JLeaks: A Featured Resource Leak Repository Collected From Hundreds of Open-Source Java Projects

Tianyang Liu
lty@bit.edu.com
Beijing Institute of Technology
Beijing, China

Weixing Ji*
jwx@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Xiaohui Dong
xhdong@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Wuhuang Yao
yaowuhuang@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Yizhuo Wang
frankwyz@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Hui Liu
liuhui08@bit.edu.cn
Beijing Institute of Technology
Beijing, China

Haiyang Peng
haiyangpeng2@gmail.com
Beijing Institute of Technology
Beijing, China

Yuxuan Wang
wangyuxuan@bit.edu.cn
Beijing Institute of Technology
Beijing, China

ABSTRACT

High-quality defect repositories are vital in defect detection, localization, and repair. However, existing repositories collected from open-source projects are either small-scale or inadequately labeled and packed. This paper systematically summarizes the programming APIs of system resources (i.e., file, socket, and thread) in Java. Additionally, this paper demonstrates the exceptions that may cause resource leaks in the chained and nested streaming operations. A semi-automatic toolchain is built to improve the efficiency of defect extraction, including automatic building for large legacy Java projects. Accordingly, 1,094 resource leaks were collected from 321 open-source projects on GitHub. This repository, named JLeaks, was built by round-by-round filtering and cross-validation, involving the review of approximately 3,185 commits from hundreds of projects. JLeaks is currently the largest resource leak repository, and each defect in JLeaks is well-labeled and packed, including causes, locations, patches, source files, and compiled bytecode files for 254 defects. We have conducted a detailed analysis of JLeaks for defect distribution, root causes, and fix approaches. We compare JLeaks with two well-known resource leak repositories, and the results show that JLeaks is more informative and complete, with high availability, uniqueness, and consistency. Additionally, we show the usability of JLeaks in two application scenarios. Future studies can leverage our repository to encourage better design and implementation of defect-related algorithms and tools.

*Corresponding author

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

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639162

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Empirical software validation**.

KEYWORDS

Resource leak, Defect repository, Open-source projects, Java language

ACM Reference Format:
Tianyang Liu, Weixing Ji, Xiaohui Dong, Wuhuang Yao, Yizhuo Wang, Hui Liu, Haiyang Peng, and Yuxuan Wang. 2024. JLeaks: A Featured Resource Leak Repository Collected From Hundreds of Open-Source Java Projects. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3639162

1 INTRODUCTION

Memory and resource leaks in software usually lead to system performance degradation or even system crashes[19, 25, 49]. However, finding them in the early stage of the software development is chanllenging, as they can only be triggered when resource allocation accumulates to a certain extent. Memory leaks occur when unused objects are not reclaimed, while resource leaks occur when finite system resources, such as files, sockets, and threads, are not explicitly released in time.

A defect repository helps to design and validate resource leak detection, location, and repair tools. Additionally, real-world defects can provide more insights and encourage researchers to resolve practical problems. It can also be used to test existing algorithms and define future improvements. However, resource leaks collected from the real world are scarce. Generally, a large data set can be automatically generated, such as *SIR* [16], according to known causes and patterns. However they are usually simple in pattern, excluding unknown patterns resulting from new language features. The manually crafted defects are mostly invented and can be quite different from the real-world samples.

It is a challenging task to collect real-world defects from open-source projects. ClabureDB [47] includes 7 memory leaks and 13 resource leaks from the Linux kernel. Machida et al. [34] investigated 5 open-source Java projects related to cloud computing and found 55 leaks. The authors of [38] found 62 resource leaks from Tomcat. Ghanavati et al. [19] collected 238 memory leaks and 253 resource leaks from 15 open-source Java projects. It is one of the largest resource leak repositories built from the real world. Though several existing repositories are available, several important issues should be addressed for widespread use. First, the number of high-quality leaks from real projects is too small to build models using machine learning approaches. Second, a significant amount of code duplication is observed in existing repositories due to the limited number of projects investigated. Third, the availability of some repositories decreases over time. For instance, some of the links to bug reports are currently unavailable.

In this paper, we present a resource leak repository collected from real-world projects on GitHub. We find high-quality Java projects with predefined rules and patterns. To ensure the accuracy of the resource leak repository, we conduct a two-round manual cross-validation. In addition, we automatically compile some projects that contain resource leaks to generate bytecode files for ease of use. Based on this, we report a high-quality resource leak repository JLeaks, labeled with extensive detail after round-by-round filtering and cross-validation. To the best of our knowledge, JLeaks is the largest and most informative resource leak repository so far. One of the key strengths of JLeaks is its high availability, providing researchers with not only GitHub commit links but also source files, patches, and other relevant information. JLeaks contains significantly fewer defects of duplicate code compared with other repositories. The main contributions of this paper include:

- We introduced a semi-automated workflow designed to extract real-world resource leaks from GitHub. Our approach aimed to reduce labor costs by applying filters, including keywords, and automatic compilation.
- We built JLeaks, a public repository available on GitHub, which contains 1,094 resource leaks from 321 open-source projects. JLeaks is built after multiple rounds of filtering and validation from more than 3,185 commits. Each defect is well-labeled in detail, including the defect code, patches, key variables, and location, providing convenience for defect code analysis, automatic testing, and tool evaluation.
- We compared JLeaks with DroidLeaks [33] and Leak19 [19] regrarding data informativeness, diversity, and quality. The results show that JLeaks is more informative with high availability, uniqueness, and consistency. Furthermore, we used JLeaks to evaluate some existing tools and subsequently summarized the strengths and limitations of each tool.

The remainder of the paper is organized as follows. In Section 2, we systematically investigate the resource management system of files, sockets, and threads in Java programming language. We describe the workflow of defect extraction from GitHub in Section 3. Based on this, we analyze resource leaks and present the results in Section 4. The application is presented in Section 5. Section 6 discusses our findings, and threats to validity are discussed in Section 7. Section 8 overviews the relevant literature, and Section 9 concludes this paper.

2 BACKGROUND

In Java, both memory and resources should be released timely for their limited availability. The garbage collector (GC) only frees data objects in memory, not system resources held by these objects, and some programmers mistakenly assume that both will be recycled by GC [46]. we summarize the resource usage models for file, socket, and thread to understand how system resources are allocated and released using existing APIs.

2.1 File and Resource Leak

In Java, file handles, generally associated with Java instances, are one of the most common resources managed by the operating system. There are varied Java streams for opening files and reading/writing data, which are associated with *FileDescriptor* instances directly or indirectly. Figure 1 provides an example using *input* in *java.io* to illustrate the relationship among streams in *open* and *close* operations, wherein *xxxInputStream* refers to byte streams and *xxxReader* represents character streams. As the primary stream in *java.io*, *FileInputStream* is directly associated with the *FileDescriptor* instance. Other streams wrap *FileInputStream* in one of their fields as their data source, transforming the data or providing additional functionality. *FileReader*, *FileInputStream* and *RandomAccessFile* are three entry classes that operate directly on the opening and closing of file handles (*FileDescriptor*). Especially, *FilterInputStream* [37] wraps some other input streams to provide additional functionality, using the wrapped-in instances as its basic data source and transforms the data along the way. Subclass of *FilterInputStream* can provide additional methods and fields. *FilterReader* also operates in this manner. The dotted lines in Figure 1 indicate that these two classes are directly related, but the internal logic follows solid lines.

