fig. 3(b)), and includes `if` check on `rounds` and `log_rounds` (Line 8-20 in Fig. 3(b)).

We explore how V-SZZ, VerJava, V0Finder, Vuddy and MVP select the statements to generate their vulnerability signature for this method, as illustrated in Table 2. There are two signature types, *i.e.,* the vulnerable signature ("vul.") and patch signature ("pat."). "vul." is designed to match a vulnerability, while "pat." is designed to validate if the vulnerability is patched. We use line numbers from Fig. 3 to represent the composing elements of the signature content. The "vul." and "pat." signatures are composed of the lines in Fig. 3(a) and Fig. 3(b), respectively. We can observe that the five approaches select the changed statements differently. On the one hand, V-SZZ, Vuddy, and V0Finder generate vulnerability signatures by analyzing the method before change. Specifically, V-SZZ focuses on deleted lines, while Vuddy and V0Finder process the entire method before change. On the other hand, VerJava and MVP utilize both the vulnerable signature and the patch signature to detect vulnerabilities and confirm whether they have been patched. VerJava uses the changed statements, whereas MVP employs program slicing to identify the context of these changes (*e.g.,* Lines 3, 5, and 7 in the "pat." are the context generated by MVP).

However, the vulnerability signature needs to be aligned with its semantic meaning to realize precise ALVs detection. Including

Table 2: Composing Lines in the Vulnerability Signature of `crypt_raw` from `spring-security` for `CVE-2022-22976` across State-of-the-Art Approaches (numbers in the signature content represents the composing lines from Fig. 3)

Approaches	Signature Type	Signature Content
V-SZZ	vul.	{1}, {2}, {5}, {8}, {10}, {17}, {22}, {23}, {28}, {29}
VerJava	vul.	{1-2,5,8,10,17, 22-23,28-29}
	pat.	{1-2, 6, 8-20, 29,34-35, 40-41}
MVP	vul.	{1,2, 3-4,5,7,10,11,12,14,15,17,19,20,22,23,25}
	pat.	{1-2,3,5,6,7,8-20,21-28,29,30-33,34-35,36-39,40-41,42-47}
Vuddy	vul.	{1-2,3-4,5,7,8,9,10,11-16,17,18-21,22-23,24-27,28-29,30-35}
V0Finder	vul.	{1-2,3-4,5,7,8,9,10,11-16,17,18-21,22-23,24-27,28-29,30-35}

unnecessary statements would hinder the accuracy of ALVs detection, regarding both precision and recall. For instance, the patch modifies the declaration scope of global variables including `i` and `j` (Line 1 and 5 in Fig. 3(a)) into the local `for` conditions (Line 29, 34, 35 and 41 in Fig. 3(b)). The declaration of `ret` is merged with array initialization (Line 40 in Fig. 3(b)). These modifications, though present in the patch, do not directly impact the vulnerability itself.

Summary. Building on these observations, we propose to detect ALVs from the package registry (*i.e.,* Maven). Moreover, we incorporate the criticality of both methods and statements from the vulnerability patch to enhance ALVs detection accuracy.

3 APPROACH

In this section, we elaborate Vision. The overview of Vision is presented in Fig. 4. The key insight of our approach is that different methods or statements contain unequal semantic knowledge of the vulnerability. Therefore, different from existing approaches which place equal importance on the methods and statements to generate vulnerability signatures, Vision distinguishes critical methods and critical statements from these methods and encodes their criticalness into weighted inter-procedural program dependency graphs (IPDGs). By comparing similarity between IPDGs, Vision detects the affected library versions. It consists of three main modules:

- *Vulnerability and Patch Signature Generation.* Given a vulnerability-fixing commit in a GitHub repository, we identify the vulnerable version and the patched version of the repository, denoted as RP_{pre} and RP_{pos}, respectively. Vision leverages RP_{pre} to generate the vulnerability signature Sig_{vul}, and RP_{pos} to generate the patch signature Sig_{pat}. The vulnerability description is used to help select the critical methods.
- *Vulnerability-potential and Patch-potential Version Signature Generation.* Given a candidate library version from the package registries, we create vulnerability-potential signature Sig'_{vul} and patch-potential signature Sig'_{pat}. Based on the observation that defining which sets of characteristics (*e.g.,* methods, statements) in a candidate library version to match is a crucial step before signature matching, Vision carefully selects these characteristics based on the specific vulnerability. As a result, "vulnerability-potential version signature" indicates the signature is specifically customized based on the vulnerability where it will *potentially* match the corresponding vulnerability, and "patch-potential version signature" to match the corresponding patch.
- *Affected Library Version Detection.* Vision calculates the similarity between Sig_{vul} and Sig'_{vul}, Sig_{pat} and Sig'_{pat}. If the signatures of

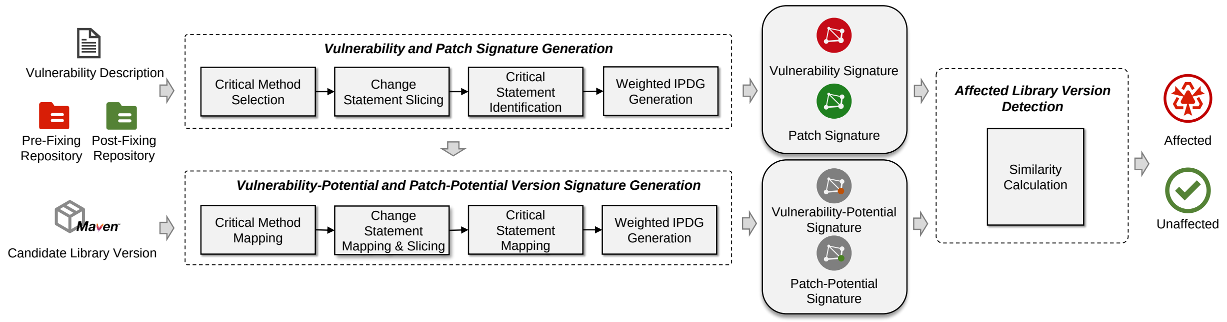

Vulnerability Description
Pre-Fixing Repository
Post-Fixing Repository
Maven
Candidate Library Version
Vulnerability and Patch Signature Generation
Critical Method Selection
Change Statement Slicing
Critical Statement Identification
Weighted IPDG Generation
Vulnerability-Potential and Patch-Potential Version Signature Generation
Critical Method Mapping
Change Statement Mapping & Slicing
Critical Statement Mapping
Weighted IPDG Generation
Vulnerability Signature
Patch Signature
Vulnerability-Potential Signature
Patch-Potential Signature
Affected Library Version Detection
Similarity Calculation
Affected
Unaffected

Figure 4: Overview of Vision

the candidate library version (*i.e.,* Sig'_{vul} or Sig'_{pat}) shows greater similarity to the vulnerability signature (Sig_{vul}) than to the patch signature (Sig_{pat}), it is classified as affected by the vulnerability; otherwise, it is considered unaffected.

3.1 Vulnerability and Patch Signature Generation

3.1.1 ***Critical Method Selection.*** Vision identifies critical methods in RP_{pre} and RP_{pos} through two-steps. First, Vision generates method reference graphs (MRGs) for the vulnerable and patched versions. Second, it identifies the critical methods within MRGs. We denote the critical changed methods as Mc_{vul} and Mc_{pat}, and the critical unchanged methods as Mu_{vul} and Mu_{pat}, in RP_{pre} and RP_{pos}, respectively.

1) Method Reference Graphs (MRGs) Generation. Solely relying on changed methods from patches without their contexts [2, 55] is insufficient. To this end, we introduce Method Reference Graphs (MRGs), which include the method calls and and mentioned methods or class names from the vulnerability description as a hybrid context to facilitate the understanding of the vulnerability and help to select the critical methods. While method calls provide the structural context of a vulnerability, they do not fully reflect its semantic meaning. Conversely, vulnerability descriptions, often contributed by experts, provide semantic insights on important classes and methods but lack contextual calling relationships. The MRGs are denoted as MRG_{vul} in RP_{pre} and MRG_{pat} in RP_{pos}. It consists of two main components: constructing call graphs for *RPs* and merging vulnerability descriptions into these call graphs.

