Identifying Affected Libraries and Their Ecosystems for Open Source Software Vulnerabilities

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

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

Kaifeng 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

Software composition analysis (SCA) tools have been widely adopted to identify vulnerable libraries used in software applications. Such SCA tools depend on a vulnerability database to know affected libraries of each vulnerability. However, it is labor-intensive and error prone for a security team to manually maintain the vulnerability database. While several approaches adopt extreme multi-label learning to predict affected libraries for vulnerabilities, they are practically ineffective due to the limited library labels and the unawareness of ecosystems.

To address these problems, we first conduct an empirical study to assess the quality of two fields, i.e., affected libraries and their ecosystems, for four vulnerability databases. Our study reveals notable inconsistency and inaccuracy in these two fields. Then, we propose Holmes to identify affected libraries and their ecosystems for vulnerabilities via a learning-to-rank technique. The key idea of Holmes is to gather various evidences about affected libraries and their ecosystems from multiple sources, and learn to rank a pool of libraries based on their relevance to evidences. Our extensive experiments have shown the effectiveness, efficiency and usefulness of Holmes.

CCS CONCEPTS

• **Information systems → Open source software**; • **Security and privacy → Vulnerability management**.

KEYWORDS

open source software, vulnerability quality, affected libraries

ACM Reference Format:
Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. 2024. Identifying Affected Libraries and Their Ecosystems for Open Source Software Vulnerabilities. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639582

1 INTRODUCTION

Open source libraries introduce security risks as they often contain vulnerabilities [26, 31, 35, 41, 42, 54, 58, 60]. To mitigate such security risks, software composition analysis (SCA) tools [27] have been increasingly adopted to notify developers about vulnerable libraries used in their software applications. Gartner [16] recently evaluate competitive vendors offering SCA tools, including Synopsys [48], Veracode [51], Snyk [46], Mend.io (formally known as WhiteSource) [34], GitHub [17] and GitLab [19].

All these vendors maintain their own vulnerability database that establishes a mapping between each vulnerability and its affected libraries, affected versions, and patches. To enable vendors provide effective SCA services, the vulnerability database is required to be up-to-date and accurate. Therefore, a security team with strong domain knowledge is needed to devote massive manual effort to curate vulnerability data from various sources. One common source is NIST's National Vulnerability Database (NVD) [39]. Each vulnerability in NVD has a CVE (Common Vulnerability Enumeration) identifier, a description, references (i.e., links to web resources related to the vulnerability), and CPE (Common Platform Enumeration) configurations (i.e., affected software configurations).

Problem. As affected libraries may not be explicitly mentioned in vulnerability description and CPE configurations [24] but hidden in direct or indirect references, it is labor-intensive and error-prone for the security team to manually identify the libraries affected by each vulnerability. To assist the security team in this vulnerability curation process, several affected library identification approaches [8, 12, 24, 33] have been recently proposed.

Dong et al. [12] leverage named entity recognition (NER) to recognize affected libraries from vulnerability description. However, as vulnerability description may not mention affected libraries [24], this NER-based approach cannot recognize affected libraries due to limited knowledge sources. Moreover, as vulnerability description may not mention the complete library coordinates, this NER-based approach may recognize affected library names that are not in line with the library coordinates declared by developers.

To mitigate these issues, Chen et al. [8], Haryono et al. [24] and Lyu et al. [33] formulate this affected library identification problem as an extreme multi-label learning (XML) problem. Such XML-based

[*] S. Wu, W. Song, K. Huang, B. Chen, and X. Peng are with the School of Computer Science and Shanghai Key Laboratory of Data Science, Fudan University, China.
[†] Kaifeng Huang and Bihuan Chen are the corresponding authors.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639582

approaches use library coordinates as labels, and predict labels for each vulnerability based on features extracted from vulnerability description, CPE configurations and direct references. However, they still suffer limitations in practice. One is the limited size of label set, which makes them incapable of handling vulnerabilities whose affected libraries are not in the label set. The other is the unawareness of ecosystems, which causes them fail to distinguish libraries with the same name but from different language ecosystems.

Empirical Study. To assess the quality of vulnerability databases curated by security teams, we conduct an empirical study to analyze the quality of two fields, i.e., affected libraries and their ecosystems, for four vulnerability databases, i.e., GitHub [17], GitLab [19], Veracode [51] and Snyk [46]. On the one hand, we select 18,633 vulnerabilities, and conduct a pairwise comparison of the four vulnerability databases to quantify the inconsistency in these two fields. Notably, affected libraries and their ecosystems are inconsistent between each pair of vulnerability databases for an average of 26.9% and 39.7% of the vulnerabilities, respectively. Specifically, affected libraries and their ecosystems are disjoint for an average of 15.5% and 29.4% of the vulnerabilities, respectively. On the other hand, we manually identify affected libraries and their ecosystems for a sample of 696 vulnerabilities, and quantify the accuracy in these two fields for the four vulnerability databases. The precision in affected libraries and their ecosystems ranges from 0.591 to 0.875 and 0.654 to 0.997, respectively, while the recall in affected libraries and their ecosystems ranges from 0.865 to 0.939 and 0.960 to 0.998, respectively.

Our Approach. To overcome the limitations of XML-based approaches and enhance the quality of vulnerability databases, we propose an automated approach, named Holmes, to identify affected libraries and their ecosystems for each vulnerability via a learning-to-rank technique. Specifically, Holmes takes a CVE identifier as an input, and works in three steps. The first step of Holmes is to gather various evidences about affected libraries and their ecosystems from multiple sources. The second step is to calculate relevance to the gathered evidences for each library in our library pool. Our pool currently contains all libraries from Maven, NPM, PyPI and Go, and is designed to be extensible to include other ecosystems. The third step is to learn to rank the pool of libraries according to their relevance to evidences. The output of Holmes is the highly ranked libraries and their corresponding ecosystems.

Evaluation. To show the effectiveness of Holmes, we compare it with three state-of-the-art approaches on the 696 vulnerabilities used in our empirical study. Our evaluation indicates that i) Holmes can identify affected libraries and their ecosystems with a mean average precision at 1 (i.e., mAP@1) of at least 0.944 and 0.855, respectively; and ii) Holmes can outperform the best of the state-of-the-art in identifying affected libraries by at least 30.1% in mAP@1. In addition, we also evaluate the efficiency of Holmes. Holmes takes 3.6 hours to train, and 18.5 seconds to predict.

To demonstrate the practical usefulness of Holmes, we first conduct a user study with 20 participants. Our evaluation indicates that with the assistance of Holmes, participants can identify affected libraries and ecosystems more accurately. Then, we use Holmes to detect vulnerabilities that are labeled with incorrect affected libraries or ecosystems in the four vulnerability databases, and report them to the four vendors. Two vendors have replied. They respectively fixed the fields for 48 of the 62 reported vulnerabilities and 62 of the 78 reported vulnerabilities.

Contribution. This work makes the following contributions.

- We conducted an empirical study to understand the quality of affected libraries and ecosystems in four vulnerability databases.
- We proposed Holmes to automatically identify affected libraries and their ecosystems for open source software vulnerabilities.
- We conducted extensive experiments to demonstrate the effectiveness, efficiency and practical usefulness of Holmes.

2 DEFINITION AND MOTIVATION

We first introduce the definition of affected library and ecosystem, and then we present two motivating examples.

2.1 Definition

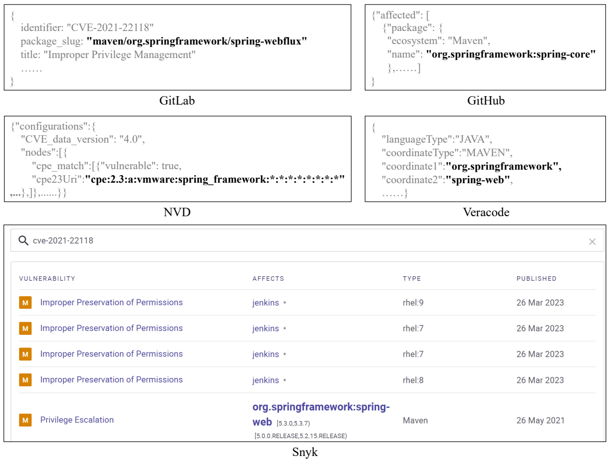
The *affected library* is referred to as the library that *directly* contains the vulnerable code. It is the root of the vulnerability where attackers can exploited the weakness and maintainers would apply fixing patches. The affected library is available on open-source software package registries with an identical name. We do not include downstream libraries transitively/directly depending on the affected library. The *ecosystem* is referred to as the open-source software package registry name hosting the affected library.

2.2 Motivating Examples

We use two vulnerabilities CVE-2021-22118 and CVE-2017-1000163. CVE-2021-22118 is detected in the Spring framework in the Maven ecosystem. Its affected library is uniquely identified by its group ID *org.springframework* and artifact ID *spring-web*. CVE-2017-1000163 locates in the Phoenix framework in both NPM and Hex ecosystem. Its affected library is named *phoenix* in both ecosystems.

Discrepancy in Vulnerability Schema. Fig. 1 shows part of the vulnerability information of CVE-2021-22118 in various vulnerability databases. Overall, different databases employ different schema to represent the affected libraries and their ecosystems. NVD uses the "cpe23uri" field to denote affected software, which includes the vendor name *vmware* and the product name *spring-_framework*, but does not report the library name *spring-web*. The other four databases denote the affected libraries in a format close to the library coordinate in the Maven ecosystem. For example, GitLab uses the "package_slug" field, which uses "/" as the separator to join ecosystem, group ID and artifact ID of the affected library; and GitHub uses the "name" field, which uses ":" as the separator to join group ID and artifact ID of the affected library. Besides, NVD does not report ecosystems, while the other four databases do report ecosystems, e.g., the "ecosystem" field in GitHub, the "coordinateType" field in Veracode, and the "TYPE" field in Snyk.