The code snippet in Figure 2 is extracted from *PhoenicisOrg/phoenicis* and labeled using the model in Figure 1. If there are no exceptions, the stream is nested along *FileOutputStream-BufferedOutputStream* and closed layer by layer after line 3 (represented as close ① and close ② in Figure 2). However, if an exception is thrown in line 1, line 2, or line 3, the resource will not be properly closed.

2.2 Socket and Resource Leak

When programmers initialize a *Socket* object, the operating system creates a *Socket* and attaches its handle to a *FileDescriptor* instance. *SocketInputStream*, a subclass of *FileInputStream*, is returned by the method *getInputStream()* of class *Socket*. Both *Socket* and *SocketInputStream* can release socket resources.

The connections based on *Socket* include network connection, database connection, etc. Similar to closing file resources, not all classes can release the connections. For instance, *sun.net.www.protocol.http.HttpURLConnection* serves as an example in Figure 1. It indirectly calls *getInputStream()* in *Socket* to get a wrapped *SocketInputStream* object, which is used for receiving data in the network. The same applies to some *InputStream*, which uses *SocketInputStream* as a field. Therefore, these classes are capable of releasing socket resources. In particular, if the connection is to be kept alive, *KeepAliveStream*, which cannot close the inner stream including

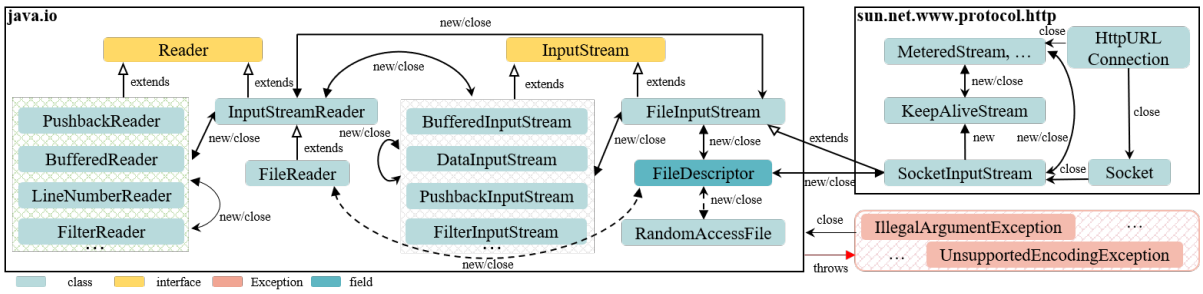


Figure 1: Relationship of classes related to I/O streams and sockets.

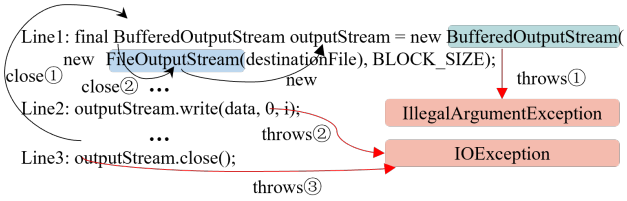


Figure 2: A file resource leak from PhoenicisOrg/phoenicis.

socket resources, will be wrapped. Consequently, calling *close()* on *KeepAliveStream* will not release the socket resources. Note that only one of the paths can be selected between path {*MeteredStream, ... - KeepAliveStream-SocketInputStream*} and path {*MeteredStream, ... - SocketInputStream*}.

2.3 Thread and Resource Leak

In an operating system, threads are a limited and precious resource. If too many threads are created and not destroyed entirely in time, it may eventually lead to resource leaks. There are two categories of thread leaks:

Unclosed and unused threads. If a thread remains active but is no longer in use, it can cause a leak in the system.

Unclosed thread pool. A thread pool makes threads reusable, but GC cannot automatically recycle them. Therefore, programmers need to destroy the thread pool explicitly.

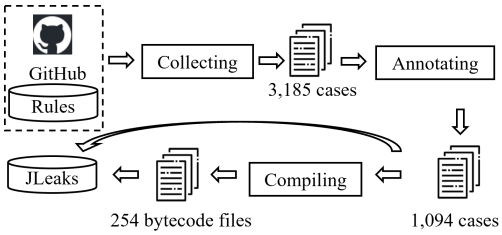


Figure 3: The workflow of building JLeaks.

Table 1: Rules for filtering projects and commits

Rules for project	Keywords for commits	
	Message	Code
# of commits > 20	leak	close
# of issues > 20	leakage	stop
# of contributors > 10	close	shutdown
# of releases > 0	closed	try(*)
# of stars > 500	release	CloseUtil
duration > 50 weeks	resource	-

3 APPROACH

Figure 3 shows our workflow, which has three steps: collecting, annotating, and compiling. Using this workflow, we created JLeaks, a publicly accessible repository of resource leaks, hosted on GitHub[1].

Collecting. GitHub is a well-known and popular version control system for open-source projects. Each commit on GitHub typically includes a message and a *diff code*. A commit message, written by programmers, describes the purpose or intention of this commit. A *diff code* shows the difference in code before and after the commit. The code starting with the "+" symbol denotes fix code, while the code starting with the "-" symbol is defect code.

First, we select suitable projects from GitHub. It is essential to filter out unsuitable projects to ensure accurate analysis and reliable results [5, 27, 39]. According to the findings in [2], we select projects based on the rules specified in the first column of Table 1.

Second, we perform two major steps to analyze commits. Based on [19] and [33], we filter commit messages using keywords shown in the second column of Table 1. Based on Section 2 and [33], we then formulate some keywords for code as shown in the third column of Table 1. If either the fix code or the defect code contains any of the words listed in the third column of Table 1, this commit likely fixes a resource leak.

Finally, we remove irrelevant and non-Java source files, commits caused by git-merge operations, and meaningless data in test cases.

Annotating. We carefully inspect each commit by analyzing the commit messages and the *diff code* to reduce false positives. This process is done manually using cross-validation.

Five experienced Java developers from the industry were employed to annotate the commits. All of them have been using Java for at least 8 years. Afterward, three students carefully examined

[1]https://github.com/Dcollectors/JLeaks

Table 2: Cohen's Kappa metric.

Dimension	Cohen's Kappa
resource leak	0.88
resource type	0.74
root cause	0.74
fix approach	0.92

the data. If there are any conflicts, we hold joint discussions. It is important to mention that we used at least two developers to annotate each defect case, ensuring accuracy and consistency. In total, we collected 1,094 <$defect$, $patch$> pairs from 321 open-source projects.

We compute the consistency of two-round annotation using Cohen's Kappa metric [10, 15], as shown in Table 2. It shows an "almost perfect agreement" on whether the commit is fixing a resource leak, and fix approaches. The Cohen's Kappa of resource types and root causes are relatively low. This is because it can be challenging to determine the resource type when it is wrapped in multiple layers, and complex code structures always influence checking out root causes. A consensus has been reached through in-depth analysis and discussions of conflicting cases.

Compiling. Though source code defect detection has been explored extensively, a number of popular defect detection tools rely primarily on bytecode files, such as SpotBugs [48] and Infer [17]. This is because source code may be incomplete or unavailable in practice. We provide the minimum and complete source files for each defect in JLeaks and try our best to compile these source files (where possible) to generate bytecode to improve the usability of our repository.

First, we compile all open-source projects with default configurations and building tool chains. However, automatically downloading all the dependencies for outdated projects (with outdated dependencies) is challenging, and only 86 project versions were successfully complied by default. Second, for defect files, we build an automatic and iterative wrapper of building tools to enumerate remote maven repositories and fix partial package missing errors by package generation and interposition. Finally, 254 defects were packed with both source files and their bytecode files.