- **Constructing Call Graphs.** Vision constructs call graphs via Joern [49] in RP_{pre} and RP_{pos}. However, the whole-program call graph include methods and call relations irrelevant to the vulnerability. To this end, Vision selects the changed methods and their callee methods, incorporating them into MRG_{vul} and MRG_{pat}, respectively.
- **Extracting Mentioned Classes and Methods in Vulnerability Description.** Vision collects vulnerability descriptions from two sources: CVE descriptions and commit descriptions. It searches for method and class names in these descriptions and matches the mentioned methods and methods within the mentioned class in MRG_{vul} (resp. MRG_{pat}). Since the actual referencer is an expert contributor, who does not appear as a valid vertex in the method reference graph, Vision creates a phantom method caller for each matched method. An extra reference relation is then established from the phantom caller to the mentioned methods, which are added to MRG_{vul} (resp. MRG_{pat}).

2) Critical Methods Selection on the MRGs. To identify critical methods, Vision utilizes the Hyperlink-Induced Topic Search (HITS) algorithm, originally developed for rating web pages [63].

In this context, the concepts of authorities and hubs are key to pinpointing critical methods. The authority score of a method is determined by the sum of the hub scores of the methods that point to it, while the hub score is based on the sum of the authority scores of the methods to which it points. These scores are calculated iteratively. In the context of vulnerabilities, authority methods are those that a malicious trigger (or sanitizer) would eventually reach, while hub methods are the triggers (or sanitizers) that are likely to be traversed. Additionally, intermediate methods lying between authority and hub methods are considered during the scoring iterations.

The authority (*auth*) and hub (*hub*) values are initialized to 1 for each method in MRG_{vul} and MRG_{pat}. The iteration process begins by calculating the *auth* and *hub* values for each method, as shown in Eq. 1 and 2. This iteration continues until convergence is achieved. In the k-th iteration, the *auth* and *hub* values for method m_i in MRG_{vul} (or MRG_{pat}) are updated according to Eq. 1 and 2. $\mathrm{Ref}(m_i, m_j)$ is 1 if there is a call relation or expert reference from m_i to m_j, and 0 otherwise. After the iteration converges, the *hub* and *auth* are normalized across different MRGs. The method is regarded as critical if the sum of its *hub* and *auth* scores exceeds a globally optimal threshold, denoted as th_{hits}.

$$\mathrm{auth}(k+1, m_i) = \sum_j \mathrm{Ref}(m_i, m_j) \cdot \mathrm{hub}(k, m_j) \tag{1}$$

$$\mathrm{hub}(k+1, m_i) = \sum_j \mathrm{Ref}(m_j, m_i) \cdot \mathrm{auth}(k, m_j) \tag{2}$$

After the critical methods are selected, we also obtain their call relations and the corresponding callsite statements. We denote them as $E^{\mathrm{call}}_{\mathrm{vul}}$ and $S^{\mathrm{call}}_{\mathrm{vul}}$ in RP_{pre}, and $E^{\mathrm{call}}_{\mathrm{pat}}$ and $S^{\mathrm{call}}_{\mathrm{pat}}$ in RP_{pos}.

3.1.2 ***Change Statement Slicing.*** Given each critical changed methods Mc_{vul} (resp. Mc_{pat}) in MRG_{vul} (resp. MRG_{pat}), Vision generates a Program Dependency Graph (PDG) using Joern [49]. It performs forward and backward program slicing on the PDG based on changed statements. Those statements who have data or control dependencies with changed statements are sliced into the partial PDG. The PDG is a 2-tuple $\langle S, E \rangle$ where S and E are collections of statements and their dependency relations. For each statement s,

$s \in S$. Each dependency relation$e \in E$ is a 2-tuple $\langle s_1, s_2 \rangle$ where s_1 and s_2 are two statements.

3.1.3 ***Critical Statement Identification.*** Vision identifies critical variables and then applies taint analysis to determine the critical statements.

1) Critical Variable Identification. A changed hunk is a block consisting of one or more contiguous added or deleted statements, denoted as a tuple $\langle S_{add}, S_{del} \rangle$, where S_{add} consists of the added statements and S_{del} consists of the deleted statements. To identify critical variables, Vision examines each critical method by processing the change hunks using `git diff`. We focus on three types:

- *Newly Introduced Variables in Added Hunks ($S_{del} = \emptyset \cap S_{add} \neq \emptyset$).* These variables are likely to play a role in fixing the vulnerability. Vision identifies them by analyzing the sub Abstract Syntax Tree (AST) of the added hunks to extract the newly introduced variables.
- *Removed Variables in Deleted Hunks ($S_{del} \neq \emptyset \cap S_{add} = \emptyset$).* Variables that were present in RP_{pre} but are removed in RP_{pos} may be indicative of vulnerable code. Vision identifies these variables by analyzing the sub AST of of the deleted hunks.
- *Newly Introduced and Removed Variables in Modified Hunks* ($S_{del} \neq \emptyset \cap S_{add} \neq \emptyset$). Variables that are altered due to changes in the code, such as modifications in method parameters or conditional statements, may be critical. Vision compares the sets of variables before and after the change, identifying the critical variables as those that differ between the two versions.

2) Taint Analysis. Once the critical variables are identified, Vision applies taint analysis to trace the flows from these variables to the method's entry point and exit points and identify critical statements that in the flows.

- *Backward Taint Analysis.* Vision traces critical statements from each critical variable backward to the method's entry point. This process reveals the statements that influence the value of the critical variables, marking them as critical.
- *Forward Taint Analysis.* Vision traces critical statements from each critical variable forward to the method's exit points (*i.e.,* `return`, `throw`, `except`, `assert`, and `catch` statement). This traces how the variables impact other statements.

Finally, the critical statements and their dependency relations are denoted as $S^{\mathrm{taint}}_{\mathrm{vul}}$, and $E^{\mathrm{taint}}_{\mathrm{vul}}$ in RP_{pre}, and $S^{\mathrm{taint}}_{\mathrm{pat}}$, and $E^{\mathrm{taint}}_{\mathrm{pat}}$ in RP_{pos}.

3.1.4 ***Signature Generation.*** The signature is a weighted inter-procedural PDG (weighted IPDG), denoted as a 3-tuple $\langle S, E, W \rangle$, where S and E are the collections of statements and their dependency relations, respectively, and each weight entry $w_o \in W$ maps a statement and its dependency relations to its weight value. The signature generation for Sig_{vul} (resp. Sig_{pat}) is as follows:

- **Connecting PDGs into IPDGs.** Vision generates PDGs for critical methods. For critical methods with changes, Vision uses the partial PDGs from of Mc_{vul} (resp. Mc_{pat}) that are generated in *change statement slicing* (see Section 3.1.2). For critical unchanged methods (*i.e., Mu_{vul}* (resp. Mu_{pat})), Vision collapses their PDGs into single method nodes where internal program dependency relations are omitted. Then, the partial PDGs and single method nodes are connected using inter-procedural call relations from call graphs generated in *critical method selection* (see Section 3.1.1). Specifically, the inter-procedural call relation starts from the call site statement in the PDG of the caller method to the PDG entry of the callee method. All inter-procedural call relations starting from statements in a single method node are collapsed into one startpoint. As a result, the connected PDGs are Inter-procedural PDGs (IPDGs).
- **Assigning Weights to IPDGs.** Each vertex and edge in the IPDG is assigned weight values of 1 by default. Then, Vision assigns higher weights using critical methods and critical statements. First, Vision assigns w_{cri_m} to callsite statement $S^{\mathrm{call}}_{\mathrm{vul}}$ (resp. $S^{\mathrm{call}}_{\mathrm{pat}}$) and call relations $E^{\mathrm{call}}_{\mathrm{vul}}$ (resp. $E^{\mathrm{call}}_{\mathrm{pat}}$) that belongs to the identified critical methods (see Section 3.1.1). Second, Vision assigns w_{cri_s} to the critical statements $S^{\mathrm{taint}}_{\mathrm{vul}}$ (resp. $S^{\mathrm{taint}}_{\mathrm{pat}}$) and their dependency relations $E^{\mathrm{taint}}_{\mathrm{vul}}$ (resp. $E^{\mathrm{taint}}_{\mathrm{pat}}$).

Finally, the vulnerability signature Sig_{vul} (resp. Sig_{pat}) is generated, represented as a weighted IPDG.

3.2 Vulnerability-potential and Patch-potential Version Signature Generation

3.2.1 ***Critical Method Mapping.*** Taking input from the candidate library version, Vision first decompiles the library version using Java Decompiler [29]. Then, Vision employs NiCad [7], a popular code clone detection tool, to identify methods in the candidate library version that are similar to those in RP_{pre} (and RP_{pos}). Instead of directly comparing method signatures, we use clone detection to handle more complex refactorings. The similar methods found in the candidate library version provide evidence that potentially contributes to the identification of the vulnerability.

