Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques

Quang-Cuong Bui, Riccardo Scandariato, Nicolás E. Díaz Ferreyra
{cuong.bui,riccardo.scandariato,nicolas.diaz-ferreyra}@tuhh.de
Hamburg University of Technology
Germany

ABSTRACT

In this work we present Vul4J, a Java vulnerability dataset where each vulnerability is associated to a patch and, most importantly, to a Proof of Vulnerability (PoV) test case. We analyzed 1803 fix commits from 912 real-world vulnerabilities in the *Project KB* knowledge base to extract the *reproducible* vulnerabilities, i.e., vulnerabilities that can be triggered by one or more PoV test cases. To this aim, we ran the test suite of the application in both, the vulnerable and secure versions, to identify the corresponding PoVs. Furthermore, if no PoV test case was spotted, then we wrote it ourselves. As a result, Vul4J includes 79 reproducible vulnerabilities from 51 open-source projects, spanning 25 different Common Weakness Enumeration (CWE) types. To the extent of our knowledge, this is the first dataset of its kind created for Java. Particularly, it targets the study of Automated Program Repair (APR) tools, where PoVs are often necessary in order to identify plausible patches. We made our dataset and related tools publically available on GitHub.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

java, vulnerability, program repair

ACM Reference Format:
Quang-Cuong Bui, Riccardo Scandariato, Nicolás E. Díaz Ferreyra. 2022. Vul4J: A Dataset of Reproducible Java Vulnerabilities Geared Towards the Study of Program Repair Techniques. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3524842.3528482

1 INTRODUCTION

As the threat landscape keeps evolving, detecting and patching software vulnerabilities are crucial activities in the software development industry. However, empirical evidence points to the painful fact that about 50% of developers cannot either identify or fix vulnerable code [10]. In this context, the silver lining is represented by automation, especially by automated tools for the localization of vulnerabilities that, although not perfect, are already available in the market (e.g., Flawfinder [2], FindSecBugs [1], etc.). Nevertheless, the automatic generation of security patches via Automated Program Repair (APR) techniques is still in its early infancy. Only a handful of security-specific approaches have been published so far [9, 12, 13, 15] and they have not been benchmarked at the moment, as there is no benchmark dataset available. Furthermore, initial evidence shows that traditional APR tools do not perform well on security bugs [18]. Therefore, there seems to be a strong call for further research in this area. Still, there is a lack of publicly available data which, in turn, impairs the experimentation and development of novel APR techniques for security bugs.

MSR '22, May 23–24, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528482

Motivation. There exist several vulnerability datasets for many programming languages [4, 7, 16, 19]. However, they do *not* contain bug-witnessing test cases to reproduce their corresponding vulnerabilities, i.e., the so-called Proof of Vulnerability (PoV). Such test cases are often essential for many APR approaches, as they are used as oracles in order to determine whether a generated patch contains an effective repair. In general, it is hard to reproduce exploitation (i.e., PoV) for some kinds of vulnerabilities. It is also desirable to have realistic patches associated to the vulnerabilities, e.g., as defined by the actual developers of the software application. For instance, such patches can be used as ground truth when assessing the generated security patches. In conclusion, curating a dataset of real vulnerabilities, with the corresponding patches and with the PoV test cases is a time-consuming task.

Contributions. The contribution of this paper is two-fold. The first one consists of a dataset with 79 reproducible, real-world Java vulnerabilities corresponding to 51 open-source projects. For this we have i) analyzed the fix commits of 912 Java vulnerabilities from the knowledge base of *Project KB* [3, 19], ii) tried to reproduce them, and iii) retained those ones that met our criteria (see Section 2). Some vulnerabilities did not include any pre-existing PoV test case. Therefore, we created them manually by using the available patches as reference. Finally, the vulnerabilities in our dataset are isolated and contain sufficient testing information to allow the evaluation of the patches generated by APR tools. To the best of our knowledge, this is the first dataset of its kind for the Java programming language. The only other similar dataset has been recently released for the C/C++ programming language [18].