Quality Issues about Affected Libraries. As shown in Fig. 1, GitLab, GitHub and Snyk have different affected libraries from the ground truth for CVE-2021-22118. On the one hand, vulnerability databases may mark libraries that depend on vulnerable libraries as the affected libraries. For example, GitLab reports *spring-webflux* as an affected library for CVE-2021-22118 because *spring-webflux* depends on the vulnerable version of *spring-web* where the vulnerability truly locates. Similarly, Snyk marks *jenkins* as an affected library for CVE-2021-22118 because *jenkins* includes the vulnerable version



```
{
  identifier: "CVE-2021-22118"
  package_slug: "maven/org.springframework/spring-webflux"
  title: "Improper Privilege Management"
  ……
}
```

GitLab

```
{"affected": [
  {"package": {
   "ecosystem": "Maven",
   "name": "org.springframework:spring-core"
   },……]
}
```

GitHub

```
{"configurations":{
  "CVE_data_version": "4.0",
  "nodes":[{
    "cpe_match":[{"vulnerable": true,
    "cpe23Uri":"cpe:2.3:a:vmware:spring_framework:*:*:*:*:*:*:*:*"
,...},]},......}}
```

NVD

```
{
  "languageType":"JAVA",
  "coordinateType":"MAVEN",
  "coordinate1":"org.springframework",
  "coordinate2":"spring-web",
  ……}
```

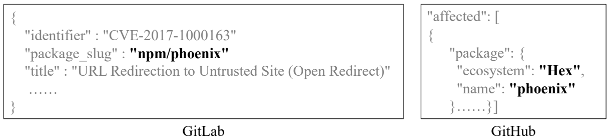
Veracode

cve-2021-22118

VULNERABILITY	AFFECTS	TYPE	PUBLISHED
M Improper Preservation of Permissions	jenkins *	rhel:9	26 Mar 2023
M Improper Preservation of Permissions	jenkins *	rhel:7	26 Mar 2023
M Improper Preservation of Permissions	jenkins *	rhel:7	26 Mar 2023
M Improper Preservation of Permissions	jenkins *	rhel:8	26 Mar 2023
M Privilege Escalation	org.springframework:spring-web [5.3.0,5.3.7) [5.0.0.RELEASE,5.2.15.RELEASE)	Maven	26 May 2021

Snyk

Figure 1: Affected Libraries and Their Ecosystems of CVE-2021-22118 in Different Vulnerability Databases



```
{
  "identifier" : "CVE-2017-1000163"
  "package_slug" : "npm/phoenix"
  "title" : "URL Redirection to Untrusted Site (Open Redirect)"
   ……
}
```

GitLab

```
"affected": [
{
  "package": {
   "ecosystem": "Hex",
   "name": "phoenix"
  }……}]
```

GitHub

Figure 2: Affected Libraries and Their Ecosystems of CVE-2017-1000163 in GitLab and GitHub Advisory

of *spring-web* in the OpenShift Container Platform of Red Hat [52]. On the other hand, vulnerability databases can report incorrect affected libraries. For example, GitHub reports *spring-core* as an affected library. However, after investigation, we confirm that *spring-core* does not transitively depend on *spring-web*, and also does not share similar vulnerable code fragment [14, 15]. Therefore, we reported this issue to GitHub, and GitHub fixed it [18].

Quality Issues about Ecosystems. Fig. 2 shows part of the vulnerability information of CVE-2017-1000163 in GitLab and GitHub advisory. GitLab identifies *phoenix* in the NPM ecosystem as the affected library, while GitHub identifies *phoenix* in the Hex ecosystem as the affected library. Actually, CVE-2017-1000163 affects the *phoenix* library in both ecosystems. Unfortunately, both GitLab and GitHub fail to provide the complete ecosystem information.

These examples raise concerns about the quality of vulnerability databases, and motivate us to systematically assess it (see Sec. 3).

3 EMPIRICAL STUDY

We conduct an empirical study to assess the quality of vulnerability databases by answering the following two research questions.

- **RQ1 Consistency Assessment:** How is the consistency in affected libraries and their ecosystems of the same vulnerabilities across different vulnerability databases?
- **RQ2 Accuracy Assessment:** How is the accuracy in affected libraries and their ecosystems in different vulnerability databases?

3.1 Study Design

Vulnerability Database Selection. We selected vendors that provided SCA tools and publicly available vulnerability databases from the list of vendors reviewed by Gartner [16]. These vendors included Veracode [51], Snyk [46], Mend.io [34], GitHub [17] and GitLab [19]. As Mend.io did not provide affected libraries as the required field, we selected the vulnerability databases of the other four vendors. Then, we crawled these four vulnerability databases, which took three months. In total, we obtained 184,981, 11,343, 24,038 and 28,034 vulnerabilities from GitHub, GitLab, Veracode and Snyk. As GitHub also contained unreviewed vulnerabilities in closed source software, its size was much larger than GitLab, Veracode and Snyk which only contained vulnerabilities in open source software. For the ease of presentation, we referred to GitHub, GitLab, Veracode and Snyk as VD_A, VD_B, VD_C and VD_D, respectively.

Vulnerability Selection. As vulnerability databases did not provide affected libraries or their ecosystems for some vulnerabilities, we removed such vulnerabilities for the ease of assessment. This resulted in 8,190, 11,343, 24,034 and 28,034 vulnerabilities in VD_A, VD_B, VD_C and VD_D, respectively. To enable a pairwise comparison in **RQ1**, for each of the vulnerabilities from these four vulnerability databases, we only kept it if it was included in at least two of the four vulnerability databases. As a result, we obtained 18,633 vulnerabilities.

Consistency Assessment (RQ1) Setup. We conducted a pairwise comparison of the four vulnerability databases with respect to the two fields, i.e., affected libraries and their ecosystems. As showed in Sec. 2, vulnerability databases had different schema for the two fields. Therefore, to enable a fair pairwise comparison, we first extracted these two fields for each vulnerability from the four vulnerability databases according to their schema, and then aligned them to a unified schema of affected library names and ecosystem names. Using such unified vulnerability data, we first measured the consistency in ecosystems for each pair of vulnerability databases. Then, we measured the consistency in affected libraries using only the vulnerabilities where the pair of vulnerability databases shared the same ecosystem.

As there could be multiple affected libraries and ecosystems for a vulnerability, we adopted topological relations [13] to represent consistency and inconsistency between a pair of vulnerability databases $\langle VD_x, VD_y \rangle$. Given a vulnerability whose ecosystems (resp. affected libraries) were respectively E_x and E_y (resp. L_x and L_y) in VD_x and VD_y, we defined the topological relations as follows.

- *Equal.* The vulnerability has equal ecosystems (resp. affected libraries) in VD_x and VD_y, i.e., $E_x = E_y$ (resp. $L_x = L_y$).
- *Disjoint.* The vulnerability has disjoint ecosystems (resp. affected libraries) in VD_x and VD_y, i.e., $E_x \cap E_y = \emptyset$ (resp. $L_x \cap L_y = \emptyset$).
- *Contain.* The ecosystems (resp. affected libraries) of the vulnerability in VD_x contain the ecosystems (resp. affected libraries) of the vulnerability in VD_y, i.e., $E_y \subseteq E_x$ (resp. $L_y \subseteq L_x$).
- *Contained.* The ecosystems (resp. affected libraries) of the vulnerability in DB_y contain the ecosystems (resp. affected libraries) of the vulnerability in DB_x, i.e., $E_x \subseteq E_y$ (resp. $L_x \subseteq L_y$).
- *Overlap.* The vulnerability has overlap ecosystems (resp. affected libraries) in VD_x and VD_y, i.e., $E_x \cap E_y \neq \emptyset \wedge E_x - E_y \neq \emptyset \wedge E_y - E_x \neq \emptyset$ (resp. $L_x \cap L_y = \emptyset \wedge L_x - L_y \neq \emptyset \wedge L_y - L_x \neq \emptyset$).

Accuracy Assessment (RQ2) Setup. To assess the accuracy in affected libraries and their ecosystems for each vulnerability database, we need to construct a ground truth dataset of vulnerabilities. To this end, we first randomly sampled 15% of the 18,633 vulnerabilities, i.e., 2,795 vulnerabilities. Then, two of the authors manually

Table 1: Consistency Results of Ecosystem

Relation	$\langle VD_A, VD_B\rangle$	$\langle VD_A, VD_C\rangle$	$\langle VD_A, VD_D\rangle$	$\langle VD_B, VD_C\rangle$	$\langle VD_B, VD_D\rangle$	$\langle VD_C, VD_D\rangle$	Total
Equal	7,135 (98.2%)	3,713 (70.9%)	1,405 (63.2%)	4,622 (70.3%)	1,370 (50.3%)	972 (8.7%)	8,577
Disjoint	25 (0.3%)	263 (5.0%)	658 (29.6%)	579 (8.8%)	1,189 (43.6%)	10,025 (89.3%)	10,751
Contain	0 (0.0%)	7 (0.1%)	2 (0.1%)	42 (0.6%)	9 (0.3%)	141 (1.3%)	188
Contained	107 (1.5%)	1,253 (23.9%)	159 (7.1%)	1,285 (19.6%)	150 (5.5%)	50 (0.4%)	1,572
Overlap	0 (0.0%)	1 (0.0%)	0 (0.0%)	43 (0.7%)	8 (0.3%)	31 (0.3%)	74
Total	7,267	5,237	2,224	6,571	2,726	11,219	18,633

identified the affected libraries and their ecosystems for these vulnerabilities by confirming whether the affected libraries and their ecosystems reported in the four vulnerability databases were correct and whether any affected libraries and their ecosystems were missed. The specific labeling process is illustrated as follows.