3.2.2 ***Change Statement Mapping & Slicing.*** Vision computes the similarity between the statements of the original method and the mapped critical method. To handle potential syntactic differences caused by decompilation, Vision conducts normalization on both ends, which includes reordering operands, transforming into default operands (e.g., transforming ">" to "<") in *IfStatement*, and unifying conditional expressions in *ForStatement*. Then, Vision calculates the syntactic similarity using Levenshtein distance [64] between the statements on both sides. If the similarity exceeds the threshold th_{ld}, Vision obtains a mapping statement pair. We set the default threshold as 0.55 following previous work [16]. After that, a mapping set of delete statements and a mapping set of add statements is constructed. Next, Vision performs forward and backward program slicing on them.

3.2.3 ***Critical Statement Mapping.*** Next, Vision identifies the statements from the original method containing the critical variable. Based on the identified statements, Vision constructs the PDG and performs taint analysis on the mapped variable, which forms the collection of the critical statements in the original method.

3.2.4 ***Signature Generation.*** The signature generation is similar to the vulnerability and patch signature generation in Section 3.1.4. Using the mapped critical methods (see Section 3.2.2) and critical statements (see Section 3.2.3), Vision connects PDGs into inter-procedural PDGs and assigns weights to their statements and dependency relations. The vulnerability-potential and patch-potential signature are denoted as Sig'_{vul} and Sig'_{pat}, respectively.

3.3 Affected Library Versions Detection

3.3.1 ***Similarity Calculation.*** Vision compares the similarity of the weighted IPDGs between Sig_{vul} and Sig'_{vul}, Sig_{pat} and Sig'_{pat} to identify ALVs. First, Vision leverages UniXcoder [23], which is a unified cross-modal pre-trained model for programming language. UniXcoder helps to generate the semantic embeddings for the textual representation. Taking input as a statement or statement set (in cases where methods were merged into a single node) s from the vertex set V of the original signature Sig_{vul} or Sig_{pat}, Vision generates the semantic embedding vector $vec(s)$, normalized by L2 norm.

Second, given s_i from the original weighted IPDG and s_j from the mapped weighted IPDG, Vision calculates their cosine similarity (*i.e.,* $\text{vec}(s_i) \cdot \text{vec}(s_j)$). Therefore, the distance in replacing from s_i to s_j is based on their similarity and the weights of both nodes, denoted as w_{s_i} and w_{s_j}, which presented in Eq. 3.

$$d(s_i, s_j) = (1 - \text{vec}(s_i) \cdot \text{vec}(s_j)) \times w_{s_i} \times w_{s_j} \tag{3}$$

Meanwhile, the edge similarity of e_i from the original weighted IPDG and e_j from the mapped weighted IPDG, is calculated by averaging the node distances between the source nodes ($e_i.s_1$ and $e_j.s_1$) and the destination nodes ($e_j.s_2$ and $e_j.s_2$), presented in Eq. 4.

$$\text{d}(e_i, e_j) = \frac{c(e_i.s_1, e_j.s_1) + c(e_i.s_2, e_j.s_2)}{2} \tag{4}$$

Third, the problem then becomes to minimize the total distance so that the mapping statements and edges are maximized, which is a bipartite matching problem. We utilize the previous work [10] to find the optimal solution where the minimal statement distance and edge distance between Sig_a and Sig_b is denoted by $\text{sDis}(Sig_a, Sig_b)$ and $\text{eDis}(Sig_a, Sig_b)$, respectively. Afterwards, we calculate the similarity between Sig_a and Sig_b, which is presented in Eq. 5.

$$\text{sim}(Sig_a, Sig_b) = 1 - \left(\frac{\text{sDis}(Sig_a, Sig_b) + \sqrt{\text{eDis}(Sig_a, Sig_b)}}{\mid Sig_a.S \mid + \mid Sig_b.S \mid} \right) \tag{5}$$

Finally, Vision generates the vulnerability similarity score sim_v between Sig_{vul} and Sig'_{vul}, as well as the patch similarity score sim_p between Sig_{pat} and Sig'_{pat}. To be considered an ALV, it must satisfy $sim_v > th_{vul}$ and $th_{pat} > sim_p$. Additionally, an ALV should be more similar to the vulnerability signature than the patch signature, meaning $th_{vul} > th_{pat}$. Therefore, we use a single threshold th_s, reconstruct our formula to $sim_v > th_s \,\&\, sim_v > sim_p$. Vision determines a candidate library version as affected if this condition is met. Otherwise, Vision determines the library version as unaffected.

4 EVALUATION

We implemented Vision using 11.4K lines of Python code and 1.6K lines of Java code. We design the following research questions.

- **RQ1: Effectiveness Evaluation.** How effective is Vision in identifying ALVs compared to state-of-the-art methods?
- **RQ2: Ablation Study.** How is the contribution of each component to the overall effectiveness of Vision?
- **RQ3: Parameter Sensitivity.** How do the parameters affect the effectiveness of Vision?

Table 3: Results of Our Effectiveness Evaluation

Dataset	Tools	#Ver.	#V.	PV.	TP	FP	FN	Pre.	Rec.
O.	V-SZZ	11,256	102	26	2,310	1,401	1,640	0.62	0.58
	VerJava	11,256	102	29	2,283	660	1,667	0.78	0.58
	Vision $^-$	11,256	102	**66**	**3,766**	**357**	**184**	**0.91**	**0.95**
C.	VerJava*	12,073	102	4	1,721	1,720	2,961	0.50	0.37
	V0Finder	12,073	102	16	1,159	**267**	3,523	0.81	0.25
	MVP	12,073	102	8	670	308	4,012	0.69	0.14
	Vuddy	12,073	102	11	738	283	3,944	0.72	0.16
	Vision	12,073	102	**62**	**4,424**	418	**258**	**0.91**	**0.94**

- **RQ4: Generality Evaluation.** How is the generality of Vision when applying Vision in other datasets?
- **RQ5: Efficiency Evaluation.** How is Vision's time overhead?
- **RQ6: Usefulness Evaluation.** How is the usefulness of Vision?

4.1 Evaluation Setup

Ground Truth. We choose Java libraries in Maven due to their popularity and complexity. The ground truth is established through a rigorous four-step examination process:

- *Collecting Vulnerabilities.* We collected CVEs with their patch (e.g., GitHub commits) in their references from January 1999 to May 2024 from NVD [41]. It covered existing datasets of V-SZZ and VerJava [2, 55]. We obtained 1,083 CVEs with its patch.
- *Collecting Test Cases/Proof-of-Concepts and Creating Test Cases.* We manually inspected test cases included in the patches. If not present, we searched for patch commits in GitHub repositories using CVE IDs and websites containing vulnerability "exploits" or "proof-of-concepts (PoCs)". We analyzed their triggering logic and transformed them into JUnit test cases. We obtained 102 CVEs with test cases, each with an assertion of the trigger status.
- *Running Test Cases across Library Versions.* We automatically changed the library versions from Maven and executed the test cases to verify the triggering status of the vulnerabilities. We manually inspected unsuccessful test case runs and modified the code for compatibility with the corresponding version.
- *Manual Inspection.* We conducted manual inspections on the remaining library versions that were not successfully executed by the test cases. We determined if the vulnerability existed by checking the presence of vulnerable or fixing methods and statements in the remaining library versions.

Two of the authors constructed the ground truth, each with over 5 years of experience in software security. We measured their agreement using Cohen's Kappa coefficient, which reached 0.879 for inspecting vulnerability-affected versions. A third author was involved in resolving disagreements. As a result, we collected 12,073 library versions corresponding to 102 CVEs within 79 libraries.

Baselines. We compare Vision with *patch-based approaches.* We obtained V-SZZ [2] and reproduced VerJava [55], which are state-of-the-art tools for identifying ALVs. We did not compare AFV [48] because it targets specific vulnerabilities in PHP. We also compare Vision with *clone-based approaches.* We obtained V0Finder [67] and VUDDY [31], and reproduced MVP [70]. We modified VUDDY to make it compatible for Java. We did not compare V1Scan [65] and Movery [66] because they depend on ranges of vulnerable versions to generate an aggregated signature, which would unfairly hinder their effectiveness if only one vulnerable version is provided.