As a second contribution, we have built an execution framework on top of the dataset. The framework is able to automatically execute some common tasks required by APR systems on the projects in our dataset, and validate the reproducibility of the newly added vulnerabilities.

2 DATASET CONSTRUCTION

We have followed the criteria proposed by Just et al. [14] to construct our vulnerability dataset. Particularly, each vulnerability included in VUL4J should meet the following requirements:

C1. Vulnerability is related to Java source code: The vulnerability should have an identifier on the *Common Vulnerabilities and Exposures* (CVE) website or in the project's bug management system, and should be both introduced and patched in Java source files. We exclude the vulnerabilities that require fix actions in Java test files, configuration files, and webpage files (e.g., JSP) since APR tools cannot fix these bugs yet.

C2. Contain an appropriate test suite: Test-based APR tools require the presence of test cases revealing the vulnerability's behavior, as the test cases are used as input to perform fault localization and validate the generated patches. These test cases should fail before, and pass after, the vulnerability's patch is applied. Therefore, we keep only the vulnerabilities which have at least one existing test case that can identified as a PoV (more details in Section 2.2), or those with sufficient information for us to manually write the PoV test cases (more details in Section 2.3).

C3. Reproducible: The project source code should be published publicly, and the revision containing the patch should be accessible. At that revision, the source code of the project should be buildable and the patch should make the status of the PoV test cases change from failing to passing.

C4. Vulnerability is isolated: The patch should only introduce changes that fix the vulnerability. It should not contain unrelated changes, such as developing other features or refactoring.

As mentioned in Section 1, there are several vulnerability datasets available in the related work but do not fulfill the above criteria. Hence, we have created our own starting from the available Java vulnerability datasets. Particularly, we selected the *Project KB* knowledge base maintained by SAP (we refer to it as SAPKB from now on) [3, 19], which is manually curated by the *National Vulnerability Database (NVD)* and project-specific web pages. It contains 912 publicly disclosed vulnerabilities with 1803 corresponding fix commits from 358 open-source Java projects. We choose SAPKB because it contains vulnerabilities from real-world projects rather than synthetic ones. The use of toy-like of synthetic data in security has been deprecated by several authors, including the recent work by Chakraborty et al. [5]. Furthermore, SAPKB contains the vulnerability fix commits from the developers. Such human-created patches are very valuable, as they can be used to validate the machine-generated patches produced by APR tools. In the following subsections, we describe the process we followed to create our vulnerability dataset.

2.1 Patches filtering

We have filtered the vulnerability list as follows:

Step 1: We tried to download the source code from all the repositories included in the dataset. Nine repositories were no longer accessible with the given information in SAPKB. Therefore, we obtained the source code of 349 out of 358 repositories. At this point, there were 899 unique vulnerabilities with 1705 fix commits.

Step 2: To satisfy criteria C4 we sought to retain only the vulnerabilities with *single-fix* commits as *multiple fix* commits may introduce unrelated changes. We found 553 vulnerabilities with a unique ID and a single-fix commit. Other records were reported as duplicated as they correspond to patches performed over different versions of the same project (i.e., located on different branches). We manually examined and validated whether such records were creating the exact same patch by looking at their commit information (e.g., commit message, commit tag, changes in the source code). We then chose one of them and added it as the official patch for that vulnerability. As a result, we discovered 132 additional vulnerabilities that matched the criteria of C4.

Step 3: We then used Git to extract the information of 685 patches retained from last step including the locations (i.e., the file level) where the source code is modified along with the number of changed lines, the revision number before the patch (V_{vul}), and the revision number after the patch (V_{fixed}). Only those patches involving changes in Java source files (i.e., with file name ending in `.java` but not starting with `Test` or ending with `Test.java`) were kept, as they satisfy criteria C1. We further excluded patches whose V_{vul} and V_{fixed} do not support building tools such as Maven, Gradle, and Ant, as these are difficult to build and compile (their building scripts often contain errors). By the end of this step we obtained 417 candidate patches.