- Step 1: Collecting package names from NVD's description, repository or package registry URLs from NVD's reference, and product names from NVD's CPE.
- Step 2: Filtering deleted vulnerabilities marked as "REJECT" on NVD and deleted libraries whose URLs for repositories or packages are unreachable (e.g., unpublished due to maliciousness).
- Step 3: Searching repositories on GitHub and official websites.
- Step 4: Labeling ecosystems via identifying package registries and key elements from repository Readme or product guides, e.g., setup commands (e.g., apt install), example snippet, etc.
- Step 5: Collecting patch commits from NVD's reference, patch commits in issue reports of NVD's reference, and patch commits whose commit message contains the CVE identifier in searched repositories in Step 1&3. Verifying the correctness of patch commits by confirming patched project and patch commits with description and NVD's CWE (e.g., the code is patched on an infinite loop which will lead to a DoS attack).
- Step 6: Labeling affected libraries. If the patch commits are collected, we identify library declaration files (e.g., Maven's POM file) from the paths of patched files. Otherwise, we obtain vulnerable versions and patched versions, and obtain code-level differences in repositories found in Step 1&3. In that way, the affected library is concluded by confirming the repository from existence of vulnerable elements in code differences.

Finally, we built a ground truth dataset of 696 vulnerabilities. It included 183, 250, 184 and 79 vulnerabilities from Maven, NPM, PyPI and Go, respectively. We used Cohen's Kappa coefficient to measure agreement, and it reached 0.952 for inspecting affected libraries and 0.986 for inspecting ecosystems. A third author was involved to solve disagreement. Our labeling process took 400 man-hours.

We used precision and recall as the indicators of accuracy. Specifically, given a vulnerability whose affected libraries are L in a vulnerability database and whose ground truth is L_{GT}, we respectively defined the precision and recall in affected libraries for this vulnerability as $|L \cap L_{GT}|/|L|$ and $|L \cap L_{GT}|/|L_{GT}|$. The precision and recall in ecosystems can be defined in the same way. Then, the precision and recall for the vulnerability database is the average precision and recall over all the vulnerabilities in the vulnerability database.

Table 2: Consistency Results of Affected Library

Relation	$\langle VD_A, VD_B\rangle$	$\langle VD_A, VD_C\rangle$	$\langle VD_A, VD_D\rangle$	$\langle VD_B, VD_C\rangle$	$\langle VD_B, VD_D\rangle$	$\langle VD_C, VD_D\rangle$	Total
Equal	6,275 (87.9%)	2,503 (67.4%)	1,096 (78.0%)	2,988 (64.6%)	981 (71.6%)	671 (69.0%)	7,465
Disjoint	85 (1.2%)	692 (18.6%)	261 (18.6%)	651 (14.1%)	261 (19.1%)	210 (21.6%)	1,104
Contain	16 (0.2%)	89 (2.4%)	17 (1.2%)	391 (8.5%)	96 (7.0%)	27 (2.8%)	482
Contained	755 (10.6%)	287 (7.7%)	28 (2.0%)	412 (8.9%)	19 (1.4%)	26 (2.7%)	1,219
Overlap	4 (0.1%)	142 (3.8%)	3 (0.2%)	180 (3.9%)	13 (0.9%)	38 (3.9%)	207
Total	7,135	3,713	1,405	4,622	1,370	972	8,577

3.2 Consistency Assessment (RQ1)

Table 1 reports the consistency assessment results for each pair of vulnerability databases with respect to ecosystems. The last row lists the total number of vulnerabilities that are both included in the two compared vulnerability databases. On the one hand, 39.7% of the vulnerabilities have inconsistent ecosystems of affected libraries in each vulnerability database pair, which is moderately large. The largest inconsistency resides in $\langle VD_C, VD_D\rangle$, where 10,247 (91.3%) of the 11,219 vulnerabilities have inconsistent ecosystems. On the other hand, *disjoint* surprisingly accounts for the most common inconsistency relation, which involves 10,751 (57.7%) of the 18,633 vulnerabilities across the six vulnerability database pairs, followed by the *contained* relation which involves 1,572 vulnerabilities. On average, 29.4% of the vulnerabilities have *disjoint* (totally different) ecosystems in each vulnerability database pair.

Table 2 shows the consistency assessment results for each pair of vulnerability databases with respect to affected libraries. The last row gives the total number of vulnerabilities whose ecosystems are *equal* in the two compared vulnerability databases. On the one hand, a moderately large proportion (i.e., 26.9%) of the vulnerabilities have inconsistent affected libraries in each vulnerability database pair. The largest inconsistency resides in $\langle VD_B, VD_C\rangle$, where 1,634 (35.4%) of the 4,622 vulnerabilities have inconsistent affected libraries. On the other hand, *contained* and *disjoint* account for the most common inconsistency relations, respectively involving 1,219 (14.2%) and 1,104 (12.9%) of the 8,577 vulnerabilities across the six vulnerability database pairs. On average, 15.5% of the vulnerabilities have *disjoint* (totally different) affected libraries in each vulnerability database pair.

Summary: A moderate proportion of vulnerabilities (i.e., respectively 39.7% and 26.9%) incur inconsistent ecosystems and affected libraries in each pair of vulnerability databases. *disjoint* accounts for the most common inconsistency relation; i.e., 29.4% and 15.5% of the vulnerabilities have totally different ecosystems and affected libraries in each pair of vulnerability databases. These results indicate the severity of inconsistency issues across vulnerability databases.

3.3 Accuracy Assessment (RQ2)

Table 3 presents the accuracy assessment results for each vulnerability database with respect to ecosystems (*E.*) and affected libraries (*L.*). We also separately report the results for the four covered ecosystems in our ground truth dataset. The number of vulnerabilities (*#V.*)

Table 3: Accuracy Results of Ecosystem and Affected Library (V. = Vulnerability, Met. = Metric, E. = Ecosystem, L. = Library)

Group	VD_A					VD_B					VD_C					VD_D				
	#V.	Met.	E.	L.	E.+L.	#V.	Met.	E.	L.	E.+L.	#V.	Met.	E.	L.	E.+L.	#V.	Met.	E.	L.	E.+L.
Maven	168	Pre.	0.994	0.631	0.631	169	Pre.	0.985	0.636	0.636	144	Pre.	0.758	0.503	0.503	182	Pre.	0.666	0.459	0.449
		Rec.	0.994	0.625	0.625		Rec.	0.994	0.722	0.722		Rec.	0.903	0.766	0.766		Rec.	0.995	0.845	0.845
NPM	240	Pre.	0.996	0.981	0.981	246	Pre.	0.978	0.957	0.954	228	Pre.	0.786	0.791	0.747	247	Pre.	0.756	0.719	0.712
		Rec.	1.000	0.992	0.992		Rec.	0.996	0.988	0.988		Rec.	0.987	0.978	0.974		Rec.	0.996	0.988	0.988
PyPI	157	Pre.	1.000	0.969	0.969	178	Pre.	0.997	0.967	0.967	127	Pre.	0.808	0.583	0.569	184	Pre.	0.511	0.583	0.464
		Rec.	1.000	0.981	0.981		Rec.	1.000	0.994	0.994		Rec.	0.969	0.843	0.835		Rec.	0.978	0.989	0.973
Go	61	Pre.	1.000	0.885	0.885	72	Pre.	1.000	0.866	0.866	70	Pre.	0.875	0.649	0.649	78	Pre.	0.643	0.514	0.505
		Rec.	1.000	0.885	0.885		Rec.	1.000	0.889	0.889		Rec.	0.971	0.743	0.743		Rec.	0.949	0.885	0.885
Total	626	Pre.	0.997	0.875	0.875	665	Pre.	0.987	0.868	0.867	569	Pre.	0.795	0.654	0.634	691	Pre.	0.654	0.591	0.553
		Rec.	0.998	0.880	0.880		Rec.	0.997	0.911	0.911		Rec.	0.960	0.865	0.862		Rec.	0.986	0.939	0.935

because some vulnerabilities are not included in the other vulnerability databases. In terms of accuracy in providing correct ecosystems of affected libraries (*E.*), none of the vulnerability databases achieve a perfect accuracy. The lowest precision is observed in VD_D (i.e., 0.654), while the lowest recall is found in VD_C (i.e., 0.960). Three of the four vulnerability databases achieve the lowest precision and recall for the vulnerabilities in the Maven ecosystem.

In terms of accuracy in providing correct affected libraries (*L.*), the accuracy is relatively lower than that of affected libraries' ecosystems. VD_D shows the lowest precision (i.e, 0.591), followed by VD_C (i.e, 0.654). At the same time, VD_C exhibits the lowest recall of 0.865. Besides, all the four vulnerability databases achieve the lowest precision and recall for the vulnerabilities in the Maven ecosystem, i.e., 0.631 and 0.625 in VD_A, 0.636 and 0.722 in VD_B, 0.503 and 0.766 in VD_C, and 0.459 and 0.845 in VD_D.

Moreover, the accuracy regarding both affected libraries and their ecosystems (*E.+L.*) remains relatively consistent to the accuracy concerning only the affected libraries (*L.*). The lowest precision is found in VD_D (i.e., 0.553), caused by inaccuracies in both ecosystems (i.e., 0.654) and affected libraries (i.e., 0.591). The lowest recall occurs in VD_C (i.e., 0.862), caused by inaccuracies in both ecosystems (i.e., 0.960) and affected libraries (i.e., 0.865).