Table 4: Results of Our Effectiveness Evaluation w.r.t CWE Types (#V. denotes the number of vulnerabilities of a CWE type)

Dataset	Tools		CWE-707	CWE-664	CWE-693	CWE-682	CWE-284	CWE-435	CWE-703	CWE-691	CWE-339	CWE-19	CWE-417	CWE-264	OTHERS
			#V.=23	#V.=34	#V.=1	#V.=2	#V.=5	#V.=3	#V.=3	#V.=8	#V.=1	#V.=1	#V.=1	#V.=1	#V.=19
O.	V-SZZ	Pre.	0.50	0.65	0.00	0.41	0.37	0.79	**1.00**	0.64	**1.00**	**1.00**	0.00	**1.00**	0.81
		Rec.	0.86	0.49	0.00	0.86	0.53	0.57	0.26	0.59	0.95	**0.62**	0.00	0.75	0.70
	VerJava	Pre.	**0.92**	0.86	0.51	0.46	0.26	**1.00**	**1.00**	0.74	**1.00**	**1.00**	0.30	0.00	**0.97**
		Rec.	0.46	0.60	**0.95**	0.99	0.47	0.67	**1.00**	0.84	0.86	**0.62**	**1.00**	0.00	0.41
	Vision $^{-}$	Pre.	0.82	**0.98**	**0.64**	**1.00**	**1.00**	0.99	**1.00**	**0.93**	0.57	**1.00**	**1.00**	0.50	0.85
		Rec.	**0.93**	**0.95**	0.86	**1.00**	**1.00**	**0.99**	**1.00**	**0.98**	**1.00**	**0.62**	**1.00**	**1.00**	**0.96**
C.	VerJava*	Pre.	0.56	0.59	0.51	0.44	0.26	0.70	0.87	0.32	0.66	0.00	0.16	0.00	0.33
		Rec.	0.45	0.36	**1.00**	0.99	0.48	0.33	0.74	0.36	0.94	0.00	**1.00**	0.00	0.12
	V0Finder	Pre.	0.84	0.96	0.46	0.38	**1.00**	0.00	**1.00**	0.93	**1.00**	0.00	0.97	**1.00**	**0.90**
		Rec.	0.25	0.16	0.87	0.70	0.28	0.00	0.74	0.19	0.94	0.00	**1.00**	0.11	0.36
	MVP	Pre.	0.91	0.70	0.00	0.28	**1.00**	0.22	0.93	**1.00**	**1.00**	**1.00**	0.00	0.64	0.36
		Rec.	0.25	0.10	0.00	0.70	0.43	0.04	**1.00**	0.19	0.94	0.06	0.00	**1.00**	0.01
	Vuddy	Pre.	**1.00**	0.94	0.00	0.28	0.00	0.00	**1.00**	0.00	**1.00**	0.00	0.00	0.00	0.75
		Rec.	0.17	0.06	0.00	0.70	0.00	0.00	0.55	0.00	0.55	0.00	0.00	0.00	0.44
	Vision	Pre.	0.84	**0.97**	**0.65**	**1.00**	**1.00**	**0.99**	**1.00**	0.77	0.67	**1.00**	**1.00**	0.64	0.87
		Rec.	**0.93**	**0.95**	0.86	**1.00**	**1.00**	**1.00**	**1.00**	**0.98**	**1.00**	**0.62**	**1.00**	**1.00**	**0.90**

Metrics and Environment. We evaluate the approaches using true positive (TP), false positive (FP), false negative (FN), precision and recall, as used in previous studies [2, 55]. Additionally, we introduce the *perfectly identified vulnerability* (PV.) to measure the number of vulnerabilities where the evaluated tool identifies the ALVs completely and correctly (*i.e.,* zero FP and zero FN).

4.2 Effectiveness Evaluation (RQ1)

RQ1 Setup. We assessed Vision using our ground truth and compared its performance with baseline approaches. Since V-SZZ and VerJava accept library versions from GitHub repositories, we evaluated these approaches on the overlapping library versions (denoted as O.) between GitHub repositories and our complete ground truth dataset (denoted as C.). We also evaluate our approach on the overlapping library versions, denoted as Vision $^{-}$. Besides, we modified VerJava to analyze complete library versions from Maven, which is denoted as VerJava*. Additionally, we observed the generalization of Vision and baseline approaches w.r.t CWE types. Our ground truth includes 83 vulnerabilities encompassing 36 distinct CWE types, while 19 vulnerabilities have not been labeled with a CWE. We categorized the CWEs into 12 groups based on their common ancestors following the classifications in [11, 12]. The 19 vulnerabilities without CWEs are collectively classified under a single group (*i.e.,* Others). Furthermore, we investigated how Vision and baseline approaches perform under two change types, including hybrid changes and solely additions. Notably, our ground truth does not contain any changes with solely deletions. We used optimal parameters for Vision according to our sensitivity analysis (see Sec. 4.4).

Overall Result. Table 3 presents the results of our effectiveness evaluation compared with five baseline approaches. Vision reports 357 false positives and 184 false negatives in the overlapping library versions, and 418 false positives and 258 false negatives in the complete ground truth. Comparatively, patch-based approaches averagely report 1,031 false positives and 1,654 false negatives, 1,720 false positives and 2,961 false negatives. Clone-based approaches averagely report 286 false positves and 3,826 false negatives in the complete ground truth. Vision achieves the highest precision of 0.91 and recall of 0.94 in the complete library versions and the highest precision of 0.91 and recall of 0.95 in the overlapping library versions. Comparatively, in the overlapping library versions, the state-of-the-art VerJava achieves a precision of 0.78 and a recall of 0.58. Vision $^{-}$ surpasses VerJava with an improvement in precision by 0.13 (16.7%) and in recall by 0.37 (63.8%). In the complete library versions, the state-of-the-art V0Finder achieves a precision of 0.81 and a recall of 0.25. Vision outperforms V0Finder with an increase in precision by 0.10 (12.0%) and in recall by 0.69 (276.0%). Moreover, Vision identifies 66 (64.7%) PVs in the overlapping library versions, surpassing VerJava with 37 (36.3%) PVs. In the complete library versions, Vision identifies 62 (60.8%) PVs, outperforming V0Finder with 46 (287.5%) PVs.

Vision generates 357 FPs and 184 FNs, resulting in 56 vulnerabilities which are not PVs in the overlapping library versions. It generates 418 FPs and 258 FNs, resulting 40 vulnerabilities which are not PVs in the complete library versions. We summarize three major reasons. First, the CVE or commit descriptions may contain irrelevant methods, which can mislead our critical method selection. Second, Vision leverages program slicing based on Joern, which contains incorrect data or control dependencies that can mislead our critical statement selection. Third, the library versions can have different fixing logics because they evolve on different branches, which may not be reflected in patch-potential signatures.

Effectiveness w.r.t CWE Types. Table 4 presents the effectiveness of Vision and the baselines with respect to CWE types. In the overlapping library versions, Vision outperforms the state-of-the-art VerJava in 8 of the 13 CWE types regarding precision and in 12 of the 13 CWE types regarding recall. It achieves 100% precision or recall in 6 CWE types. On average, Vision outperforms the state-of-the-art VerJava with a precision of 0.87 and a recall of 0.95 among the 13 CWE types. In the complete library versions, Vision outperforms the state-of-the-art Vuddy in 8 of the 13 CWE types regarding precision and in 12 of the 13 CWE types regarding recall. On average, Vision outperforms the state-of-the-art Vuddy with a precision of 0.88 and a recall of 0.94 among the 13 types.

Effectiveness w.r.t Changed Methods. Table 5 illustrates the effectiveness of Vision and the baselines with respect to the number of changed methods. We observe that as the number of changed methods increases, the effectiveness of Vision remains stable. The precision of Vision ranges from 0.84 to 0.97 in the overlapping library versions and from 0.86 to 0.95 in the complete library versions. The recall of Vision ranges from 0.92 to 0.95 in the overlapping

Table 5: Results of Our Effectiveness Evaluation w.r.t the Number of Changed Methods (# CM.)