2.2 Vulnerabilities reproduction

After filtering and obtaining the list of patches along with their V_{vul} and V_{fixed} revisions, we tried to reproduce the corresponding vulnerabilities in our local machines. Particularly, we checked whether patches indeed fix their targeted vulnerabilities or not. For this, we compared the test-suite execution results of the V_{fixed} and the V_{vul} revisions of each patch. If there existed test cases failing on V_{vul} but passing on V_{fixed} (not vice versa), we concluded that the vulnerability patch was reproducible and thus included it in our dataset.

From a technical point of view, the process outlined above was conducted as follows. First, we tried to build the project by executing the command corresponding to its supported building tool (e.g., `mvn install`, `gradle build`, `ant`). Typically, a building process involves four steps: i) resolving dependencies, ii) compiling the project, iii) running test suites, and iv) packaging the output artifacts. Hence, we first tried to build the V_{fixed} revision and checked whether it terminated after the first two steps. If so, we discarded the vulnerability and marked the patch as *"not reproducible"*. Otherwise, we saved the corresponding *failing-test list*. Only 138 out of 417 patches passed the first two building steps and were thus retained for further analysis. Next, we used the `git revert` command to switch the project's source code back to the V_{vul} revision. We then repeated the building process and collected the failing test list once again.

After collecting the failing test lists of V_{vul} and V_{fixed} we proceeded with the identification of *PoV test cases*. This was done by extracting the test cases failing in the V_{vul} revision but passing the V_{fixed}. Test cases failing in both revisions were considered *irrelevant* and removed when constructing the dataset. As a result, we gathered 69 vulnerability patches that successfully passed the whole reproduction processes. Still, another 70 patches that passed

```
1  // Human patch
2  public void parseCentralDirectoryFormat(final byte[] data, final
        int offset, final int length) {
3  ....
4  - for (int i = 0; i < this.rcount; i++) {
5  + for (long i = 0; i < this.rcount; i++) {
6      for (int j = 0; j < this.hashSize; j++) {
7      }
8    }
9    ....
10 // Manually created PoV test case
11 @Test(timeout = 2000,
12       expected = ArrayIndexOutOfBoundsException.class)
13 public void testCVE_2018_1324() throws IOException {
14   ZipFile zf = new ZipFile(getFile("vulnerable.zip"));
15   zf.close();
16 }
```

Figure 1: Human patch and PoV test case for CVE-2018-1324.

the building process did not have a pre-defined PoV test case. For these ones, we wrote the missing PoV tests manually.

2.3 Missing PoV test cases creation

Inspecting all 70 vulnerability patches manually was not feasible given the high amount of time it would demand. Therefore, we only considered patches containing three modified lines or less. For writing the test cases, we followed the vulnerability exploitation guidelines and resources available either at the NVD website or inside the project's Bug Management System. We carefully examined the patches to understand the fix actions and the context of the vulnerability. Then, we manually wrote test cases that perform the exploit and assert the system's expected behavior. In total, we successfully wrote new test cases for 10 vulnerability patches, increasing our dataset to a total of 79 vulnerabilities.

Example: Figure 1 shows the developer patch and the corresponding manually-written test case for vulnerability CVE-2018-1324[1]. Here, the development team of the Apache Commons Compress library used the wrong variable type in the iteration loop (line 4). Consequently, some extra fields from the input file (a Zip file) are parsed unintentionally. With a specially-crafted Zip file, a hacker could perform a denial of service attack by generating an infinite loop in the system. We checked the corresponding bug report in the project's Bug Management System[2] and located a ZIP file ("vulnerable.zip") that can be used for recreating this attack. Then we wrote a test code that creates a new ZipFile object out of the original one (line 14). As mentioned before, if such a vulnerability is present, then the test code will cause an infinite loop. Hence, we added test assertions to make sure that the unit test will finalize in no more than two seconds (line 11).