JLeaks, a resource leak repository, supports research of defect-related algorithms by providing project information, defect information, code characteristics, and file information. Project information covers project details (name, summary, timestamps), maintainability metrics (commits, releases, contributors, pull requests), and popularity metrics (stars, issues, forks). Defect information includes commit URLs, code line ranges, defect methods, modified lines, resource types, root causes, and fix approaches. Additionally, JLeaks captures code characteristics such as the standard and third-party libraries, whether the defect is inter-procedural, and key variable attributes. Researchers also have access to file information, including hash values and URLs for defect and non-defect files, the defect and non-defect source files, some bytecode files, and the defect or non-defect methods files. This allows hands-on exploration and experimentation, empowering researchers to develop advanced defect-related algorithms.

4 REPOSITORY ANALYSIS

4.1 Research Questions

In this section, we study all the resource leaks and present the multi-dimensional analysis by answering the following questions:

- RQ1 **(Defects Distribution):** What is the distribution of defects?
- RQ2 **(Root Causes):** What is the root cause of each defect? What are the common causes of defects?
- RQ3 **(Fix Approaches):** How do programmers fix resource leaks? Are there some common repair patterns?
- RQ4 **(Patch Correctness):** Do these patches correctly fix the leaks?
- RQ5 **(Repository Comparison):** What are the advantages of JLeaks compared with other resource leak repositories regarding informativeness, diversity, and quality?

Research question RQ1 addresses the overall distribution of selected projects and defects in JLeaks. To prove that our selected projects are well-maintained and popular, we draw the distribution of projects from the aspects of commits, duration, stars, issues, and releases. We also calculate the proportion of different resources and interprocedural defects to aid in the design of defect detection tools. Furthermore, we count the number of standard and third-party libraries in resource leaks and hope developers can pay special attention when using these libraries. Research question RQ2 delves into understanding the root causes of resource leaks. By answering RQ2, we discover the common root causes for different resources, which is essential for detecting resource leaks in open-source projects. Research question RQ3 investigates the appropriate patches for different resources and defect causes, aiding in patch generation. Research question RQ4 explores the correctness of the fix approaches. We summarize common incorrect patterns to avoid reintroducing resource leaks during patch generation. Research question RQ5 compares JLeaks with two resource leak repositories regarding informativeness, diversity, and quality. It helps to select suitable resource leak repositories for different research purposes.

4.2 RQ1: Defects Distribution

In this section, we present the distribution of selected projects, and the distribution of collected defects.

In this work, 1,094 resource leaks were collected from 321 projects, including 869 different project versions. Figure 4 shows the features of the 321 projects and it can be observed that these projects are *well-maintained*. They have a median maintenance duration of 458 weeks. Most of them are active and under development, with a median of 4,814 commits and 133 issues. These projects are popular as their average stars are 5,067, and half have over 1,822 stars. The median of releases is 35 and the selected project contains at least one release.

JLeaks includes all types of system resources mentioned in Section 2, including 797(73%) file leaks, 279(26%) socket leaks, and 18(2%) thread leaks. Although the number of sockets and threads is relatively small, they are highly valuable due to their lower usage frequency and stronger invisibility. A surprising finding is that the majority of resource leaks are not interprocedural. Specifically, the instance that wraps with system resources does not propagate to other methods within the project. Consequently, resource leak

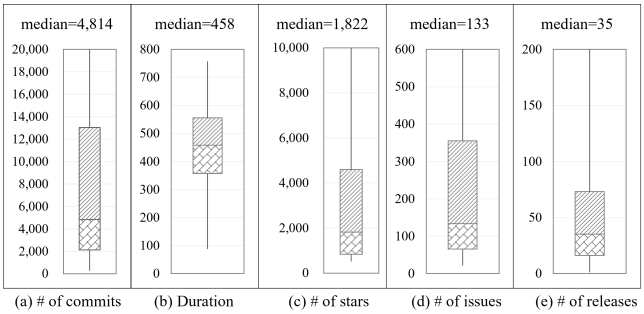


Figure 4: Features of selected projects.

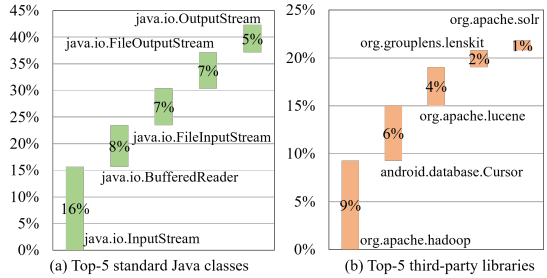


Figure 5: Top ranked libraries of resource leaks in JLeaks.

detection tools may yield improved detection results at the method level.

Investigating the standard Java and third-party libraries used in resource leaks is extremely necessary. It can guide future research on defect detection and remind programmers to be cautious when using these libraries. Figure 5 lists the top-5 standard Java classes and the top-5 third-party packages in JLeaks. Only the first three layers of package names are displayed for third party libraries. File handles wrapped in *java.io.InputStream* and *java.io.BufferedReader* are most likely to leak in standard Java libraries. The reason is that they are the most frequently used class for reading files. As mentioned in Section 2.1, *java.io.InputStream* is a basic stream, and *java.io.BufferedReader* simplifies file reading and writing. Programmers should pay more attention to *org.apache.hadoop*, *android.database.Cursor*, and *org.apache.lucene*, as they are top-ranked in third-party libraries.

Table 3: The root causes of resource leaks.

Short	Causes	Description	# defects
C1	noCloseEPath	No close on exception paths	701
C2	noCloseRPath	No close on regular paths	375
C3	notProClose	Not provided close()	11
C4	noCloseCPath	No close for all branches paths	7

4.3 RQ2: Root Cause Analysis

This section aims to categorize the root causes of resource leaks. From a scope perspective, they can be classified into three categories: intra-procedure, inter-procedure, and inter-library. If a class has a field that holds a resource, but the class itself does not provide a method to close the resource, it can potentially cause a resource leak. This situation (notProClose) can also be an inter-library defect. From a code structure perspective, we can identify the root causes of three types of defects, i.e., "No close on exception paths (noCloseEPath)", "No close on regular paths (noCloseRPath)", and "No close for all branches paths (noCloseCPath)". We can get the location (*cl*) where the resource should be closed based on the patch provided by the programmer. If there is no *close()* for the resource in the corresponding method of the defect file, the root cause is *noCloseRPath*. Otherwise, we start a backward analysis from *cl* and identify the smallest code block containing *cl* to determine the defect's root cause. If the code block is a "catch" or "finally", the root cause is classified as *noCloseEPath*. If the code block is an "if" or "else", the root cause is *noCloseCPath*. Table 3 shows the proportion of these four root causes of defects in JLeaks. A heat map on the left of Figure 6 illustrates the relationship between resource types and root causes. Next, we show a detailed description and analysis of each cause.

No close on exception paths. Exceptions are unavoidable during program execution and should be captured to release resources. About 64% of resource leaks are caused by *noCloseEPath*. The proportion for this cause in file, socket, and thread are 65%, 62%, and 44% respectively. Listing 1 shows a code fragment from "antlr/antlr4", a famous and powerful parser generator. The resource cannot be released if an exception is thrown in *w.write(context)*. In addition, the *try-with-resources* introduced in JDK 1.7 provides a concise way to release resources automatically. However, its improper use by programmers can lead to new resource leaks, especially in wrapping operations. In Listing 2, *new FileInputStream(getOutputFile(extension))* cannot be closed if *new TarArchiveInputStream()* fails to close.