Summary: The precision in affected libraries and their ecosystems ranges from 0.591 to 0.875 and 0.654 to 0.997, respectively, and the recall ranges from 0.865 to 0.939 and 0.960 to 0.998, respectively.The four vulnerability databases achieve the lowest accuracy for vulnerabilities in Maven. These results indicate the severity of accuracy issues across vulnerability databases.

4 APPROACH

To improve the quality of vulnerability databases, we propose Holmes. It takes a CVE identifier as an input, and returns its affected libraries and their ecosystems. Fig. 3 shows an overview of Holmes, including *evidence gathering*, *relevance calculation*, and *library ranking*.

4.1 Evidence Gathering

Vulnerability disclosure process results scattered evidences about vulnerable libraries across multiple sources. Here, we utilize two sources, i.e., NVD, where vulnerability description, CPE configurations, and references of a vulnerability can be obtained, and GitHub, where code repository of the affected libraries can be obtained. We summarize six types of evidences that can potentially pinpoint the affected libraries and their ecosystems of a vulnerability.

- **Vendor and Product Name (*vpn*)**: Vendor names and product names from CPE configurations may directly reveal the affected libraries or hint the belonging product of the affected libraries.
- **Product Version (*ver*):** Semantic versions of products from CPE configurations can narrow down the scope of affected libraries as a product and its libraries usually share the same version.
- **Library Coordinate (*lc*)**: Library coordinates from URLs of package registries in references or configuration files of package managers in code repository can pinpoint the affected libraries.
- **File Path (*fp*)**: Paths of source code files mentioned in references or involved in fixing commits in code repository can hint the affected libraries that may contain these paths.
- **Class Name (*cn*)**: Names of classes mentioned in references or involved in fixing commits in code repository can hint the affected libraries that may contain these classes.
- **Language (*lang*)**: Language information from code repository can indicate the potential ecosystems of affected libraries.

We use the following sub-steps to gather potential evidences.

Collecting CVE Raw Data. Given a CVE identifier, we request its advisory in JSON format from NVD. We then parse the advisory to extract the vulnerability description, CPE configurations, and references. Then, we parse CPE configurations based on CPE specification to gather *vpn* and *ver* evidences. Next, we iterate URLs of the references. If a URL points to a package registry (e.g., Maven central repository) of the four ecosystems, we directly parse the URL to gather *lc* evidence.

Otherwise, we crawl the web page of the referenced URL, and process referenced URLs in the crawled web page in the same way except that we do not further process their referenced URLs to strike a balance of enriching evidence source and avoiding evidence noise. We distinguish web pages into two types, i.e., commit pages, denoted as W_c, and other online resource pages (reporting discussions or resolutions about the vulnerability), denoted as W_o, by matching the URL against a regular expression of commits.

Discarding Irrelevant Web Pages from W_o. The crawled web pages, especially those of transitive references, may include some irrelevant web pages, which may introduce noise in gathered evidences. To this end, we use *vpn* to determine the relevance of each page in W_o and discard irrelevant ones from W_o. Specifically, we first apply tokenization to *vpn*. Then, for each page in W_o, we parse it into a DOM (Document Object Model) tree using BeautifulSoup [30], and check whether any of the tokens of *vpn* appear in text nodes of the DOM tree. If not, we consider the page as irrelevant and discard it.

Pruning Distracting CVEs from W_o. The crawled web pages may contain descriptions about other irrelevant CVEs (i.e., referred

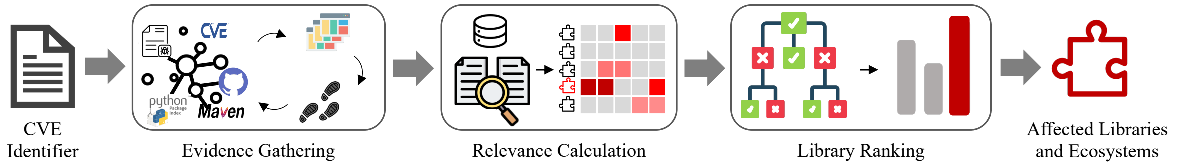
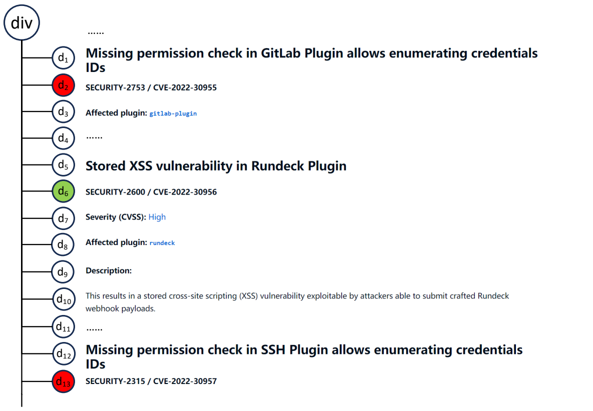


Figure 3: Approach Overview of Holmes



......

Missing permission check in GitLab Plugin allows enumerating credentials IDs

SECURITY-2753 / CVE-2022-30955

Affected plugin: gitlab-plugin

......

Stored XSS vulnerability in Rundeck Plugin

SECURITY-2600 / CVE-2022-30956

Severity (CVSS): High

Affected plugin: rundeck

Description:

This results in a stored cross-site scripting (XSS) vulnerability exploitable by attackers able to submit crafted Rundeck webhook payloads.

......

Missing permission check in SSH Plugin allows enumerating credentials IDs

SECURITY-2315 / CVE-2022-30957

Figure 4: DOM Tree of Jenkins Security Advisory Page [28]

to as distracting CVEs). For example, the reference of CVE-2022-30956 links to Jenkins Security Advisory page [28] which describes all vulnerabilities in Jenkins. To avoid such noise, we need to prune descriptions about distracting CVEs from the pages in W_o.

To tackle this, we prune nodes related to distracting CVEs from the DOM tree of each page in W_o. Specifically, we first iterate each node in the DOM tree to mark the node. If a node contains the given CVE identifier, but does not contain more than two other CVE identifiers, we mark it as *relevant*. This means there is a high chance that the description in this node may mainly talk about the given CVE identifier. If a node contains the given CVE identifier, but also contains more than two other CVE identifiers, we mark it as *distracting*. This means there is a high chance that the description in this node may not mainly talk about the given CVE identifier. If a node only contains other CVE identifier, we also mark it as *distracting*. If a node does not contain any CVE identifier, we mark it as *unknown*.

After node marking, we start to prune distracting nodes. Specifically, for each distracting node, we prune it from its parent node in the DOM tree, and continue to prune its sibling nodes in the right position until we encounter a relevant node. For example, Fig. 4 shows part of the DOM tree of Jenkins Security Advisory page [28], where the relevant, distracting and unknown nodes are highlighted in green, red and white. Nodes d_2 to d_5 are pruned, nodes from d_{13} are pruned, and nodes d_6 to d_{12} are kept. d_5 is actually relevant but is pruned, while d_{12} is actually distracting but is kept. This is the price to ensure the generality of our pruning strategy.

Gathering Evidences from W_o. For each page in W_o, we leverage regular expression matching to gather evidences of file paths *fp* and class names *cn*. For example, the string of file paths contains the path separator "/", and the string of class names follows camel case naming convention. Besides, bug trackers or vendor advisories may assign a bug identifier to a vulnerability. For example, as shown in node d_6 in Fig. 4, SECURITY-2600 is a bug identifier that is an alias to CVE-2022-30956. Hence, for each page in W_o, we use a regular expression to extract bug identifiers, which are used in next step.

Locating Code Repository and Fixing Commits. Code repository can provide valuable evidences about affected libraries. Therefore, we first attempt to locate code repository related to the vulnerability. Specifically, we parse URLs of the commit pages in W_c to extract repository names. To further validate them, we compare each repository name r with each vendor and product name vp in vpn using word-level Levenshtein distance. For the repository with the least Levenshtein distance ed, only if ed is less than 70% of the maximum word length of r and vp, we regard it as a potential repository. In case there is no such a match or W_c is empty, we query GitHub using each vendor and product name in vpn to obtain a list of returned repositories, and choose one returned repository with the least Levenshtein distance in the same way above.

If a code repository can be located, for each commit page in W_c, if it does not belong to the located code repository, we discard it from W_c. Further, we iterate all the commits in the located code repository, and identify commits whose commit message contains the given CVE identifier or its corresponding bug identifiers. We regard these as fixing commits that potentially fix the vulnerability, and we crawl their web pages and add them into W_c. Moreover, we gather language evidence *lang* from the located code repository by extracting composing languages and their percentages.

Gathering Evidences from W_c. For each commit page in W_c, we use the same regular expression matching to gather evidences of file paths *fp* and class names *cn* as in gathering evidences from W_o. In addition, for each changed file in each commit page, we locate the package manager's configuration file that is closest to the changed file in the file directory. Here, we target configuration files of Maven, NPM, PyPI and Go via matching file names (i.e., *pom.xml*, *package.json*, *setup.py* and *go.mod*). Then, we parse configuration files to gather library coordinate evidence *lc*.

4.2 Relevance Calculation

We generate indexes for our library pool to facilitate searching, and calculate the relevance of each library to our gathered evidences.

Indexing Library Pool. We prepare our library pool to facilitate relevance calculation and library ranking, which is a one-time job. Specifically, we first crawl library names and artifacts from Maven central repository, NPM, PyPI and Go packages. In total, our pool contains 4.0 million libraries with 54.7 million library versions.