# CM.	#V.	Dataset	Tools	PV.	TP	FP	FN	Pre.	Rec.
1	52	O.	V-SZZ	17	1,075	666	967	0.62	0.53
			VerJava	20	985	331	1,057	0.75	0.48
			Vision $^{-}$	**35**	**1,942**	**59**	**100**	**0.97**	**0.95**
		C.	VerJava*	2	996	1,148	1,518	0.46	0.40
			V0Finder	7	313	11	2,201	**0.97**	0.12
			MVP	2	94	**8**	2,420	0.92	0.04
			Vuddy	5	275	15	2,239	0.95	0.11
			Vision	**32**	**2,397**	113	**117**	0.95	**0.95**
2-5	33	O.	V-SZZ	4	861	421	423	0.67	0.67
			VerJava	4	944	**135**	340	**0.87**	0.74
			Vision $^{-}$	**19**	**1,248**	238	**36**	0.84	**0.97**
		C.	VerJava*	2	511	298	1,027	0.63	0.33
			V0Finder	5	470	**3**	1,068	0.99	0.31
			MVP	3	341	155	1,197	0.69	0.22
			Vuddy	3	226	8	1,312	**0.97**	0.15
			Vision	**18**	**1,445**	245	**93**	0.86	**0.94**
>5	17	O.	V-SZZ	5	374	314	250	0.54	0.60
			VerJava	5	354	194	270	0.65	0.57
			Vision $^{-}$	**12**	**576**	**60**	**48**	**0.91**	**0.92**
		C.	VerJava*	0	214	274	416	0.44	0.34
			V0Finder	4	376	253	254	0.60	0.60
			MVP	3	235	145	395	0.62	0.37
			Vuddy	3	237	260	393	0.48	0.38
			Vision	**12**	**582**	**60**	**48**	**0.91**	**0.92**

library versions and from 0.92 to 0.95 in the complete library versions. In terms of precision, Vision outperforms the state-of-the-art tools in the overlapping library versions. However, it has slightly lower precision than Vuddy in the complete library versions, with a difference of 0.02 for vulnerabilities with one changed method and 0.13 for vulnerabilities with 2 to 5 changed methods. Nevertheless, as the number of changed methods increases, Vision regains its leading advantage. In terms of recall, Vision outperforms the state-of-the-art tools in both overlapping and complete library versions.

Effectiveness w.r.t Changed Types. Table 6 presents the effectiveness of Vision and the baselines with respect to the changed types. Overall, Vision consistently outperforms the baseline approaches in both the overlapping library versions and the complete versions for hybrid changes and some additions. Notably, V-SZZ achieves a precision of 0.62 and a recall of 0.78 for hybrid changes, but V-SZZ does not support solely additions and fails. Nevertheless, Vision outperforms V-SZZ in hybrid changes, achieving a 0.27 higher precision and a 0.16 higher recall.

Summary: Vision achieves the highest precision and recall in both complete and overlapping library versions with 0.91 and 0.94, 0.91 and 0.95, respectively, averagely surpassing the state-of-the-arts by 0.23 (33.8%) in precision and 0.71 (308.7%) in recall. Vision identifies averagely 3,352 (312.7%) more true ALVs than state-of-the-arts. Vision is effectiveacross different CWE types, changed method numbers, and change types.

4.3 Ablation Study (RQ2)

RQ2 Setup. We created five ablated versions of Vision: (a) keeping only those critical methods identified from expert references, thereby ablating the call relations (w/o CR); (b) keeping call relations as critical methods without expert references (w/o EF); (c) ablating the critical method selection and using changed methods instead (w/ CM); (d) setting the weight to 1 to mark the critical statements

Table 6: Results of Our Effectiveness Evaluation w.r.t Change Types

Dataset	Tools	Hybrid Changes (#V.= 81) TP	FP	FN	Pre.	Rec.	Solely Additions (#V.=21) TP	FP	FN	Pre.	Rec.
O.	V-SZZ	2,310	1,401	639	0.62	0.78	0	**0**	1,001	0.00	0.00
	VerJava	1,445	553	1,504	0.72	0.49	838	107	163	0.89	0.84
	Vision $^{-}$	**2,766**	**328**	**183**	**0.89**	**0.94**	**1,000**	29	**1**	**0.97**	**1.00**
C.	VerJava*	783	1,139	2,676	0.41	0.23	938	581	285	0.62	0.77
	V0Finder	996	**266**	2,463	0.79	0.29	163	1	1,060	0.99	0.13
	MVP	577	308	2,882	0.65	0.17	93	**0**	1,130	**1.00**	0.08
	Vuddy	610	268	2,849	0.69	0.18	128	15	1,095	0.90	0.10
	Vision	**3,206**	357	**253**	**0.90**	**0.93**	**1,218**	61	**5**	0.95	**1.00**

Table 7: Results of Our Ablation Evaluation

	w/o CR	w/o EF	w/ CM	w/o CS	w/ LD	V0Finder w/ CMS	MVP w/ CMS	Vuddy w/ CMS
Pre.	0.81	0.90	0.87	0.85	0.86	0.91	0.87	0.88
ΔPre.	-0.10	-0.02	-0.04	-0.07	-0.05	+0.10	+0.18	+0.16
Rec.	0.93	0.88	0.84	0.80	0.84	0.21	0.12	0.13
ΔRec.	-0.02	-0.07	-0.11	-0.15	-0.11	-0.04	-0.02	-0.03

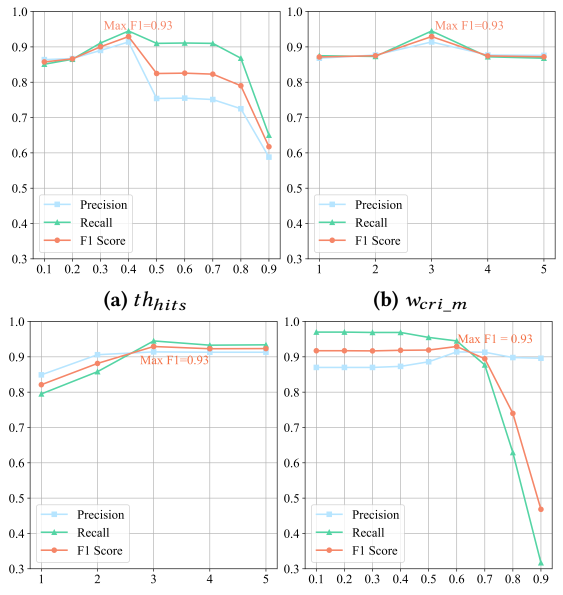
as normal statements (w/o CS); and (e) replacing UniXcoder in similarity calculation with Levenshtein distance [64] (w/ LD). Besides, we applied critical method selection in clone-based approaches. *i.e.,* V0Finder w/ CMS, MVP w/ CMS, and Vuddy w/ CMS.

Overall Results. Table 7 presents the results of our ablation study. Overall, the precision and recall decrease across the five ablated versions. Specifically, Vision without critical statements (w/o CS) exhibits the most significant recall drop of 0.15 and the second-largest precision drop of 0.07, underscoring the importance of selecting critical statements. Meanwhile, Vision with changed methods (w/ CM) experiences a recall drop of 0.11 and a precision drop of 0.04, highlighting the importance of critical method selection. Additionally, removing call relations (w/o CR) or expert references (w/o EF) results in decreased precision and recall, indicating that both call relations and expert references contribute to Vision's effectiveness. We also observe a decrease in precision and recall when UniXcoder is replaced with Levenshtein distance (w/ LD). Besides, using our critical method selection, V0Finder, MVP and Vuddy undergo significant increases in precision by 0.10, 0.18 and 0.16 and slight drops in recall by 0.04, 0.02 and 0.03, respectively.

Summary: Removing any component of Vision results in noticeable drops in precision and recall. Ablating critical statements causes the largest recall drop of 0.15, while removing call relations leads to the largest precision drop of 0.10. Using our critical method selection, clone-based methods undergo significant increases of precision by 0.15 on average with slight drops of recall by 0.03 on average.

4.4 Parameter Sensitivity (RQ3)

RQ3 Setup. Four parameters are configurable in Vision, including the threshold for selecting critical methods (th_{hits}), the weight for inter-procedural calls between the statements from critical methods (w_{cri_m}), the weight for the critical statements (w_{cri_s}), and the similarity for threshold for identifying Vulnerabilities (th_s). The default parameter is 0.4, 3, 3 and 0.6. To evaluate the sensitivity of these thresholds to Vision's accuracy, we reconfigured one parameter and fixed the other three against our dataset.



(a) th_{hits} (b) w_{cri_m} (c) w_{cri_s} (d) th_s

Figure 5: Results of Our Sensitivity Analysis

Overall Result. Fig. 5 illustrates the results of our sensitivity analysis. It achieves the optimal performance when th_{hits} is set to 0.4, w_{cri_m} and w_{cri_s} are both set to 3, and th_s is set to 0.6.

Summary: VISION performs best with th_{hits} set to 0.4, w_{cri_m} and w_{cri_s} set to 3, and th_s set to 0.6.

4.5 Generality Evaluation (RQ4)

RQ4 Setup. We applied VISION to the original datasets of V-SZZ and VERJAVA. We selected 50% (41/81) of the CVEs from the overlapping set between V-SZZ and VERJAVA, resulting in a total of 3,381 complete (C.) and 3,078 overlapping library versions (O.).