No close on regular paths. "Regular path" refers to the sequential execution path from allocating a resource to the end of its use or method exit. *noCloseRPath* means that there is no operation to release the resource in regular paths. About 34% of the studied resource leaks are caused by *noCloseRPath*. In Figure 6, we can see that the proportion of this cause in file, socket, and thread are 33%, 37%, and 44% respectively, which indicates that *noCloseRPath* is a common cause of resource leaks. Besides forgetting to call *close()* for resource objects created explicitly, developers often overlook the need to close anonymous resource objects.

Not provided close(). In this case, member variables in a class hold resources after initialization or configuration setting, but an error method or no method is provided to close these resources. This lack of *close()* can result in potential resource leaks, particularly when other programs or classes invoke the class.

No close for all branches paths. The taken branch of conditional statements is decided at run-time, and any given branches can be selected. Resource should be released on all paths if no deallocation is performed after the branch converges. We collected 7 resource leaks caused by *noCloseCPath*. Listing 3 shows an example in "apache/zookeeper", a distributed configuration service for large-scale distributed computing that has over 10k stars. In this example, *cnxn* holds the resource and should be closed, regardless of whether *LOG.isDebugEnabled()* is true or false. However, it fails to close the resource held by *cnxn* if *LOG.isDebugEnabled()* is false.

Listing 1: An example of noCloseEPath from "antlr/antlr4".

```
/* version : ba6c711e85a6d8c3e287d39b3ee5c99ee77df747 */
public static void writeFile (String fileName, String content)
throws IOException {
    FileWriter fw = new FileWriter(fileName);
    Writer w = new BufferedWriter(fw);
    w.write(content);
    w.close ();}
```

Listing 2: An example of noCloseEPath from "opensearch-project/OpenSearch".

```
/* version :110 cef7882863cd3762631ec5f975e15f433dfc07 */
try (TarArchiveInputStream tar = new TarArchiveInputStream(
wrapper.apply(new FileInputStream(getOutputFile(extension ))))) {...}
```

Listing 3: An example of noCloseCPath from "apache/-zookeeper".

```
/* version : d8adc547f9856747905b7d46450f13fa98df147f */
if (cnxn != null) {
    if (LOG.isDebugEnabled()) {
        LOG.debug("Closing " + cnxn);
        cnxn.close ();}}
```

4.4 RQ3: Fix Approaches

Each commit message in GitHub contains the buggy code and its patch. Based on these patches, we summarize the approaches to fix resource leaks into five strategies, as listed in Table 4.

Table 4: Fix approaches for resource leak.

Short	Fix Approaches	Description	# defects
F1	try-with	Use try-with-resources	498
F2	CloseInFinally	Close in finally	431
F3	CloseOnEPath	Close on exception paths	100
F4	CloseOnRPath	Close on regular paths	53
F5	AoRClose	Add or rewrite close	12

Use try-with-resources. *try-with-resources* is a syntactic sugar introduced in JDK 1.7, which automatically closes resource in a method without the need of *finally*. The *AutoCloseable* interface must be implemented for resources that can be closed with *try-with-resources*. As it is simple to use, *try-with-resources* is the most popular way of fixing resource leaks, accounting for nearly 46% of the total.

Close in finally. To ensure that resources can be closed regardless of whether an exception occurs, programmers place *close()* into the *finally* block. As the code in *finally* is guaranteed to be executed, this approach is a prevalent choice among programmers to close resources. *CloseInFinally* is ranked second, accounting for nearly 39% of the total.

Close on exception paths. To avoid the possibility of resource closing failure in the event of an exception being thrown, programmers frequently surround resource operations with *try-catch*. This approach has been used to fix 100 defects in JLeaks.

Close on regular paths. In this case, programmers add *close()* to the regular path defined in Section 4.3. This approach accounts for 5% of the fix approaches we have studied. However, this is unsafe as exceptions may be thrown before the close statement.

Add or rewrite close method. This fix approach rewrite *close()* for the causes of *notProClose*.

The right of Figure 6 is a heat map illustrating the relationship between resource types and fix approaches, and Figure 7 investigates the impact of root causes on the patches, from which we get the following observations. First, *try-with* and *CloseInFianlly* are the most commonly employed fix approaches for file and socket resources. This may be because using *try-with-resources* and *finally* are the two most straightforward approaches to guaranteeing that resources are correctly closed. For thread, *CloseOnEPath* is the most frequently utilized because thread leak rarely occurs and most programmers believe that thread destruction on the regular path is safe. Second, despite the root cause being *noCloseRPath*, some programmers directly call *close()* into the regular path. *CloseOnEPath* is unsuitable for the defects caused by *notProClose* and *noCloseCPath*.

4.5 RQ4: Patch Correctness

An appropriate patch should be capable of preventing the recurrence of resource leaks. Nevertheless, patches submitted by programmers may not operate as intended. Ensuring the absence of vulnerabilities is challenging [11–13]. Therefore, assuming that the patched code is defect-free is not reliable. By analyzing the collected resource leaks and their patches, and following the models given in Section 2, we summarize three incomplete repair patterns in Figure 8. Based on these patterns, we find that 67 patches still have potential resource leaks, accounting for 6%. These unsafe and incomplete patches are also confirmed by experienced developers from the industry.

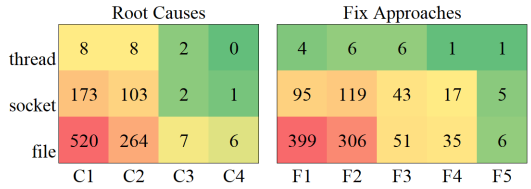


Figure 6: Relationship between resources type, causes, and fix approaches.

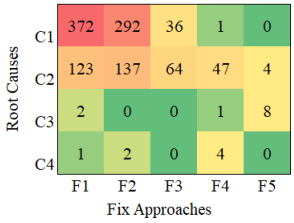


Figure 7: Relationship between root causes and fix approaches.

The most common unsafe fix approach directly calls *close()* on the regular path, unaware of exceptions. As shown in Figure 8 (a), the

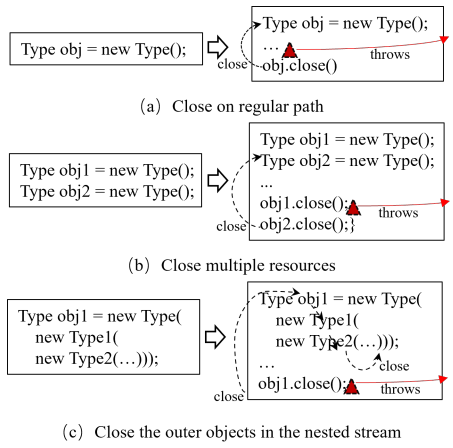


Figure 8: Examples of three incomplete repair patterns.

resources held by *obj* cannot be released if an exception is thrown before the execution reaches *obj.close()*, leading to a potential resource leak. Another unsafe fix approach is observed when multiple resources need to close simultaneously. In such cases, failure to close the previous resources can also result in failure to close the subsequent resources. As shown in Figure 8 (b), if an exception is thrown in *obj1.close()*, the resource held by *obj2* cannot be released. As illustrated in Figure 8 (c), it is common to create nested objects. However, calling *close()* on the outermost object may not ensure the release of the inner resources. Although *try-with-resources* is helpful for automatically closing the resources, it is important to note that if an exception occurs during the closing of the outer resources, the inner resources may not be properly closed. As shown in Figure 8 (c), the resources held by anonymous variables *new Type1()* and *new Type2()* cannot be released if an exception is thrown in *obj.close()*, leading to a resource leak.