Then, for each library, we prepare four documents that respectively store library names, all library version numbers, all file paths, and all class names. In detail, to prepare the library name document, we first split the library name into tokens based on punctuation such as period ("."), hyphen ("-") and underscore ("_") as well as camel case naming convention. Then, we remove duplicated tokens as duplication does not convey the importance of this token in our scenario;

we remove suffix digits that are often irrelevant to the library name but may reduce the relevance to our evidences; and we remove commonly used domain names such as "org", "edu" and "com" that may also reduce the relevance to our evidences. Finally, we convert all tokens to lowercase, and store them to the library name document.

To prepare the library version document, we extract the version numbers of all versions of the library, leverage *packaging* [5] to convert these version numbers into the same format, and store them to the library version document. To prepare the file path document, we unzip the artifacts of all library versions, obtain the path of each file, and store non-duplicated file paths to the file path document. To prepare the class name document, we parse the code files of all library versions using *tree-sitter* [50], obtain class names from each code file, and store non-duplicated class names to the class name document.

Finally, after analyzing all libraries, we obtain four collections of the above four kinds of documents, respectively denoted as L_{ln}, L_v, L_{fp} and L_{cn}. We leverage *Lucene* [2] to index our library pool.

Calculating Relevance Score. Using our library pool, we calculate the relevance of each library to our gathered evidences. On the one hand, for evidences *vpn*, *ver*, *fp* and *cn*, we use them to construct queries to our *Lucene* backend. Specifically, for *vpn*, we prepare the query by applying the same processing steps to *vpn* as in preparing the library name document, and perform the query search over L_{ln}. For *ver*, we prepare the query by formatting *ver* by the same format in preparing the library version document, and perform the query search over L_v. For *fp* and *cn*, we directly use them as the query, and perform query search over L_{fp} and L_{cn}, respectively.

For each query search, our *Lucene* backend returns the relevance scores of all libraries in our pool to one of our evidences, denoted as a $N \times 1$ vector, where N is the number of libraries in our pool, and the value of the i_{th} cell corresponds to the relevance score of the i_{th} library computed by BM25 [43]. In particular, the relevance score between a query q and a document l (i.e., $l \in L_{ln}$, L_v, L_{fp} or L_{cn} depending on the query q) is calculated by Eq. 1,

$$score(q,l) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,l) \cdot (k_1+1)}{f(q_i,l) + k_1 \cdot (1 - b + b \cdot \frac{|l|}{avgdl})} \qquad (1)$$

where q_i denotes each individual token in q; $IDF(q_i)$ denotes the inverse document frequency weight of q_i; $|l|$ denotes the number of tokens in l; $avgdl$ denotes the average number of tokens in each document of the document collection; and k_1 and b denotes free parameters, which are set to default value 1.2 and 0.75, respectively.

In the original implementation of BM25, $f(q_i,l)$ denotes the number of times q_i occurs in l, because BM25 was initially designed for searching in long documents, considering token occurrences and document length for relevance computation. However, in our scenario, the frequency of a token plays a less significant role. To address this issue, we modify BM25 by deactivating the importance counting based on token frequency. Instead, we use a gating function in Eq. 2, i.e., setting $f(q_i,l)$ to 1 if q_i occurs in l.

$$f(q_i,l) = \begin{cases} 1, & \text{if } q_i \text{ occurs in } l \\ 0, & \text{otherwise} \end{cases} \qquad (2)$$

On the other hand, for evidence *lc*, we create a $N \times 1$ vector of zeros. For each library name in *lc*, we locate its position i in the vector, and set the value of the i_{th} cell to 1. In other words, due to the high confidence of this evidence, we directly use it to locate the potential affected libraries. For evidence *lang*, we also create a $N \times 1$ vector of zeros. For each composing language in *lang*, we locate the positions of all the libraries within this language ecosystem in the vector, and set their cell value to the percentage of this composing language.

Finally, we obtain a relevance matrix X whose size is $N \times 6$ by merging the six $N \times 1$ vectors generated from each evidence.

4.3 Library Ranking

Based on the relevance matrix, we need to rank our pool of libraries. To this end, we leverage LambdaMART [6], one of the most popular and effective learning-to-rank algorithms. In offline training, we construct a training dataset of vulnerabilities. For each vulnerability, we compute its relevance matrix X, and prepare its $N \times 1$ label vector Y by assigning "1" to the cells that correspond to the ground truth of affected libraries and ecosystems and "0" to the others. Given such $\langle X, Y \rangle$ pairs, we use LambdaMART to train a ranking model. In online prediction, for a vulnerability, we compute its relevance matrix X, and use the ranking model to predict Y. From Y, we use the top k ranked libraries and their corresponding ecosystems as the affected libraries and ecosystems.

5 EVALUATION

We design the following five research questions to evaluate Holmes. We run the experiments on a machine with an Intel Core i7 CPU, a NVIDIA GeForce RTX 2080Ti GPU, 256 GB memory.

- **RQ3 Effectiveness Evaluation:** What is the effectiveness of Holmes in finding affected libraries and ecosystems?
- **RQ4 Ablation Study:** What is the contribution of each type of evidences to the achieved effectiveness of Holmes?
- **RQ5 Efficiency Evaluation:** What is the time overhead of Holmes in finding affected libraries and ecosystems?
- **RQ6 Generality Evaluation:** What is the generality of Holmes to vulnerabilities beyond our ground truth dataset?
- **RQ7 Usefulness Evaluation:** What is the usefulness of Holmes in real-world practice?

5.1 Evaluation Setup

RQ3 Setup. We compare Holmes to three state-of-the-art methods, i.e., Chronos [33], LightXML [24] and FastXML [8]. We consider two usage scenarios. The first is *random-order* (RO) scenario, where training and testing datasets are randomly split. LightXML and FastXML are designed for this scenario. The second is *chronological-order* (CO) scenario, where training and testing datasets are chronologically split. It trains the model on chronologically older vulnerabilities while testing the model on chronologically newer ones. Chronos is designed for this scenario. In both scenarios, we split our ground truth (see Sec. 3) into training and testing datasets by 9:1. For the RO scenario, we employ 10-fold cross-validation.

RQ4, RQ5 and RQ6 Setup. For **RQ4**, we first create six ablated versions of Holmes by removing each of the six types of evidences from Holmes, and then measure the effectiveness changes of Holmes in both RO and CO scenarios with the same setup in **RQ3**. For **RQ5**, we measure the average time taken by Holmes and three state-of-the-art methods for training and predicting. For **RQ6**, we randomly pick another 464 vulnerabilities which at least exist

Table 4: Results of Our Effectiveness Evaluation Compared to the State-of-the-Art

Group	Top k	Random-Order (RO) Scenario						Chronological-Order (CO) Scenario					
		Holmes			Chronos	LightXML	FastXML	Holmes			Chronos	LightXML	FastXML
		mAP$_{E.}$	mAP$_{L.}$	mAP$_{E.+L.}$	mAP$_{L.}$	mAP$_{L.}$	mAP$_{L.}$	mAP$_{E.}$	mAP$_{L.}$	mAP$_{E.+L.}$	mAP$_{L.}$	mAP$_{L.}$	mAP$_{L.}$
Maven	1	0.870	0.677	0.677	0.616	0.175	0.219	1	0.824	0.824	0.615	0.000	0.000
	2	0.935	0.718	0.718	0.678	0.197	0.240	1	0.853	0.853	0.712	0.029	0.000
	3	0.945	0.737	0.737	0.698	0.197	0.242	1	0.873	0.873	0.750	0.049	0.000
NPM	1	0.982	0.920	0.920	0.671	0.044	0.080	1	1	1	0.417	0.133	0.000
	2	0.990	0.943	0.943	0.727	0.052	0.086	1	1	1	0.562	0.133	0.033
	3	0.990	0.943	0.943	0.742	0.056	0.089	1	1	1	0.604	0.133	0.033
PyPI	1	0.989	0.962	0.957	0.686	0.543	0.565	1	1	1	0.500	0.565	0.522
	2	0.992	0.970	0.967	0.750	0.552	0.573	1	1	1	0.583	0.565	0.522
	3	0.992	0.972	0.969	0.755	0.555	0.575	1	1	1	0.583	0.565	0.522
Go	1	0.886	0.810	0.810	0.689	0.177	0.228	0.800	0.733	0.733	0.636	0.000	0.133
	2	0.918	0.816	0.816	0.733	0.203	0.272	0.867	0.733	0.733	0.773	0.033	0.133
	3	0.918	0.816	0.816	0.748	0.211	0.281	0.867	0.733	0.733	0.773	0.033	0.133
Total	1	0.944	0.855	0.853	0.657	0.226	0.261	0.957	0.900	0.900	0.553	0.214	0.200
	2	0.968	0.877	0.876	0.715	0.239	0.277	0.971	0.907	0.907	0.651	0.229	0.207
	3	0.970	0.882	0.881	0.730	0.243	0.279	0.971	0.912	0.912	0.678	0.233	0.207

in one of the four vulnerability databases, establish their ground truth in the same way as in Sec. 3, and feed them to the models trained in **RQ3** for Holmes and the state-of-the-arts to measure their effectiveness.