Overall Results. Table 8 presents the results of our generality evaluation. Compared to the results from our effectiveness evaluation (see Section 4.2), VISION, VISION $^-$, MVP, V0FINDER, and VUDDY produce similar outcomes. In contrast, V-SZZ, VERJAVA, and VERJAVA* achieve higher precision and recall in their own datasets. Specifically, VISION undergoes a slight drop in precision by 1% and recall by 2%. VISION $^-$ has a slight increase in precision by 2% and a slight drop in recall by 1%. It demonstrates the generality of VISION in external datasets. Compared to state-of-the-arts, VISION $^-$ outperforms VERJAVA, achieving a 0.11 (13.41%) increase in precision and 0.35 (59.32%) increase in recall. Additionally, VISION $^-$ identifies 29 (70.73%) perfectly identified vulnerabilities (PVs) in the overlapping library versions, surpassing VERJAVA with 16 (123.08%) PVs.

Summary: VISION's precision and recall drop slightly in the overlapping dataset of V-SZZ and VERJAVA. Besides, V0FINDER, MVP and VUDDY undergo significant increases in precision by 10%, 18% and 16% and slight drops in recall by 4%, 2% and 3%. The results demonstrate the generality of VISION.

4.6 Efficiency Evaluation (RQ5)

RQ5 Setup. We measured the average time taken to identify ALVs for each vulnerability as well as for each library.

Table 8: Results of Our Generality Evaluation in the Overlapping Dataset of V-SZZ and VERJAVA

Dataset	Tools	#Ver.	#V.	PV.	TP	FP	FN	Pre.	Rec.
O.	V-SZZ	3,078	41	9	768	282	445	0.73	0.63
	VERJAVA	3,078	41	13	716	157	497	0.82	0.59
	VISION $^-$	3,078	41	**29**	**1,140**	**86**	**73**	**0.93**	**0.94**
C.	VERJAVA*	3,381	41	10	697	382	587	0.65	0.54
	V0FINDER	3,381	41	1	338	100	946	0.77	0.26
	MVP	3,381	41	1	257	94	1,027	0.73	0.20
	VUDDY	3,381	41	1	141	49	1,143	0.74	0.11
	VISION	3,381	41	**27**	**1,179**	**128**	**105**	**0.90**	**0.92**

Table 9: Results of Our Efficiency Evaluation

Time (s)	V-SZZ	VERJAVA	VERJAVA*	V0FINDER	MVP	VUDDY	VISION
Per L.	29.35	1.73	**1.42**	425.06	17771.16	72.57	1094.12
Per V.	4.09	0.01	**0.01**	3.84	186.93	1.92	9.24

Table 10: Results of Usefulness Evaluation among Vulnerability Databases (Δ denotes the accuracy gains of VISION over the corresponding database)

Tools	#V.	PV/Δ	TP/Δ	FP/Δ	FN/Δ	Pre./Δ	Rec./Δ
NVD	102	34/+27	3,745/+677	1,410/-992	937/-677	0.73/+0.19	0.80/+0.14
VERACODE	97	34/+24	3,853/+522	596/-178	768/-522	0.87/+0.05	0.83/+0.11
GITHUB	95	35/+22	4,085/+246	1,777/-1,388	478/-246	0.70/+0.22	0.90/+0.05
GITLAB	93	27/+27	3,569/+563	1,639/-1,223	795/-563	0.69/+0.22	0.82/+0.13
SNYK	97	37/+19	3,988/+323	1,182/-764	583/-323	0.77/+0.14	0.87/+0.07

Overall Result. Table 9 presents the time cost of VISION compared to the other five tools. On average, VISION takes 9.24 seconds to process a candidate library version and determine if it is affected, and 1,094.11 seconds to process all library versions for a vulnerability. The time cost is higher than existing tools, primarily due to decompiling jars in critical method mapping, generating call graphs in critical method selection, and similarity calculation. However, we believe it is acceptable given VISION's superior effectiveness in identifying vulnerability-affected versions compared to other tools.

Summary: VISION takes an average of 9.24s to identify an ALV and 1,094.11s to identify all ALVs for a vulnerability.

4.7 Usefulness Evaluation (RQ6)

RQ6 Setup. Following previous work [68], we compare VISION with five vulnerability databases NVD [41], VERACODE [59], GITHUB [19], GITLAB [22], and SNYK [51]. We exclude any unknown CVEs and library versions absent from our ground truth. We measured the accuracy of the five databases using the same metrics as in RQ1 in Sec. 4.2. Additionally, we evaluate the accuracy of VISION using the same set of vulnerabilities in each database and calculate the accuracy gains of VISION over the corresponding databases.

Overall Result. Table 10 lists the results of our usefulness evaluation. VISION improves precision and recall across all five databases. It achieves a maximum precision increase of 0.22 in both GITHUB and GITLAB, and a maximum recall improvement of 0.14 over NVD. Further, it helps to identify more *perfectly identified vulnerabilities* (PVs), with an increase of at least 19 in SNYK and up to 27 in NVD.

Furthermore, we select inaccurate vulnerabilities identified by VISION from our ground truth and the dataset from V-SZZ and VERJAVA. We reported 52, 31 and 46 incorrectly labeled vulnerabilities to NVD, VERACODE and SNYK via email, and 39 and 42 to GitHub and GitLab by creating issues. By now, GITHUB and GITLAB have resolved all incorrect affected library versions. NVD have fixed incorrect affected versions in 8 vulnerabilities while the rest are under review. Notably, GitLab has expressed interest in our tool for ALV detection.

Summary: Vision improves the accuracy of ALVs across five vulnerability databases. We reported incorrect ALVs to the databases, resulting in fixes for 39, 42 and 8 vulnerabilities by GitHub, GitLab and NVD, respectively. These results highlight the practical usefulness of Vision in assisting security teams to identify affected library versions and enhance the quality of vulnerability databases.

4.8 Discussion

Threats. One primary threat to our evaluation is the construction of ground truth. While we execute Proof-of-Concepts and validate affected library versions as thoroughly as possible, a portion of the dataset is constructed through manual confirmation, which may introduce human error. We mitigate this threat by involving three of the authors in constructing the ground truth. Besides, the evaluated results are influenced by the dataset's size. We build a dataset of 102 vulnerabilities and 12,073 affected and unaffected library versions, requiring 800 man-hours, which is the largest databset.

Limitations. First, Vision relies on several language-dependent tools such as Java Decompiler and Joern, limiting its applicability to other programming languages. Second, Vision lacks explanations for the predicted affected library versions. We intend to address this limitation by adding interpretations with visualized vulnerability causes and fixes, based on critical methods and statements. Third, Vision takes a patch commit as an input but the target patch may contain multiple commits, leading to the overlooking of the vulnerability signature and patch signature. This can result in both false positives and false negatives.

5 RELATED WORK

Affected Library Version Identification. It is a crucial task to identify library versions affected by an open source software vulnerability. Various approaches have been proposed to automatically achieve this task [2, 13, 14, 27, 38, 48, 55]. Specifically, Dong et al. [14] adopted named entity recognition (NER) to identify affected library versions in vulnerability reports. Instead of relying on vulnerability reports, several approaches [2, 27, 38, 55] proposed to use vulnerability patches. Nguyen et al. [38] detected vulnerable versions on source code repository by the original SZZ algorithm [50], which assumed that the last modification of the deleted lines in the patch introduced the vulnerability. Bao et al. [2] improved the original SZZ algorithm by tracing back the deleted lines in the patch to determine the vulnerability-inducing versions. These SZZ-based approaches fail when the patch only contains added lines. Sun et al. [55] identified affected versions by measuring the presence of deleted lines and the non-presence of added lines in the patch in target versions. He et al. [27] leveraged developer logs and patches to identify affected versions. However, this approach heavily relies on the completeness and accuracy of developer logs that contain vulnerability fixing information, and requires manual verification, which reduces the cost-efficiency. Patch-based approaches fail to consider the context of modified lines in the patch, leading to inaccuracies. Vision is also patch-based, and it captures the context for an accurate version identification. Shi et al. [48] also proposed a patch-based approach but targeted Web vulnerabilities. Dai et al. [13] utilized Proof of Concept (PoC), and adopted directed fuzzing to identify the affected versions but the vulnerability may lack a PoC [35, 55], limiting the practical effectiveness.