3 DATASET DESCRIPTION AND STATISTICS

After completing the curation phase, our dataset contained 79 reproducible vulnerabilities from 51 real-world Java projects. The dataset is publicly available in a GitHub repository[3] where each vulnerability is stored on a separated branch. We included a *csv* file with some summary information for each entry in the dataset[4]

[1]https://nvd.nist.gov/vuln/detail/CVE-2018-1324
[2]https://issues.apache.org/jira/browse/COMPRESS-432
[3]https://github.com/bqcuong/vul4j
[4]https://github.com/bqcuong/vul4j/blob/main/dataset/vul4j_dataset.csv

Table 1: Projects included in the dataset VUL4J.

Project	#Vuls	kLOC	#Tests	#PoVs
apache/struts	10	359	1,697	10
apache/commons-compress	4	48	927	5
jenkinsci/jenkins	3	275	518	3
spring-projects/spring-framework	3	684	2,189	6
spring-projects/spring-security	3	198	513	3
apache/camel	2	925	5,222	4
apache/commons-fileupload	2	6	70	3
apache/commons-imaging	2	42	562	2
apache/cxf	2	737	89	3
apache/pdfbox	2	145	359	2
apache/sling	2	507	25	3
cloudfoundry/uaa	2	182	2,669	2
FasterXML/jackson-dataformat-xml	2	9	140	2
inversoft/prime-jwt	2	2	33	2
OpenRefine/OpenRefine	2	144	516	2
36 other projects (mean)	1	121	567	2.1
all projects (mean)	**1.5**	**169**	**705**	**2.5**

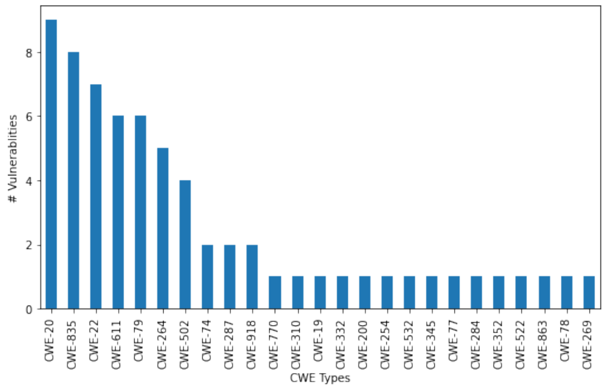

Figure 2: Distribution of vulnerabilities by CWE categories.

including the vulnerability ID, the human patch URL, the build configuration (build system and compliance level), the names of failing tests and failing modules, the build command, and the tests execution commands. Such information is essential for reproducing the vulnerabilities. It should be noted that only the vulnerability ID and the human patch URL are currently available in SAPKB.

Table 1 shows the projects added to our dataset. Due to space limitations, we only report detailed information of the top 15 projects containing most of the vulnerabilities. For the rest, we have computed average values. Apache Camel is the project with the highest number of Lines of Code (LOC) (more than 925k), whereas the one with the smallest amount is Prime JWT (around 2k LOC). The number of test cases per project ranges from 25 to 5,222. The projects in our dataset span over multiple domains including libraries, web frameworks, data-processing desktop apps, and CI/CD servers. This suggests that the included vulnerabilities come from projects of different sizes and application domains.

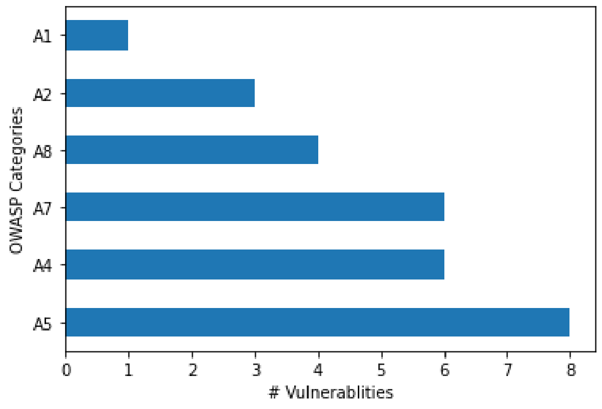

Figure 3: Coverage of OWASP Top 10 Web Application Security Risks (2017)

As illustrated in Figure 2, our dataset covers 25 unique classes of vulnerabilities (or CWEs) being *CWE-20 – Improper Input Validation* the most frequent one (11.4%). Due to the scarcity of information available, 13 vulnerabilities could not be mapped into any of the CWE types included in the NVD website.