RQ7 Setup. On the one hand, We design a user study with 20 participants who are required to find affected libraries and their ecosystems for 12 CVEs with the help of Holmes, with the help of Chronos, and without the help of any tools. We recruit 20 participants from security laboratories in multiple universities. They are Postdocs, PhD students and master researchers having at least two years' experience in software security. We randomly select 12 CVEs from our ground truth dataset as tasks. To have a fair comparison, we divide participants into four groups to apply a crossover study [56]. We measure the effectiveness achieved and time taken by participants. On the other hand, we select from our ground truth datasets in **RQ3** and **RQ6** the CVEs whose affected libraries or ecosystems are not correctly labeled in the four vulnerability databases but Holmes correctly identifies them in its top-3 ranking results. We report them to the four vendors via issues or emails to obtain their feedback.

Effectiveness Metric. We adopt mean average precision at k (i.e., mAP@k) [32, 57], a widely used metric for ranking evaluation, to measure the effectiveness of Holmes and state-of-the-arts in **RQ3**, **RQ4** and **RQ6**. Besides, we use precision and recall defined in Sec. 3 to measure the effectiveness achieved by participants in **RQ7**.

5.2 Effectiveness Evaluation (RQ3)

Table 4 reports the results of our effectiveness evaluation by comparing Holmes with the state-of-the-art approaches. In terms of the effectiveness in predicting affected libraries correctly (i.e., mAP$_{L.}$), Holmes achieves a mAP$_{L.}$ of 0.855, 0.877 and 0.882 at the top 1, 2 and 3 results in the RO scenario, and a mAP$_{L.}$ of 0.900, 0.907 and 0.912 at the top 1, 2 and 3 results in the CO scenario. Holmes outperforms all the state-of-the-arts across all the four ecosystems in both RO and CO scenarios. In particular, Chronos is the best of the state-of-the-arts, and Holmes outperforms it by 30.1%, 22.7% and 20.8% in mAP$_{L.}$ at the top 1, 2 and 3 results in the RO scenario, and by 62.7%, 39.3% and 34.5% in mAP$_{L.}$ at the top 1, 2 and 3 results in the CO scenario. Holmes outperforms LightXML and FastXML by at least 216.1% in mAP$_{L.}$ at the top 1, 2 and 3 results in both RO and CO scenarios.

In terms of the effectiveness in predicting ecosystems correctly (i.e., mAP$_{E.}$), Holmes achieves a mAP$_{E.}$ of 0.944, 0.968 and 0.970 at the top 1, 2 and 3 results in the RO scenario, and a mAP$_{E.}$ of 0.957, 0.971 and 0.971 at the top 1, 2 and 3 results in the CO scenario. In terms of the effectiveness in predicting both ecosystems and affected libraries correctly (i.e., mAP$_{E.+L.}$), the mAP$_{E.+L.}$ of Holmes is relatively consistent with its mAP$_{L.}$ because we use the corresponding ecosystems of our predicted affected libraries as the predicted ecosystems. As the state-of-the-art approaches are designed to only identify affected libraries, we are not able to report their mAP$_{E.}$ and mAP$_{E.+L.}$.

We summarize four reasons of inaccuracies in Holmes. First, CPE configurations could be totally different from the affected library names, leading to false positives and false negatives. Second, fixing commits may contain changes to unaffected libraries, leading to false positives. Third, some file paths and class names can be shared by multiple libraries due to code reuse, leading to false positives. Fourth, the language information could be misleading when the vulnerability locates in code written with low-percentage languages as we assign a high relevance to libraries in high-percentage languages, leading to both false positives and false negatives.

Summary: Holmes outperforms all the state-of-the-art approaches across all the four ecosystems in both RO and CO scenarios. Specifically, Holmes outperforms the best of the state-of-the-arts in identifying affected libraries by 30.1% and 62.7% in mAP@1 in the RO and CO scenarios, respectively. Overall, Holmes demonstrates its effectiveness in identifying both affected libraries and their ecosystems for a vulnerability.

5.3 Ablation Study (RQ4)

Table 5 presents the results of our ablation study in the RO scenario. We use mAP@1 with respect to ecosystems (*E.*), affected libraries (*L.*), and both of them (*E.+L.*). Overall, removing any of the six types of evidences from Holmes leads to a drop (↓) in mAP@1. Occasionally, the ablated version of Holmes achieves an increase

Table 5: Results of Our Ablation Study

Removed Evidence	Group	$\text{mAP}_{E.}$	$\text{mAP}_{L.}$	$\text{mAP}_{E.+L.}$
vpn	Maven	**0.922 (↑0.052)**	0.651 (↓0.026)	0.651 (↓0.026)
	NPM	0.958 (↓0.024)	0.744 (↓0.176)	0.740 (↓0.018)
	PyPI	0.951 (↓0.035)	0.837 (↓0.125)	0.826 (↓0.131)
	Go	0.759 (↓0.127)	0.696 (↓0.114)	0.696 (↓0.114)
	Total	**0.924 (↓0.020)**	**0.739 (↓0.116)**	**0.734 (↓0.119)**
ver	Maven	0.792 (↓0.078)	0.631 (↓0.046)	0.631 (↓0.046)
	NPM	0.978 (↓0.004)	0.860 (↓0.060)	0.856 (↓0.064)
	PyPI	0.984 (↓0.005)	0.924 (↓0.038)	0.913 (↓0.044)
	Go	**0.911 (↑0.025)**	0.785 (↓0.025)	0.785 (↓0.025)
	Total	**0.923 (↓0.021)**	**0.808 (↓0.047)**	**0.804 (↓0.049)**
lc	Maven	0.856 (↓0.014)	0.526 (↓0.151)	0.526 (↓0.151)
	NPM	0.982 (–)	0.894 (↓0.026)	0.894 (↓0.026)
	PyPI	**0.995 (↑0.006)**	0.962 (–)	0.957 (–)
	Go	0.873 (↓0.013)	0.342 (↓0.468)	0.342 (↓0.468)
	Total	**0.940 (↓0.004)**	**0.753 (↓0.102)**	**0.751 (↓0.102)**
fp	Maven	0.867 (↓0.003)	0.613 (↓0.064)	0.613 (↓0.064)
	NPM	0.982 (–)	0.912 (↓0.008)	0.912 (↓0.008)
	PyPI	0.989 (–)	**0.967 (↑0.005)**	0.957 (–)
	Go	0.886 (–)	0.785 (↓0.025)	0.785 (↓0.025)
	Total	**0.943 (↓0.001)**	**0.834 (↓0.021)**	**0.831 (↓0.022)**
cn	Maven	0.859 (↓0.011)	0.625 (↓0.052)	0.625 (↓0.052)
	NPM	**0.986 (↑0.004)**	0.916 (↓0.004)	0.916 (↓0.004)
	PyPI	0.984 (↓0.005)	0.957 (↓0.005)	0.946 (↓0.011)
	Go	0.886 (–)	0.797 (↓0.013)	0.797 (↓0.013)
	Total	**0.941 (↓0.003)**	**0.837 (↓0.018)**	**0.834 (↓0.019)**
lang	Maven	0.777 (↓0.093)	0.642 (↓0.035)	0.642 (↓0.035)
	NPM	0.946 (↓0.036)	0.912 (↓0.008)	0.892 (↓0.028)
	PyPI	0.973 (↓0.016)	0.957 (↓0.005)	0.940 (↓0.017)
	Go	0.734 (↓0.152)	0.709 (↓0.101)	0.709 (↓0.101)
	Total	**0.885 (↓0.059)**	**0.830 (↓0.025)**	**0.818 (↓0.035)**

(↑) in mAP@1 only on the vulnerabilities in some ecosystems. The average mAP@1 drop is 0.055 (i.e., 6.4%) when any of the evidences is removed. The removal of *vpn* results in the most significant drop in mAP@1 of 0.116 (i.e., 13.6%). Concerning mAP@1 in ecosystems of the affected libraries, the average mAP@1 drop is 0.018 (i.e., 1.9%) when any of the evidences is removed. The removal of *lang* leads to the largest mAP@1 drop of 0.059 (i.e., 6.3%) among all the evidences.

Summary: Removing any of the six types of evidences from Holmes causes the mAP@1 to drop by 6.4% and 1.9% with respect to affected libraries and ecosystems respectively.

5.4 Efficiency Evaluation (RQ5)

Table 6 presents the results of our efficiency evaluation. Specifically, Holmes takes a total of 3.6 hours to train a ranking model, while Holmes consumes an average of 18.5 seconds to identify affected libraries and their ecosystems for a vulnerability. In terms of the time overhead of each step, in the training phase, Holmes consumes 2.4 hours in the *evidence gathering* step, accounting for 66.7% of the total training time. In the predicting phase, Holmes also spends most of the time in the *evidence gathering* step, taking 11.7 seconds on average, which accounts for 63.2% of the total predicting time. We also show the performance of FastXML, LightXML and Chronos.

Summary: It takes 3.6 hours for Holmes to train a ranking model and 18.5 seconds to predict for a vulnerability.