4.6 RQ5: Repository Comparison

This section compares JLeaks with two well-known repositories DroidLeaks [33] and Leak19 [19] on their informativeness and completeness, diversity and overlap, and quality. Other defect repositories, e.g, Defects4j [26] and Bears [35], contain different defects with few resource leaks, making them unsuitable for comparative experiments with JLeaks.

Informativeness & Completeness. Table 5 compares informativeness and completeness, in which a circle indicates that the annotated items provided by the original repository are not very accurate and precise. For example, DroidLeaks only shows method names for defect locations instead of line numbers. The precise location is advantageous in evaluating defect detection tools and algorithms. The resource type (file, socket, thread) is also not specified in DroidLeaks, prompting us to manually annotate all the records in DroidLeaks for detailed comparison with JLeaks. For source files and patches, DroidLeaks provides only links. Similarly, Leak19 also offers links exclusively for patches.

DroidLeaks provides links to fix revisions, buggy revisions, buggy methods, the concerned classes, defect file name, and leak extent for each record. Among 298 defects, 6 are already unavailable from the Internet for website update or migration. Leak19 is built based on issues from Bugzilla, JIRA, and GitHub. Only links to bug or issue reports are provided. To prevent these resources from becoming inaccessible, JLeaks provides source code for all defects, and bytecode files for some defects. This facilitates the evaluation of existing tools and algorithms, as extracting the code of defects from these large projects and compiling them into bytecode are also very time-consuming and laborious tasks.

All three repositories conduct an empirical study, which aids in defect detection, patch generation, and defect interpretability. They explore the root causes of resource leaks, with JLeaks and Leak19 additionally presenting the fix approaches for these defects. Leak19 and JLeaks give the specific analysis results for each data case, while DroidLeaks creates a website to provide public access. Additionally, JLeaks offers defect method names, accompanied by their start and end line numbers, to support the precise benchmarking defect-detecting tools and algorithms. JLeaks also provides locations and attributes of resource variables, which serve as a baseline for defect localization studies. In comparison, JLeaks provides richer defect-related information.

Diversity & Overlap. These three resource leak repositories differ in project and defect selection, and data distribution. DroidLeaks has 298 resource leaks collected from 32 Android projects hosted on GitHub, and Leak19 has 253 resource leaks collected from 15 large projects hosted on GitHub, Bugzilla, and JIRA. DroidLeaks defines four criteria to select projects: popular, traceable, actively-maintained, and large-scale. Leak19 selects 15 large-scale, well-established, and well-developed applications. The projects in JLeaks are selected using the rules shown in Table 1. The proportion of different resources in these three repositories exhibits significant variation, as shown in Table 6. DroidLeaks predominantly comprises of sockets and files, accounting for 62% and 37%, respectively, while there are 2 thread leaks. This trend can be attributed to the nature of Android projects. Leak19 contains 73 thread-related defects, making it the repository with the most thread cases among these three. JLeaks has the most file and socket cases, with 797 and 279 respectively. It is interesting to note that the three repositories have individual characteristics, and complement each other. Combining these repositories enables better benchmarking of defect detection tools and algorithms.

It is important to explore the overlap among JLeaks, DroidLeaks, and Leak19. A Venn diagram illustrating the relationship among three repositories is shown in Figure 9, in which Figure 9 (a) displays the overlap of projects and Figure 9 (b) presents the overlap of defects. Overall, there is less overlap between the three repositories. This is because different filter rules are applied in project defects selection.

An extended question is why JLeaks involves so many projects. Both DroidLeaks and Leak19 collected several resource leaks from a small group of projects. On average, DroidLeaks has 9 resource leaks per project, while Leak19 has 17 resource leaks per project. However, on average, there are only 3 resource leaks per project in JLeaks. To clarify the underlying reasons, we perform code clone detection on defect code extracted from top-5 projects in DroidLeaks using Allamanis [3]. Since no source files are included in the repository, Leak19 is not tested for clone detection. However, we inspect the records in Leak19 one by one in the overlapped projects. The

Table 5: Informativeness & completeness comparison.

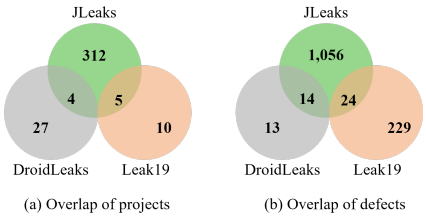
Repository	Project	Type	Link	Location	Root cause	Source files	Bytecode	Patch	Patch correctness
DroidLeaks	✓	×	✓	○	✓	○	✓	○	×
Leak19	✓	✓	✓	×	✓	×	×	○	×
JLeaks	✓	✓	✓	✓	✓	✓	✓	✓	✓

results show that both DroidLeaks and Leak19 contain many similar records. Usually, these records are submitted by developers to fix similar resource leaks at different locations of the same project. For example, in DroidLeaks, the methods "getInt", "getFloat", and "getBool" comes from the same file in "AnkiDroid" are very similar. JLeaks removes similar code blocks in the manual annotation. As a result, the number of overlapped defects is also reduced.

Table 7 presents the top-5 projects with the most defects in different repositories. DroidLeaks, Leak19, and JLeaks contain numerous resource leaks in some large-scale projects. The number of defects collected from the same project *HBase* in JLeaks is much larger than that in Leak19. After in-depth analysis, we found that there are 14 defects included in both Leak19 and JLeaks. Among the 28 defects included only in JLeaks, 18 defects are committed to the codebase after the date when Leak19 was built, and others are missed by Leak19 for it only uses the keyword "Leak" in the filter stage. We have reached the conclusion that a considerable portion of resource leaks in JLeaks are also collected from a limited number of projects. Comparatively, JLeaks covers a wide range of projects that are active and popular in practice.

Table 6: Distribution of resource types in three repositories.

Repository	# file	# socket	# thread
DroidLeaks	108	**182**	2
Leak19	129	51	**73**
JLeaks	**797**	**279**	18



(a) Overlap of projects (b) Overlap of defects

Figure 9: Overlap between JLeaks, DroidLeaks, and Leak19.

Quality. Based on [12] and the standardized data quality framework ISO/IEC 25012[1], we outline three data quality attributes: uniqueness, consistency, and currentness. It is important to note that Leak19 does not provide source files at the method level, thus limiting the evaluation to DroidLeaks and JLeaks. We have employed Allamanis [3], a code cloning detection tool, for uniqueness and consistency. Syntactic cloning consists of three types:

Type-1: The code remains identical after eliminating spaces, blank lines, and comments.

Type-2: The code is identical, except for identifier names.

Table 7: Top-5 projects with the most defects in repositories.

Repository	Project	# defects	# stars	# issues	# commit
DroidLeaks	ChatSecure	31	1,075	164	2,909
	K-9 Mail	31	8,133	672	12,638
	Wordpress	29	2,803	945	76,025
	AnkiDroid	28	6,665	437	17,652
	Quran	19	1,735	272	3,650
Leak19	hadoop	76	13,676	838	26,767
	hbase	29	4,918	180	4,918
	activemq	26	2,169	39	11,446
	hive	25	4,915	94	16,867
	cassandra	16	8,101	312	25,598
JLeaks	hbase	42	4,918	180	4,918
	jenkins	34	20,999	72	33,906
	lucene	31	1,781	2,097	36,573
	alluxio	29	6,274	857	34,034
	dragonwell8	27	3,943	117	91,027

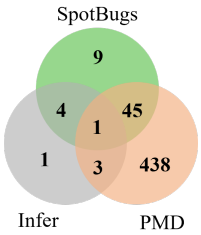


Figure 10: Overlap of defect detection results between PMD, SpotBugs, and Infer.