Figure 3 shows the vulnerability coverage over the OWASP Top 10 Web Application Security Risks (2017)[5], which is a well-known taxonomy often used to assess the diversity of vulnerability datasets. *OWASP A5 - Broken Access Control* is the most prevalent category, as it was found in eight different projects. Overall, 28 out of 79 vulnerabilities correspond to the OWASP Top 10 (35.4%). We plan to curate more vulnerabilities for covering all the OWASP categories in the future.

4 DATASET USAGE FOR APR STUDIES

Our main goal is to encourage practitioners and researchers to use our dataset as a benchmark for the evaluation of current and forthcoming APR tools. There are only a few repair tools focusing on Java vulnerabilities [6, 15, 20], however, their implementations are not publicly accessible. Therefore, we hope many better APR techniques for fixing security bugs will be proposed in near future with the presence of our dataset.

On top of our dataset, we have built an execution framework that allows running several APR-evaluation tasks, as well as to validate and reproduce vulnerabilities in a convenient way. The following example illustrates the usage of our framework and dataset included in the VUL4J repository.

Assisting APR evaluation: APIs can automatically:

- *checkout* the vulnerabilities from VUL4J repository to a *temp* folder for evaluation purposes.
- *compile & build* the project's source code and test code. Currently, only the projects using Java 7 and Java 8 are supported.
- *run tests* of the project and report their results using JavaScript Object Notation (JSON) format, so it can be easily consumed by other programs.

Validating vulnerability's reproducibility: After adding a new vulnerability to the dataset, this API could be used to automatically validate whether its corresponding project is buildable and if PoV test cases are successfully identified.

5 RELATED WORK

There are several vulnerability datasets created from real-world open-source projects in the literature. Ponta et al. [19] (used in this work) continuously monitored the NVD website and 50 other project-specific web pages, seeking for new Java vulnerability disclosures then. Their approach consists of manually extracting and verifying fix commits, what translates into a high confidence level of the patch quality. Gkortzis et al. [11] collected a multilingual vulnerability dataset of open-source projects by automatically crawling reports from the NVD website and downloading source code of these projects from Version Control Systems. Then they mapped the vulnerable project versions they found in NVD reports to the version references (i.e., commit tags and branches) in the project repositories. Hence, it is possible to know at which revision number a project is vulnerable, but no further information can be obtained. Nikitopoulos et al. [16] also presented a vulnerability dataset covering more than 40 programming languages. Particularly, they only considered and collected vulnerabilities from the NVD website whose fix commits were available. They categorized such vulnerabilities by CWE types and located patches on a file granularity level. Furthermore, for each vulnerability in their dataset, they provided a pair of vulnerable and non-vulnerable source code files.

6 LIMITATIONS

Currently, we have considered only simple vulnerability patches when creating the missing PoV test cases. We wrote PoV test cases for 10/70 vulnerability patches. In the future, we plan to cover more vulnerabilities with the help of test generation tools, such as Randoop [17] and EvoSuite [8].

7 CONCLUSIONS

In this paper, we present for the first time a dataset, namely VUL4J, consisting of 79 real world Java vulnerabilities from 51 open-source projects along with their exploiting test cases. This dataset is designed for the purpose of aiding the evaluation of current and future APR techniques on security bugs. We provide our released dataset as a single Git repository, where each vulnerability lies on a branch, along with a *csv* file that summarizes information about vulnerabilities. There are also useful APIs implemented to automate some common tasks on the dataset. This helps the practitioners to easily set up the vulnerabilities to feed to APR tools, also validate the reproduction progress of newly added vulnerabilities in the dataset. In ongoing work, we are extending the dataset to include more vulnerabilities, in order to cover more vulnerability classes.