Table 6: Results of Our Efficiency Evaluation (EG, RC and LR Are the Three Steps of Holmes)

	Holmes				FastXML	LightXML	Chronos
	EG	RC	LR	Total			
Training	2.4h	1.1h	0.1h	3.6h	2.9h	4.4h	4.7h
Predicting	11.7s	6.0s	0.8s	18.5s	7.7s	27.1s	40.6s

Table 7: Results of Our Generality Evaluation

Group	Top	Holmes			Chronos	LightXML	FastXML
	k	$\text{mAP}_{E.}$	$\text{mAP}_{L.}$	$\text{mAP}_{E.+L.}$	$\text{mAP}_{L.}$	$\text{mAP}_{L.}$	$\text{mAP}_{L.}$
Maven	1	0.933	0.658	0.658	0.233	0.199	0.158
	2	0.946	0.675	0.675	0.242	0.284	0.163
	3	0.949	0.678	0.678	0.247	0.295	0.163
NPM	1	0.985	0.904	0.892	0.117	0.029	0.035
	2	0.994	0.909	0.904	0.135	0.029	0.038
	3	0.994	0.913	0.907	0.136	0.031	0.038
PyPI	1	0.918	0.918	0.877	0.369	0.410	0.385
	2	0.943	0.93	0.898	0.410	0.414	0.385
	3	0.951	0.939	0.911	0.421	0.417	0.388
Go	1	0.892	0.716	0.716	0.235	0.265	0.069
	2	0.892	0.716	0.716	0.235	0.265	0.069
	3	0.899	0.729	0.729	0.248	0.278	0.075
Total	1	0.943	0.822	0.807	0.222	0.196	0.162
	2	0.957	0.833	0.822	0.246	0.222	0.165
	3	0.960	0.839	0.829	0.252	0.228	0.166

5.5 Generality Evaluation (RQ6)

Table 7 shows the results of our generality evaluation. The used 464 vulnerabilities include 120, 171, 122 and 51 vulnerabilities in Maven, NPM, PyPI and Go, and have no overlap with the dataset in **RQ3**. As they are randomly selected and thus are not in chronological order with the training data in **RQ3**, we use the trained models in the RO scenario in **RQ3**. In terms of the generality in predicting affected libraries correctly (i.e., $\text{mAP}_{L.}$), Holmes achieves a $\text{mAP}_{L.}$ of 0.822, 0.833 and 0.839 at the top 1, 2 and 3 results, a slight drop of 3.9%, 5.0% and 4.9% compared with the $\text{mAP}_{L.}$ in **RQ3**. However, Chronos, LightXML and FastXML suffer a significant drop, ranging from 6.2% to 66.2%. The reason is that the affected libraries of 288 vulnerabilities in the dataset are not in the scope of the label set of the dataset in **RQ3**. The three state-of-the-arts are constrained by this limited label set, while Holmes does not due to our library pool. Thus, in this practical scenario, Holmes outperforms Chronos, LightXML and FastXML by at least 233.0% in $\text{mAP}_{L.}$ at the top 1, 2 and 3 results. Furthermore, in terms of the generality in predicting ecosystems correctly (i.e., $\text{mAP}_{E.}$) and predicting both ecosystems and affected libraries correctly (i.e., $\text{mAP}_{E.+L.}$), Holmes achieves relatively consistent results with those in **RQ3**.

Summary: When used in practical scenario, Holmes outperforms the best of the state-of-the-arts in identifying affected libraries by at least 233.0% in mAP at the top 1, 2 and 3 results. Overall, Holmes demonstrates its generality in identifying both affected libraries and their ecosystems for a vulnerability.

5.6 Usefulness Evaluation (RQ7)

User Study. Table 8 presents the results of our user study. With the help of Holmes, participants finish the tasks with an average time of 4.36 minutes for each CVE, an increase of 1.11 (34.2%) and 1.23 (39.3%) minutes compared to participants with the help of

Table 8: Results of Our User Study

Group	Time (min)	$\text{Pre}_{E.+L.}$	$\text{Rec}_{E.+L.}$
w/ Holmes	4.36	0.842	0.850
w/Chronos	3.25	0.650	0.683
w/o Tools	0.313	0.675	0.683

Chronos and without the help of any tools. In return, they respectively achieve a 29.5% and 24.5% higher precision (i.e., $\text{Pre}_{E.+L.}$) and recall (i.e., $\text{Rec}_{E.+L.}$) than participants with the help of Chronos and a 24.7% and 24.5% higher precision and recall than participants without the help of any tools. This is reasonable because Holmes predicts more affected libraries that would be missed by participants, and thus participants spend more time in verifying them while enhancing precision and recall. Both the increased time consumption and the improved precision and recall are significant for the 12 tasks.

Vendor Reporting. We report 81 and 40 incorrectly labeled CVEs to Veracode and Snyk by sending emails, and 78 and 62 incorrectly labeled CVEs to GitHub and GitLab by submitting issues. At the time of writing, Veracode and Synk have not replied yet, while GitLab has replied and assigned three maintainers to review these CVEs in their vulnerability database, and GitHub has fixed some of them. Specifically, GitLab has fixed incorrect affected libraries for 48 CVEs based on our suggested ones, and refused to fix for one CVE because the affected library versions have been deleted from Maven central repository. The other 13 CVEs are still waiting for review by GitLab. GitHub has fixed 62 CVEs based on our suggested ones, refused to fix for two CVE, and the other 14 CVEs are still waiting for review by GitLab. It is worth mentioning that GitLab is also interested in the tool that helps find these issues.

Summary: Holmes helps participants to improve the precision and recall in identifying affected libraries and their ecosystems by 24.7% and 24.5% at an additional time price of 1.23 minutes. Holmes helps to find quality issues in 62 and 78 CVEs from GitLab's and GitHub's vulnerability database. 48 and 62 of them have been fixed by GitLab and GitHub. The results indicate the practical usefulness of Holmes.

6 DISCUSSION

Threats. One primary threat to our evaluation is the construction of ground truth. First, the dataset is sampled from the scope of vulnerabilities included in at least two databases rather than from the entire space of vulnerabilities, which can bring the sampling bias to the evaluation. Consequently, the accuracy assessment may represent an estimated upper bound of actual accuracy because there are excluded vulnerabilities in the wild. Second, the dataset is built manually which may be exposed to human error. This threat is mitigated by involving three of the authors who follow an open coding procedure to construct the ground truth. Third, the manual process finds direct vulnerable libraries as the affected libraries to build the ground truth, leading to a lower precision because transitively/directly depending libraries are regarded as incorrect. The precision of four vulnerability databases is measured on the direct vulnerable libraries in the reported affected libraries, which may be lower than the actual value where transitively/directly depending libraries are considered in the ground truth. Fourth, the evaluated results are threatened by the size of the dataset. We manage to build a dataset with 1,160 vulnerabilities, taking 600 man-hours.

Another threat is that Holmes has empirically-set configurable parameters, potentially affecting the generality of Holmes. To address it, we perform a generality evaluation to investigate the robustness of empirically-set configurable parameters on a new dataset, which shows consistent results. Nevertheless, the effectiveness of Holmes might vary for different parameter configurations.

Limitations. First, Holmes relies on a local library pool, which may become outdated due to changes in library repositories. To mitigate this, we implement a pipeline to achieve incremental update regularly to synchronize with the library repositories. Besides, Holmes supports four ecosystems, and we plan to extend to include more ecosystems. Second, Holmes lacks explanation for the predicted results. We plan to identify chain of evidences to further help security teams to quickly confirm the results. Third, Holmes identifies direct vulnerable libraries. We plan to integrate software component analysis to support the detection of transitively/directly depending libraries.

7 RELATED WORK

Vulnerability disclosures contain unstructured, inconsistent and incomplete information, impeding the practical usability for downstream users. In light of these limitations, we undertake a literature review of approaches that address these drawbacks.

Unstructured Information. Some key aspects (e.g., affected library names and versions) of vulnerabilities are provided in unstructured natural language descriptions. While it is easy to manually comprehend them, it is not feasible to directly utilize them in an automated pipeline (e.g., vulnerability propagation analysis). To address this problem, several approaches have been proposed.

With respect to affected library names, Dong et al. [12] leverage named entity recognition and relation extraction to recognize affected library names. However, the recognized library names may not in line with the library coordinates used in package managers. To solve this issue, Chen et al. [8] use library coordinates as labels, and employ extreme multi-label learning (XML) to identify affected library names. As they use the labels of training data as the label set, their approach fails to work on testing data whose label is out of the scope of the label set. To mitigate this issue, Lyu et al. [33] use zero-shot XML to identify previously unseen library names (i.e., they do not appear in training data). They use the labels of training and testing data as the label set. However, their approach does not work in production because the practical label set is too large to be pre-defined. Besides, Haryono et al. [24] evaluate the effectiveness of different XML techniques on this affected library name identification problem. In summary, the first major shortcoming of these XML-based approaches is that they are inevitably limited by the small label set, hindering their practical usability, and they also fail to distinguish libraries with the same name but from different ecosystems. To address this shortcoming, we take a different perspective by feeding the entire label space of library pool to Holmes, making it "know" all libraries in the package registries, and adopting learning-to-rank instead of classification algorithms. The cost is to continuously update the library pool. The second major shortcoming is that they use limited knowledge source to extract limited features. To address this shortcoming, we additionally incorporate the source of knowledge in both direct and indirect webpages from references and source code, apart from CVE descriptions, CPE and text in URLs. We set the depth of crawler to two so as to extract more valuable information that might miss in the directly referenced URLs. We also extract valuable elements (i.e., library names, all library version numbers, all file paths, and all class names) to serve as diverse evidences. This motivates the design of our evidence gathering and relevance calculation.

With respect to affected library versions, Nguyen et al. [38] reveal a significant error rate of affected library versions of Chrome vulnerabilities. Static analysis [3, 11, 25, 37, 45] and dynamic analysis [10] are often employed to precisely determine affected library versions. Besides, several approaches have been proposed to automatically extract vulnerability concepts (i.e., root cause, attack vector, and impact) [61], assign Part-of-Speech (POS) tags [62], and extract vulnerability events (i.e., cause, attacker, consequence, operation, location and version) [55] from vulnerability descriptions. Our work differs from them in identifying affected library names.