Type-3: The code remains similar, despite a few statements that are added, deleted, or modified.

Uniqueness of the data within a repository is beneficial for researching defect-related algorithms, such as training defect detection models [3, 12]. To evaluate the uniqueness, we focus on the methods with defects and extract these methods based on method names from DroidLeaks and the method boundaries from JLeaks. Due to invalid links and unmatched method names provided by DroidLeaks, we identified only 276 defect methods from DroidLeaks and 1,094 defect methods from JLeaks. Allamanis is applied to detect Type-3 clones among these defect methods. A uniqueness score is yielded with $uniqueness = \frac{allNum - dNum}{allNum} \times 100\%$, where $allNum$ is the total number of defect methods and $dNum$ denotes the number of the identified duplicate methods.

Consistency is employed as a measure to evaluate whether there are conflicting annotations. In theory, similar code fragments should have consistent labels. If the fix code and the defect code are identified as a pair of Type-1 level clone code, it indicates the presence of inconsistent labels in the repository. We only consider Type-1 in consistency evaluation, because slight functional changes can

result in variations in functionality between the defect code and the fix code [12]. For consistency, we apply Allamanis to identify methods between defect and fix methods. We assess the consistency using $consistency = \frac{allNum-cNum}{allNum} \times 100\%$, where $allNum$ denotes the total number of defect methods, and $cNum$ represents the count of defects that duplicate the fix methods.

Currentness aims to ensure that the repository has the same temporal characteristics as its application context [12]. After sorting by defect introduction time, we denoted the original and current data as the oldest and newest half of JLeaks. A currentness score is yielded with $currentness = (1 - J) \times 100\%$, where J is Jenson-Shannon divergence, a measure matric for the statistical distance of the original and current data.

Eventually, JLeaks achieves a uniqueness score of 88%, a consistency score of 93%, and a currentness score of 77%. DroidLeaks sores 64% for uniqueness, 71% for consistency and 73% for currentness. Compared with DroidLeaks, JLeaks exhibits higher uniqueness, consistency and currentness. This can be attributed to the manual labeling process, during which some duplicate defects were filtered out. Additionally, the currentness of both JLeaks and DroidLeaks are relatively low. In general, it is practical to include specific outdated projects. Resource leaks are only triggered when resource allocation accumulates to a certain extent, leading to extended repair times. As a result, a limited number of resource leaks were collected in new versions. We focused on well-known projects with widespread and extensive usage, enhancing the chances of uncovering defects. Furthermore, many significant but outdated projects are still in operation and maintenance, urgently requiring defect detection and repair approaches.

5 APPLICATION

5.1 Evaluation of Defect Detection Tools

We evaluated three open-source defect detection tools (PMD [40], Infer [17], and SpotBugs [48]) using JLeaks. PMD is a source code analyzer that can find common programming defects. SpotBugs is a program that uses static analysis to look for bugs in Java. It can find many types of resource leaks, including "Method may fail to close stream(OS_OPEN_STREAM)" and "Method may fail to close stream on exception(OS_OPEN_STREAM_EXCEPTION_PATH)". Infer commonly reports the "exceptions skipping past close() statements" issue for resource leaks in Java. In the experiment, PMD detects source code files containing 1,094 defects, SpotBugs analyzes bytecode files with 254 defects, and Infer operates on fully compiled projects with 76 defects. The difference in input number between Infer and SpotBugs is that some projects may not be fully compiled, yet bytecode files with defects have already been generated. Upon completing detections, we check whether the resource leak defect is correctly detected in the method. We are able to extend the inspection granularity to the method level, thanks to the precise tagging of defect methods in JLeaks.

Table 8 presents the evaluation results of PMD, SpotBugs, and Infer. Figure 10 displays the Venn graph of their detection results. We observe a higher accuracy for PMD and a lower accuracy for Infer. After statistical analysis, we discovered that these tools have limitations in detecting resource leaks caused by third-party libraries, threads, *noCloseCPath*, and *notProClose*. For cases where PMD failed, the proportion of resource variable types related to third-party libraries is up to 358/607=59.0%, as the source code of third-party libraries is not included. Surprisingly, the accuracy of SpotBugs is only 23.2% and it fails to report all the defects caused by *noCloseCPath* or *notProClose*. Among the methods successfully detected, 59/59=100% use standard libraries, indicating that SpotBugs also has difficulty in identifying resource leaks from third-party libraries. The accuracy of Infer is only 11.8% and falls into 1/16=6.3% when the defects are related to third-party libraries.

5.2 Evaluation of GPT-3.5

LLM (Large Language Model) undergoes pre-training on vast amounts of source code and natural language data, giving it a remarkable ability to understand code structure and generate code [51]. Consequently, LLMs have been increasingly applied to software engineering, such as code defect detection [9], program repair [14, 22, 55], and code generation [7, 8, 18, 28, 36]. Both GPT-3.5 and GPT-4 models can perform classification tasks without any additional training [20]. However, it should be noted that when GPT-3.5 performs binary defect detection tasks on real-world datasets, its accuracy is only 51% [9]. Therefore, we evaluate GPT-3.5 only for its capability to commits classification and resource leak detection.

Commits Classification. We assume that ChatGPT can be used to help with the preliminary filtering of commits. We randomly select 100 commits as the test set, in which there are 63 cases that have nothing to do with system resources.

We measure the capability of GPT-3.5 using accuracy $Acc = \frac{TP+TN}{TP+TN+TN+FN}$, false-positive rate $FPR = \frac{FP}{FP+TN}$, and false-negative rate $FNR = \frac{FN}{TP+FN}$. In the experiment, we employed a comprehensive prompt. This prompt, which meets the Prompt Engineering Guide [45], is mainly comprises of various elements, including a task description, resource explanation, typical scenario illustration, sample code fragments, and the expected output. In general, for GPT-3.5, the more examples in the prompt, the better its detection performance [6, 28]. Therefore, We conducted experiments using varying numbers of randomly selected examples from JLeaks, ranging from 0 to 15. Figure 11 displays the trends of *ACC*, *FPR*, and *FNR* with varying the number of examples. Concerning defects extraction, a higher *FNR* indicates a higher probability of missing defects, while a higher *FPR* indicates that the manual effort increase. We observe that GPT-3.5 performs best when no examples are provided, as it is challenging to list all the resource-related code examples, and providing a limited number of examples can easily introduce bias into GPT-3.5's results.

Table 8: Defects detection results of PMD, SpotBugs, and Infer.

Tools	# Defects	TP	Accuracy
PMD	1,094	487	44.5%
SpotBugs	254	59	23.2%
Infer	76	9	11.8%

Table 9: Evaluation results of GPT-3.5 defect detection.

Code	# Tests	TP	TN	Uncertain	Accuracy
defects	299	45	-	31	15%
non-defects	299	-	257	30	86%

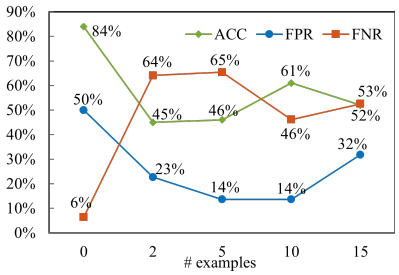


Figure 11: Commits classification results of GPT-3.5 with different numbers of examples in prompt.

Defect Detection. We also evaluate the ability of GPT-3.5 to detect resource leaks, using 299 defect cases and 299 non-defect cases randomly selected from JLeaks. We submitted the code fragments with defects and without defects to the GPT-3.5 respectively, and obtained the results in JSON format. The prompt for this experiment is also publicly available on GitHub, which mainly includes task description and expected output.