ACKNOWLEDGMENTS

This work was partly funded by the European Union's Horizon 2020 programme under grant agreement No. 952647 (AssureMOSS).

[5]https://owasp.org/www-project-top-ten/2017/Top_10

REFERENCES

[1] [n. d.]. Find Security Bugs. https://find-sec-bugs.github.io. Accessed: 2022-01-15.

[2] [n. d.]. Flawfinder Home Page. https://dwheeler.com/flawfinder. Accessed: 2022-01-15.

[3] [n. d.]. Home page of project "KB". https://github.com/sap/project-kb. Accessed: 2022-01-15.

[4] Guru Bhandari, Amara Naseer, and Leon Moonen. 2021. CVEfixes: automated collection of vulnerabilities and their fixes from open-source software. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering*. 30–39.

[5] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. 2021. Deep learning based vulnerability detection: Are we there yet. *IEEE Transactions on Software Engineering* (2021).

[6] Jianlei Chi, Yu Qu, Ting Liu, Qinghua Zheng, and Heng Yin. 2020. SeqTrans: Automatic Vulnerability Fix via Sequence to Sequence Learning. *arXiv preprint arXiv:2010.10805* (2020).

[7] Jiahao Fan, Yi Li, Shaohua Wang, and Tien N Nguyen. 2020. A C/C++ Code Vulnerability Dataset with Code Changes and CVE Summaries. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 508–512.

[8] Gordon Fraser and Andrea Arcuri. 2011. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 416–419.

[9] Xiang Gao, Bo Wang, Gregory J Duck, Ruyi Ji, Yingfei Xiong, and Abhik Roychoudhury. 2021. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–27.

[10] Tiago Espinha Gasiba, Ulrike Lechner, Maria Pinto-Albuquerque, and Daniel Mendez. 2021. Is Secure Coding Education in the Industry Needed? An Investigation Through a Large Scale Survey. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 241–252.

[11] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: A Dataset of Security Vulnerabilities in Open-Source Systems. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 18–21.

[12] Jacob Harer, Onur Ozdemir, Tomo Lazovich, Christopher P Reale, Rebecca L Russell, Louis Y Kim, and Peter Chin. 2018. Learning to repair software vulnerabilities with generative adversarial networks. *arXiv preprint arXiv:1805.07475* (2018).

[13] Zhen Huang, David Lie, Gang Tan, and Trent Jaeger. 2019. Using safety properties to generate vulnerability patches. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 539–554.

[14] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose, CA, USA, 437–440. Tool demo.

[15] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. 2017. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*. Springer, 229–246.

[16] Georgios Nikitopoulos, Konstantina Dritsa, Panos Louridas, and Dimitris Mitropoulos. 2021. CrossVul: A Cross-Language Vulnerability Dataset with Commit Data. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 1565–1569. https://doi.org/10.1145/3468264.3473122

[17] Carlos Pacheco and Michael D Ernst. 2007. Randoop: feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. 815–816.

[18] Eduard Pinconschi, Rui Abreu, and Pedro Adão. 2021. A Comparative Study of Automatic Program Repair Techniques for Security Vulnerabilities. In *2021 IEEE 32th International Symposium on Software Reliability Engineering (ISSRE)*.

[19] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A Manually-Curated Dataset of Fixes to Vulnerabilities of Open-Source Software. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 383–387. https://doi.org/10.1109/MSR.2019.00064

[20] Ying Zhang, Mahir Kabir, Ya Xiao, Na Meng, et al. 2021. Data-Driven Vulnerability Detection and Repair in Java Code. *arXiv preprint arXiv:2102.06994* (2021).