Vulnerable Code Clone Detection. Code similarity analysis is commonly used in detecting vulnerable code clones (VCCs, or recurring vulnerabilities) across different libraries based on syntactic or semantic signatures, which can also be adapted to affected version identification. For example, Kim et al. [31] introduced VUDDY to detect VCCs based on abstraction and normalization of vulnerable functions. Xiao et al. [70] proposed MVP, which used program slicing to generate a vulnerability signature from the vulnerable function and a patch signature from the patched function. Woo et al. developed Movery [66] and V1Scan [65], which leveraged all vulnerable versions to overcome the syntax diversity of vulnerable functions. These two approaches also motivate the necessity to identify all affected versions. Woo et al. [67] designed V0Finder to discover the software where a vulnerability first originated. All these VCC detection approaches can generate signatures that contain code that is irrelevant to the vulnerability and the patch, posing a significant challenge in generating precise signatures. Moreover, all these approaches can over-abstract identifiers or statements into a same symbol, potentially making vulnerable and non-vulnerable code difficult to distinguish and leading to false positives.

Vulnerability Knowledge Enhancement. The quality of vulnerability knowledge in vulnerability databases often lacks reliability, raising significant concerns [14, 39, 68]. Chen et al. [6], Haryono et al. [26] and Lyu et al. [33] utilized extreme multi-label learning to identify affected libraries; Chen et al. [4, 5] adopted large language model to identify affected libraries; and Wu et al. [68] used learning-to-rank to identify affected libraries and their ecosystems. Besides, several approaches [17, 40, 47, 56, 60, 71] are focused on identifying the patches for a vulnerability, and several techniques [3, 18, 34, 57] are focused on developing predictive models for vulnerability exploitability. Moreover, Anwar et al. [1], Croft et al. [9] and Wunder et al. [69] empirically investigated the inconsistency of CVSS score. Several approaches [24, 25, 46, 53, 54, 62, 72] predicted the key aspects (e.g., vulnerability type, root cause, attack vector, and attacker type) of a vulnerability.

6 CONCLUSIONS

We have introduced Vision, a novel approach for identifying affected library versions for OSS vulnerabilities. Vision selects critical methods and statements, encoding their criticality into signatures represented by weighted IPDGs. It identifies affected library versions through comparing library version signatures with patch signatures. Extensive experiments have demonstrated Vision's effectiveness, efficiency, and practical usefulness. The source code of Vision as well as all our experimental data are available at our replication website [58]. In the future, we plan to extend Vision to support other programming languages to improve its generality.

ACKNOWLEDGMENT

This work was supported by the National Natural Science Foundation of China (Grant No. 62372114, 62332005 and 62402342).

REFERENCES

[1] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. 2021. Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2021), 4255–4269.

[2] Lingfeng Bao, Xin Xia, Ahmed E Hassan, and Xiaohu Yang. 2022. V-SZZ: automatic identification of version ranges affected by CVE vulnerabilities. In *Proceedings of the 44th International Conference on Software Engineering*. 2352–2364.

[3] Haipeng Chen, Rui Liu, Noseong Park, and VS Subrahmanian. 2019. Using twitter to predict when vulnerabilities will be exploited. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data Mining*. 3143–3152.

[4] Tianyu Chen, Lin Li, Bingjie Shan, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. 2023. Identifying Vulnerable Third-Party Libraries from Textual Descriptions of Vulnerabilities and Libraries. *arXiv preprint arXiv:2307.08206* (2023).

[5] Tianyu Chen, Lin Li, Liuchuan Zhu, Zongyang Li, Guangtai Liang, Ding Li, Qianxiang Wang, and Tao Xie. 2023. VulLibGen: Identifying Vulnerable Third-Party Libraries via Generative Pre-Trained Model. *arXiv preprint arXiv:2308.04662* (2023).

[6] Yang Chen, Andrew E Santosa, Asankhaya Sharma, and David Lo. 2020. Automated identification of libraries from vulnerability data. In *Proceedings of the 42nd International Conference on Software Engineering: Software Engineering in Practice*. 90–99.

[7] James R Cordy and Chanchal K Roy. 2011. The NiCad clone detector. In *Proceedings of the 19th International Conference on Program Comprehension*. 219–220.

[8] Roland Croft, M Ali Babar, and M Mehdi Kholoosi. 2023. Data quality for software vulnerability datasets. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. 121–133.

[9] Roland Croft, M Ali Babar, and Li Li. 2022. An investigation into inconsistency of software vulnerability severity across data sources. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. 338–348.

[10] Lei Cui, Zhiyu Hao, Yang Jiao, Haiqiang Fei, and Xiaochun Yun. 2020. Vuldetector: Detecting vulnerabilities using weighted feature graph comparison. *IEEE Transactions on Information Forensics and Security* 16 (2020), 2004–2017.

[11] CWE. 2024. *CWE VIEW: Research Concepts*. Retrieved May 25, 2024 from https://cwe.mitre.org/data/definitions/1000.html

[12] CWE. 2024. *CWE VIEW: Software Development*. Retrieved May 25, 2024 from https://cwe.mitre.org/data/definitions/699.html

[13] Jiarun Dai, Yuan Zhang, Hailong Xu, Haiming Lyu, Zicheng Wu, Xinyu Xing, and Min Yang. 2021. Facilitating vulnerability assessment through poc migration. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 3300–3317.

[14] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the detection of inconsistencies in public security vulnerability reports. In *Proceedings of the 28th USENIX security symposium*. 869–885.

[15] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the detection of inconsistencies in public security vulnerability reports. In *28th USENIX security symposium (USENIX Security 19)*. 869–885.

[16] Ekwa Duala-Ekoko and Martin P Robillard. 2007. Tracking code clones in evolving software. In *Proceedings of the 29th International Conference on Software Engineering*. 158–167.

[17] Trevor Dunlap, Elizabeth Lin, William Enck, and Bradley Reaves. 2023. VFCFinder: Seamlessly pairing security advisories and patches. *arXiv preprint arXiv:2311.01532* (2023).

[18] Michel Edkrantz and Alan Said. 2015. Predicting Cyber Vulnerability Exploits with Machine Learning. In *Proceedings of the Thirteenth Scandinavian Conference on Artificial Intelligence*. 48–57.

[19] GitHub. 2024. *GitHub Advisory Database*. Retrieved May 25, 2024 from https://github.com/github/advisory-database

[20] GitHub. 2024. *GitHub Repository for lukashinsch/spring-boot-actuator-logview*. Retrieved May 25, 2024 from https://github.com/lukashinsch/spring-boot-actuator-logview/tags

[21] GitHub. 2024. *GitHub Repository for spring-projects/spring-integration-extensions*. Retrieved May 25, 2024 from https://github.com/spring-projects/spring-integration-extensions/tags

[22] GitLab. 2024. *GitLab Advisory Database*. Retrieved May 25, 2024 from https://gitlab.com/gitlab-org/security-products/gemnasium-db

[23] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[24] Hao Guo, Sen Chen, Zhenchang Xing, Xiaohong Li, Yude Bai, and Jiamou Sun. 2022. Detecting and augmenting missing key aspects in vulnerability descriptions. *ACM Transactions on Software Engineering and Methodology* 31, 3 (2022), 1–27.

[25] Hao Guo, Zhenchang Xing, Sen Chen, Xiaohong Li, Yude Bai, and Hu Zhang. 2021. Key aspects augmentation of vulnerability description based on multiple security databases. In *Proceedings of the 2021 IEEE 45th Annual Computers, Software, and Applications Conference*. 1020–1025.

[26] Stefanus A Haryono, Hong Jin Kang, Abhishek Sharma, Asankhaya Sharma, Andrew Santosa, Ang Ming Yi, and David Lo. 2022. Automated identification of libraries from vulnerability data: Can we do better?. In *Proceedings of the 30th International Conference on Program Comprehension*. 178–189.

[27] Yongzhong He, Yiming Wang, Sencun Zhu, Wei Wang, Yunjia Zhang, Qiang Li, and Aimin Yu. 2024. Automatically Identifying CVE Affected Versions With Patches and Developer Logs. *IEEE Transactions on Dependable and Secure Computing* 21, 02 (2024), 905–919.

[28] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 90.

[29] java decompiler. 2024. jd-gui. Retrieved May 20, 2024 from https://github.com/java-decompiler/jd-gui

[30] Hyeonseong Jo, Jinwoo Kim, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. 2020. GapFinder: Finding inconsistency of security information from unstructured text. *IEEE Transactions on Information Forensics and Security* 16 (2020), 86–99.

[31] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. 2017. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the Symposium on Security and Privacy*. 595–614.