The testing results are given in Table 9, in which TP refers to the number of leak-related code fragments correctly identified as such, TN corresponds to the number of non-resource-leak related codes accurately identified as non-resource-leak related code fragments. Unfortunately, GPT-3.5 does not work well and its accuracy is limited to 12.7% for code fragments with defects. However, it achieves high accuracy for code fragments without defects. Interestingly, GPT-3.5 responds with "Based on the provided code fragment, it is not possible to determine with certainty whether there are any resource leak defects. The code does not contain any explicit resource allocation or deallocation related to these resources" for 31 cases. After dedicated inspection, we found that these cases are all about third-party libraries.

6 DISCUSSION

Based on our experience in building JLeaks, we have the following suggestions to share:

First, both manual and automatic approaches are necessary to extract more defects from open-source projects. A large and high-quality resource leak repository is essential for academic research. On the one side, a small-scale defect repository inadequately supports the design of machine learning-based tools. On the other hand, tools and algorithms designed or tested on a low-quality defect repository may not perform well when applied to industrial projects. However, it is challenging to build a fully automated toolchain from end to end. Even if the commit message contains "resource leak", etc., there is still irrelevant, duplicate, and meaningless data. We collect 1,094 resource leaks from open-source projects using a semi-automatic toolchain. Manual classification is time-consuming, and costing us approximately 900 manhours to annotate and verify JLeaks. JLeaks is currently the largest known repository of resource leaks, in which each defect is annotated, and both defect code fragments and patches are included. It is also significant for subsequent research work such as defect pattern analysis, defect detection, defect localization, and automatic defect repair.

Second, technical training on coding style and software security is essential after the kick-off meeting of projects. According to our statistical results, most resource leak defects are not interprocedural. The large number of defects is due to the programmers' lack of awareness in preventing resource leaks. Therefore, before software development, the programmers must undergo special training, which is expected to reduce the occurrence of resource leaks in the code significantly.

Third, automated compilation of open-source projects (including historical versions) is challenging. Even with our best efforts, only 86 project versions were compiled and the 254 bytecode files corresponding to the defects were generated. We leave it as a future work to experiment with more sophisticated approaches within the framework.

Fourth, when designing and implementing defect detection tools, there should be a greater focus on resources related to third-party libraries, specifically targeting intra-procedural resource leak checks. We observed that most resource leaks are intra-procedural defects, indicating that the locations for resource allocation and resource release are in the same method. Furthermore, third-party libraries wrap up system resources and offer interfaces for specific functionalities. However, resource leaks related to these libraries are frequently overlooked during defect detection, leading to a high rate of false negatives.

7 THREATS TO VALIDITY

Some particular resource leaks are not included in JLeaks, such as the defect in test cases. We found about 34% of the 7,483 preliminary filtered leaks in test cases. The primary purpose of the test cases is to validate the application's functionality, ensuring that the application operates as intended. Additionally, test cases, which are coded casually, are not run widely over a long time, which prevents the exposure of potential resource leaks. Therefore, we excluded test cases in JLeaks.

Compiling historical versions of projects poses extensive challenges to validity. We have exerted our utmost efforts to compile the entire open-source project. Regrettably, only 86/871=9.9% project versions can be fully compiled. The causes of compilation failures are usually attributable to missing dependency packages, incorrect Java versions, incomplete projects, and the unique configurations.

Manual analysis of code can be prone to errors and bias affecting the correctness of the JLeaks. However, we have taken great care and made the best effort to ensure the accuracy of the data in JLeaks. First, five experienced Java developers from the industry are employed to annotate the commits. All of them have been using Java for at least eight years. Second, we modeled resource leaks on the theory and trained the participants. Finally, we performed cross-validation during the classification process. We compute the consistency of two-round annotation using Cohen's Kappa metric, as shown in Table 2. The results prove that the two rounds of annotating are consistent.

Memory leaks are not included in JLeaks. Manually building a high-quality and large-scale defect repository is time-consuming. We currently focus solely on resource leaks. In the future, we will include more defects related to memory leaks in JLeaks and continue expanding our repository's scale.

8 RELATED WORK

A data set of wild defects and patches is vital for defect detection, localization, and repair. Two principal approaches for building wild defects repositories are collecting data from competitions or assignments and extracting data from vulnerability database or open source project management platforms. Codeflaws [50], QuixBugs [31], and CodeNet [42] are examples of defect repositories built using the first approach. Although this approach can collect a large number of defects, differences still exist compared with real-world industry defects. Therefore, researchers have adopted the second approach to improve the diversity and practicality of defect repositories. Ponta et al. [41] used NVD as the data source to collect 624 software vulnerabilities from 1,282 commits in 205 projects. Similarly, Deepcva [29] contains 1,229 vulnerabilities from 246 projects involving 542 software vulnerabilities. The data source NVD ensures the precision of defect labeling, yet it simultaneously limits the scale of defect repositories. Besides NVD, the open-source project management platforms, with GitHub being particularly significant, serve as crucial sources for building defect repositories. For instance, Wang et al. [54] collected 7,997 wild-based security patches. Russell et al. [43] compiled millions of function-level examples of C and C++ from SATE IV, Debian Linux distribution, and public Git repositories on GitHub. Tomassi et al. [52] and Madeiral et al. [35] adopted continuous-integration approaches to ensure the continuous growth of defect data sets. However, these repositories use the before-commit code as defect code, which is inaccurate.

Automatic, semi-automatic and manual approaches have been proposed to enhance the accuracy of wild defects collection. Using three static analyzers. VulinOSS [21] picked up 17,738 vulnerabilities from 23,884 versions in 153 projects. GrowingBug, described in [23, 24], utilizes the code refactoring technique and heuristic algorithms to automatically filter out bug-irrelevance changes in GitHub commits, extracting wild defects. Defects4J [26] is a highly accurate data set for defects and patches, primarily due to manual participation. The initial release, Defects4J 1.0, encompasses 357 defects sourced from five real-world projects. Beyond its authenticity and reliability, Defects4J offers the significant advantage of reproducibility. It is worth noting that GrowingBug follows the framework established by Defects4J. In addition, researchers also have built some defect repositories for their works. For example, Sabetta et al. [44] used 540 patches from 220 open-source projects, and Zhou et al. [56] manually collected 48,687 defects.

Some repositories have been built for specific defect types. JaConTeBe [32] manually collected 47 concurrency defects from 8 open-source projects, involving race, deadlocks, and Java-memory-model-related bugs. Similarly, NodeCB [53] extracted 57 concurrency defects from 53 Node.js projects, while GoBench [30] is the first benchmark suite for Go concurrency bugs. Another defect repository, as discussed in [30], focused on data race and was generated by extracting defects from real projects and generating variants automatically, supplemented by manual test cases authoring.

Resource leaks seriously affect the safety and stability of the software system. However, the Java resource leak repositories have not been studied adequately. DroidLeaks [33] offers a comprehensive resource leak repository for Android to address this gap. It contains 292 resource leaks from 32 popular projects and has been built using a combination of keyword-based and manual filtering approaches. Additionally, Bhatt et al. [4] identified 50 resource leaks in the Android system. In the context of the Linux kernel, Slaby et al. [47] uncovered 7 memory leaks and 13 resource leaks, while Pashchenko et al. [38] documented 62 resource leaks in Tomcat. The authors of [19] collected 238 memory leaks and 253 resource leaks from 15 open-source projects on Apache or GitHub. We conclude that the repositories that contain high-quality leaks from real projects are scarce. First, though several existing repositories are available, the number of resource leaks is still too small. Second, a significant amount of code duplication is observed in existing repositories due to the limited number of projects. Third, the availability of some repositories decreases over time. It is challenging to build a high-quality defect repository automatically. However, manually creating a defect repository is indispensable for ensuring quality, providing insight, and addressing shortages.