Inconsistent Information. Vulnerability disclosures from multiple sources contain inconsistent information which conflicts with each other. Dong et al. [12] quantify the inconsistency in affected library versions between NVD and CVE databases at a massive scale. Jo et al. [29] identify semantic inconsistencies within the cybersecurity domain. Anwar et al. [1] reveal inconsistencies in standardized non-free-form fields, i.e., publication date, CWE class, CVSS rating and affected CPE, in NVD database. Roland et al. [9] report inconsistencies in software vulnerability severity across multiple data sources. To the best of our knowledge, our work is the first to measure the inconsistency in affected library names and ecosystems.

Incomplete Information. Vulnerability disclosures contain incomplete fields. Mu et al.[36] uncover the prevalence of missing reproduction information in vulnerability reports; and Chaparro et al. [7] propose an automated approach to detect the absence of reproduction steps and expected behavior in vulnerability descriptions. Han et al. [23] and Gong et al. [20] adopt machine learning to predict the missed CVSS score. Tan et al. [49], Xu et al. [59], and Wang et al. [53] identify vulnerability patches, which are often missing in most of the vulnerability disclosures. Guo et al. [21, 22] first quantify how many CVEs miss the key aspects of vulnerability type, root cause, attack vector and attacker type in vulnerability descriptions, and propose a neural-network-based approach to augment these missing aspects. Bozorgi et al. [4], Sabottke et al. [44] and Suciu et al. [47] predict the exploitability of vulnerabilities, which is often an optional field in vulnerability databases. Pan et al. [40] predict vulnerability type by hierarchical multi-label classification. They focus on augmenting missing information, but affected library names are not missing but hidden in vulnerability descriptions.

8 CONCLUSIONS

In this paper, we first conduct an empirical study to assess the quality of affected libraries and their ecosystems in four vulnerability databases. Then, we propose Holmes, an automated tool for identifying affected libraries and their ecosystems with a learning-to-rank technique. Finally, our extensive experiments have demonstrated the effectiveness, efficiency and practical usefulness of Holmes. We plan to extend Holmes to support more ecosystems.

ACKNOWLEDGMENT

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

REFERENCES

[1] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. 2022. Cleaning the NVD: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing* 19, 6 (2022), 4255–4269.

[2] Apache. 2023. *Lucene: a Java library providing powerful indexing and search features.* Retrieved July 30, 2023 from https://lucene.apache.org

[3] 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.

[4] Mehran Bozorgi, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. 2010. Beyond Heuristics: Learning to Classify Vulnerabilities and Predict Exploits. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 105–114.

[5] brettcannon, dstufft, pf_moore, and pradyunsg. 2023. *Packaging: Core utilities for Python packages.* Retrieved July 30, 2023 from https://github.com/pypa/packaging

[6] Christopher JC Burges. 2010. From ranknet to lambdarank to lambdamart: An overview. *Learning* 11, 23-581 (2010), 81.

[7] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 396–407.

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

[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] 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.

[11] Stanislav Dashevskyi, Achim D Brucker, and Fabio Massacci. 2019. A screening test for disclosed vulnerabilities in foss components. *IEEE Transactions on Software Engineering* 45, 10 (2019), 945–966.

[12] 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.

[13] Max Egenhofer. 1990. A mathematical framework for the definition of topological relations. In *Proceedings of the fourth international symposium on spatial data handing*. 803–813.

[14] Spring Framework. 2023. *Patch for Spring framework vulnerability.* Retrieved July 14, 2023 from https://github.com/spring-projects/spring-framework/commit/0d0d75e25322d8161002d861fff3ec04ba8be5ac

[15] Spring Framework. 2023. *Patch for Spring framework vulnerability.* Retrieved July 14, 2023 from https://github.com/spring-projects/spring-framework/commit/cce60c479c22101f24b2b4abebb6d79440b120d1

[16] Gartner. 2023. *Gartner Magic Quadrant for Application Security Testing.* Retrieved July 14, 2023 from https://www.microfocus.com/en-us/assets/cyberres/magic-quadrant-for-application-security-testing

[17] GitHub. 2023. *GitHub Advisory Database.* Retrieved July 14, 2023 from https://github.com/github/advisory-database

[18] GitHub. 2023. *Publish GHSA-gfwj-fwqj-fp3v.* Retrieved July 27, 2023 from https://github.com/github/advisory-database/commit/21baa3d

[19] GitLab. 2023. *GitLab Advisory Database.* Retrieved July 14, 2023 from https://gitlab.com/gitlab-org/security-products/gemnasium-db

[20] Xi Gong, Zhenchang Xing, Xiaohong Li, Zhiyong Feng, and Zhuobing Han. 2019. Joint prediction of multiple vulnerability characteristics through multi-task learning. In *Proceedings of the 24th International Conference on Engineering of Complex Computer Systems*. 31–40.

[21] 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.

[22] 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 IEEE 45th Annual Computers, Software, and Applications Conference*. 1020–1025.

[23] Zhuobing Han, Xiaohong Li, Zhenchang Xing, Hongtao Liu, and Zhiyong Feng. 2017. Learning to predict severity of software vulnerability using only vulnerability description. In *Proceedings of the IEEE International conference on software maintenance and evolution*. 125–136.

[24] 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 IEEE/ACM International Conference on Program Comprehension*. 178–189.

[25] Yongzhong He, Yiming Wang, Sencun Zhu, Wei Wang, Yunjia Zhang, Qiang Li, and Aimin Yu. 2023. Automatically Identifying CVE Affected Versions with Patches and Developer Logs. *IEEE Transactions on Dependable and Secure Computing* (2023).

[26] 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).

[27] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. 2021. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 1–11.

[28] Jenkins.io. 2023. *Jenkins Security Advisory*. Retrieved July 14, 2023 from https://www.jenkins.io/security/advisory/2022-05-17/#SECURITY-1969

[29] Hyeonseong Jo, Jinwoo Kim, Phillip Porras, Vinod Yegneswaran, and Seungwon Shin. 2021. GapFinder: Finding inconsistency of security information from unstructured text. *IEEE Transactions on Information Forensics and Security* 16 (2021), 86–99.

[30] leonard. 2023. *Beautiful Soup*. Retrieved July 14, 2023 from https://pypi.org/project/beautifulsoup4/

[31] 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.

[32] Tie-Yan Liu. 2009. Learning to rank for information retrieval. *Foundations and Trends in Information Retrieval* 3, 3 (2009), 225–331.

[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] mend.io. 2023. *Mend.io (formerly known as WhiteSource)*. Retrieved July 14, 2023 from https://www.mend.io/vulnerability-database/

[35] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*. 201–211.

[36] 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.

[37] 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.

[38] 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.

[39] NVD. 2023. *NVD*. Retrieved July 14, 2023 from https://nvd.nist.gov

[40] 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 IEEE/ACM 45th International Conference on Software Engineering*. 957–969.

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

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

[43] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[44] Carl Sabottke, Octavian Suciu, and Tudor Dumitraș. 2015. Vulnerability disclosure in the age of social media: Exploiting twitter for predicting {Real-World} exploits. In *Proceedings of the 24th USENIX Security Symposium*. 1041–1056.

[45] 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.

[46] SNYK. 2023. *SNYK Open Source Vulnerability Database*. Retrieved July 14, 2023 from https://security.snyk.io/

[47] Octavian Suciu, Connor Nelson, Zhuoer Lyu, Tiffany Bao, and Tudor Dumitraș. 2022. Expected exploitability: Predicting the development of functional vulnerability exploits. In *Proceedings of the 31st USENIX Security Symposium*. 377–394.

[48] synopsys. 2023. *Synopsys Software Composition Analysis*. Retrieved July 14, 2023 from https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis/knowledgebase.html

[49] 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.

[50] tree sitter. 2023. *Tree-sitter: An incremental parsing system for programming tools*. Retrieved May 1, 2023 from https://tree-sitter.github.io/tree-sitter/

[51] Veracode. 2023. *Veracode Vulnerability Database*. Retrieved July 14, 2023 from https://sca.analysiscenter.veracode.com/vulnerability-database/search

[52] Vulmon. 2023. *CVE-2021-22118*. Retrieved July 14, 2023 from https://vulmon.com/vendoradvisory?qidtp=red_hat_cve_database&qid=CVE-2021-22118

[53] 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 IEEE International Conference on Software Analysis, Evolution and Reengineering*. 589–600.

[54] 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 IEEE International Conference on Software Maintenance and Evolution*. 35–45.

[55] Ying Wei, Lili Bo, Xiaobing Sun, Bin Li, Tao Zhang, and Chuanqi Tao. 2023. Automated event extraction of CVE descriptions. *Information and Software Technology* 158 (2023), 107178.

[56] wikipedia. 2023. *Crossover Study*. Retrieved July 14, 2023 from https://en.wikipedia.org/wiki/Crossover_study

[57] Wikipedia. 2023. *Evaluation Measures (information retrieval)*. Retrieved July 14, 2023 from https://en.wikipedia.org/wiki/Evaluation_measures_(information_retrieval)#Mean_average_precision

[58] 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 (USENIX Security 20)*. 1165–1182.

[59] 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.

[60] Meiqiu Xu, Ying Wang, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2022. Insight: Exploring Cross-Ecosystem Vulnerability Impacts. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[61] Sofonias Yitagesu, Zhenchang Xing, Xiaowang Zhang, Zhiyong Feng, Xiaohong Li, and Linyi Han. 2021. Unsupervised labeling and extraction of phrase-based concepts in vulnerability descriptions. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*. 943–954.

[62] Sofonias Yitagesu, Xiaowang Zhang, Zhiyong Feng, Xiaohong Li, and Zhenchang Xing. 2021. Automatic part-of-speech tagging for security vulnerability descriptions. In *Proceedings of the IEEE/ACM 18th International Conference on Mining Software Repositories*. 29–40.