[32] Chengwei Liu, Sen Chen, Lingling Fan, Bihuan Chen, Yang Liu, and Xin Peng. 2022. Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem. In *Proceedings of the 44th International Conference on Software Engineering*. 672–684.

[33] Yunbo Lyu, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Zhipeng Zhao, Xuan-Bach D Le, Ming Li, and David Lo. 2023. Chronos: Time-aware zero-shot identification of libraries from vulnerability reports. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*. 1033–1045.

[34] Lucas Miranda, Cainã Figueiredo, Daniel Sadoc Menasché, and Anton Kocheturov. 2023. Patch or Exploit? NVD Assisted Classification of Vulnerability-Related GitHub Pages. In *Proceedings of the International Symposium on Cyber Security, Cryptology, and Machine Learning*. 511–522.

[35] Dongliang Mu, Alejandro Cuevas, Limin Yang, Hang Hu, Xinyu Xing, Bing Mao, and Gang Wang. 2018. Understanding the reproducibility of crowd-reported security vulnerabilities. In *Proceedings of the 27th USENIX Security Symposium*. 919–936.

[36] mvnrepository. 2024. *Maven Artifact for eu.hinsch/spring-boot-actuator-logview*. Retrieved May 25, 2024 from https://mvnrepository.com/artifact/eu.hinsch/spring-boot-actuator-logview

[37] mvnrepository. 2024. *Maven Artifact for org.springframework.integration/spring-integration-zip*. Retrieved May 25, 2024 from https://mvnrepository.com/artifact/org.springframework.integration/spring-integration-zip

[38] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. 2016. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering* 21 (2016), 2268–2297.

[39] Viet Hung Nguyen and Fabio Massacci. 2013. The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. 493–498.

[40] Giang Nguyen-Truong, Hong Jin Kang, David Lo, Abhishek Sharma, Andrew E Santosa, Asankhaya Sharma, and Ming Yi Ang. 2022. Hermes: Using commit-issue linking to detect vulnerability-fixing commits. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. 51–62.

[41] NVD. 2023. *NVD*. Retrieved July 14, 2023 from https://nvd.nist.gov/vuln/data-feeds

[42] NVD. 2024. *CVE-2021-43795*. Retrieved May 25, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2021-43795

[43] NVD. 2024. *CVE-2021-43795*. Retrieved May 25, 2024 from https://github.com/line/armeria/pull/3855/files/a380cf982f665459b79909555b5d4b024d7daf1a

[44] NVD. 2024. *CVE-2021-43795*. Retrieved May 25, 2024 from https://github.com/line/armeria/commit/e2697a575e9df6692b423e02d731f293c1313284

[45] NVD. 2024. *CVE-2022-22976*. Retrieved May 25, 2024 from https://nvd.nist.gov/vuln/detail/CVE-2022-22976

[46] Shengyi Pan, Lingfeng Bao, Xin Xia, David Lo, and Shanping Li. 2023. Fine-grained commit-level vulnerability type prediction by CWE tree structure. In *Proceedings of the 45th International Conference on Software Engineering*. 957–969.

[47] Antonino Sabetta, Serena Elisa Ponta, Rocio Cabrera Lozoya, Michele Bezzi, Tommaso Sacchetti, Matteo Greco, Gergő Balogh, Péter Hegedűs, Rudolf Ferenc, Ranindya Paramitha, et al. 2024. Known Vulnerabilities of Open Source Projects: Where Are the Fixes? *IEEE Security & Privacy* 22, 2 (2024), 49–59.

[48] Youkun Shi, Yuan Zhang, Tianhan Luo, Xiangyu Mao, and Min Yang. 2022. Precise (Un) Affected Version Analysis for Web Vulnerabilities. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[49] ShiftLeftSecurity. 2024. *Joern*. Retrieved April 20, 2024 from https://github.com/ShiftLeftSecurity/joern

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

[51] SNYK. 2023. *SNYK Open Source Vulnerability Database*. Retrieved May 25, 2024 from https://security.snyk.io/

[52] sonatype. 2023. *9th Annual State of the Software Supply Chain*. Retrieved May 25, 2024 from https://www.sonatype.com/state-of-the-software-supply-chain/introduction

[53] Jiamou Sun, Zhenchang Xing, Qinghua Lu, Xiwei Xu, Liming Zhu, Thong Hoang, and Dehai Zhao. 2023. Silent Vulnerable Dependency Alert Prediction with Vulnerability Key Aspect Explanation. In *Proceedings of the 45th International Conference on Software Engineering*. 970–982.

[54] Jiamou Sun, Zhenchang Xing, Xin Xia, Qinghua Lu, Xiwei Xu, and Liming Zhu. 2023. Aspect-level information discrepancies across heterogeneous vulnerability reports: Severity, types and detection methods. *ACM Transactions on Software Engineering and Methodology* 33, 2 (2023), 1–38.

[55] Qing Sun, Lili Xu, Yang Xiao, Feng Li, He Su, Yiming Liu, Hongyun Huang, and Wei Huo. 2022. VERJava: Vulnerable Version Identification for Java OSS with a Two-Stage Analysis. In *Proceedings of the International Conference on Software Maintenance and Evolution*. 329–339.

[56] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. 2021. 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*. 3282–3299.

[57] Nazgol Tavabi, Palash Goyal, Mohammed Almukaynizi, Paulo Shakarian, and Kristina Lerman. 2018. Darkembed: Exploit prediction with neural language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*. 7849–7854.

[58] Vision. 2024. *Vision*. Retrieved May 25, 2024 from https://vision-version.github.io

[59] Veracode. 2024. *Veracode Vulnerability Database*. Retrieved May 25, 2024 from https://sca.analysiscenter.veracode.com/vulnerability-database/search

[60] Shichao Wang, Yun Zhang, Liagfeng Bao, Xin Xia, and Minghui Wu. 2022. Vcmatch: a ranking-based approach for automatic security patches localization for OSS vulnerabilities. In *Proceedings of the International Conference on Software Analysis, Evolution and Reengineering*. 589–600.

[61] Ying Wang, Bihuan Chen, Kaifeng Huang, Bowen Shi, Congying Xu, Xin Peng, Yijian Wu, and Yang Liu. 2020. An empirical study of usages, updates and risks of third-party libraries in java projects. In *Proceedings of the 2020 IEEE International Conference on Software Maintenance and Evolution*. 35–45.

[62] Xin-Cheng Wen, Cuiyun Gao, Feng Luo, Haoyu Wang, Ge Li, and Qing Liao. 2024. LIVABLE: Exploring Long-Tailed Classification of Software Vulnerability Types. *IEEE Transactions on Software Engineering* (2024).

[63] Wikepedia. 2024. *HITS algorithm*. Retrieved May 25, 2024 from https://en.wikipedia.org/wiki/HITS_algorithm

[64] wiki. 2024. *Levenshtein Distance*. Retrieved May 25, 2024 from https://en.wikipedia.org/wiki/Levenshtein_distance

[65] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. 2023. V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques. In *Proceedings of the 32nd USENIX Security Symposium*. 6541–6556.

[66] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. 2022. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In *Proceedings of the 31st USENIX Security Symposium*. 3037–3053.

[67] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. 2021. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In *Proceedings of the 30th USENIX Security Symposium*. 3041–3058.

[68] Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. 2024. Identifying Affected Libraries and Their Ecosystems for Open Source Software Vulnerabilities. In *Proceedings of the 46th International Conference on Software Engineering*. 1–12.

[69] Julia Wunder, Andreas Kurtz, Christian Eichenmüller, Freya Gassmann, and Zinaida Benenson. 2023. Shedding Light on CVSS Scoring Inconsistencies: A User-Centric Study on Evaluating Widespread Security Vulnerabilities. *arXiv preprint arXiv:2308.15259* (2023).

[70] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. 2020. MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In *Proceedings of the 29th USENIX Security Symposium*. 1165–1182.

[71] Congying Xu, Bihuan Chen, Chenhao Lu, Kaifeng Huang, Xin Peng, and Yang Liu. 2022. Tracking patches for open source software vulnerabilities. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 860–871.

[72] Sofonias Yitagesu, Zhenchang Xing, Xiaowang Zhang, Zhiyong Feng, Xiaohong Li, and Linyi Han. 2023. Extraction of phrase-based concepts in vulnerability descriptions through unsupervised labeling. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–45.

[73] Zhuotong Zhou, Yongzhuo Yang, Susheng Wu, Yiheng Huang, Bihuan Chen, and Xin Peng. 2024. Magneto: A Step-Wise Approach to Exploit Vulnerabilities in Dependent Libraries via LLM-Empowered Directed Fuzzing. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*.