9 CONCLUSION

We systematically investigate and summarize the resource allocation in Java, using this to assess whether the resources are released correctly and timely in both defect code fragments and their committed patches. After extensive manual analysis, we collected 1,094 resource leak defects from 321 open-source projects on GitHub and built an accurately labeled repository JLeaks. It is currently the largest known repository of resource leaks built from open-source projects (excluding memory leaks), in which each defect is annotated, and both defect code fragments and patches are included. Several interesting conclusions have been drawn after an in-depth analysis of the repository. We find that the pattern of most resource leak problems is relatively simple, and the most effective fix approach is try-with-resources. This repository is also of great significance for subsequent research work such as defect pattern analysis, defect detection, and automatic defect repair. In the future, we will include more defects related to memory leaks in JLeaks and continue to expand the scale of our repository.

ACKNOWLEDGEMENT

The authors would like to thank the anonymous reviewers from ICSE'2024 for their insightful comments and constructive suggestions. This work was partially supported by the National Natural Science Foundation of China (62232003).

REFERENCES

[1] 2008. ISO/IEC 25012:2008 - Systems and software engineering – Software product Quality Requirements and Evaluation (SQuaRE) – Data quality model. International Organization for Standardization. https://www.iso.org/standard/35736.html

[2] Amritanshu Agrawal, Akond Rahman, Rahul Krishna, Alexander Sobran, and Tim Menzies. 2018. We don't need another hero?: the impact of "heroes" on software development. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Frances Paulisch and Jan Bosch (Eds.). ACM, 245–253. https://doi.org/10.1145/3183519.3183549

[3] Miltiadis Allamanis. 2019. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, Onward! 2019, Athens, Greece, October 23-24, 2019*, Hidehiko Masuhara and Tomas Petricek (Eds.). ACM, 143–153. https://doi.org/10.1145/3359591.3359735

[4] Bhargav Nagaraja Bhatt and Carlo A. Furia. 2022. Automated repair of resource leaks in Android applications. *J. Syst. Softw.* 192 (2022), 111417. https://doi.org/10.1016/j.jss.2022.111417

[5] Christian Bird, Peter C. Rigby, Earl T. Barr, David J. Hamilton, Daniel M. Germán, and Premkumar T. Devanbu. 2009. The promises and perils of mining git. In *Proceedings of the 6th International Working Conference on Mining Software Repositories, MSR 2009 (Co-located with ICSE), Vancouver, BC, Canada, May 16-17, 2009, Proceedings*, Michael W. Godfrey and Jim Whitehead (Eds.). IEEE Computer Society, 1–10. https://doi.org/10.1109/MSR.2009.5069475

[6] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, and Prafulla Dhariwal et al. 2020. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (Eds.). https://proceedings.neurips.cc/paper/2020/hash/1457c0d6bfcb4967418bfb8ac142f64a-Abstract.html

[7] Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. 2022. NatGen: generative pre-training by "naturalizing" source code. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (Eds.). ACM, 18–30. https://doi.org/10.1145/3540250.3549162

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, and Greg Brockman et al. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[9] Anton Cheshkov, Pavel Zadorozhny, and Rodion Levichev. 2023. Evaluation of ChatGPT Model for Vulnerability Detection. *CoRR* abs/2304.07232 (2023). https://doi.org/10.48550/arXiv.2304.07232 arXiv:2304.07232

[10] Jacob Cohen. 1960. A Coefficient of Agreement for Nominal Scales. *Educational and Psychological Measurement* 20, 1 (1960), 37–46. https://doi.org/10.1177/001316446002000104 arXiv:https://doi.org/10.1177/001316446002000104

[11] Roland Croft, M. Ali Babar, and Huaming Chen. 2022. Noisy Label Learning for Security Defects. In *2022 IEEE/ACM 19th International Conference on Mining Software Repositories (MSR)*. 435–447. https://doi.org/10.1145/3524842.3528446

[12] Roland Croft, Muhammad Ali Babar, and M. Mehdi Kholoosi. 2023. Data Quality for Software Vulnerability Datasets. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 121–133. https://doi.org/10.1109/ICSE48619.2023.00022

[13] Roland Croft, Yongzheng Xie, and Muhammad Ali Babar. 2023. Data Preparation for Software Vulnerability Prediction: A Systematic Literature Review. *IEEE Trans. Softw. Eng.* 49, 3 (mar 2023), 1044–1063. https://doi.org/10.1109/TSE.2022.3171202

[14] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. 2023. GitHub Copilot AI pair programmer: Asset or Liability? *J. Syst. Softw.* 203 (2023), 111734. https://doi.org/10.1016/j.jss.2023.111734

[15] Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. 2008. The CLOSER: Automating Resource Management in Java. In *Proceedings of the 7th International Symposium on Memory Management* (Tucson, AZ, USA) *(ISMM '08)*. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/1375634.1375636

[16] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empir. Softw. Eng.* 10, 4 (2005), 405–435. https://doi.org/10.1007/s10664-005-3861-2

[17] FaceBook. 2023. GitHub - facebook/infer: A static analyzer for Java, C, C++, and Objective-C. https://github.com/facebook/infer. (Accessed on 03/30/2023).

[18] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Scott Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=hQwb-lbM6EL

[19] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. 2020. Memory and resource leak defects and their repairs in Java projects. *Empir. Softw. Eng.* 25, 1 (2020), 678–718. https://doi.org/10.1007/s10664-019-09731-8

[20] Fabrizio Gilardi, Meysam Alizadeh, and Maël Kubli. 2023. ChatGPT Outperforms Crowd-Workers for Text-Annotation Tasks. *CoRR* abs/2303.15056 (2023). https://doi.org/10.48550/arXiv.2303.15056 arXiv:2303.15056

[21] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: a dataset of security vulnerabilities in open-source systems. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*, Andy Zaidman, Yasutaka Kamei, and Emily Hill (Eds.). ACM, 18–21. https://doi.org/10.1145/3196398.3196454

[22] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of Code Language Models on Automated Program Repair. *CoRR* abs/2302.05020 (2023). https://doi.org/10.48550/arXiv.2302.05020 arXiv:2302.05020

[23] Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu, Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang. 2022. BugBuilder: An Automated Approach to Building Bug Repository. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3177713

[24] Yanjie Jiang, Hui Liu, Nan Niu, Lu Zhang, and Yamin Hu. 2021. Extracting Concise Bug-Fixing Patches from Human-Written Patches in Version Control Systems. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 686–698. https://doi.org/10.1109/ICSE43902.2021.00069

[25] Anshul Jindal, Paul Staab, Pooja Kulkarni, Jorge Cardoso, Michael Gerndt, and Vladimir Podolskiy. 2021. Memory Leak Detection Algorithms in the Cloud-based Infrastructure. *CoRR* abs/2106.08938 (2021). arXiv:2106.08938 https://arxiv.org/abs/2106.08938

[26] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, Corina S. Pasareanu and Darko Marinov (Eds.). ACM, 437–440. https://doi.org/10.1145/2610384.2628055

[27] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. 2014. The promises and perils of mining GitHub. In *11th Working Conference on Mining Software Repositories, MSR 2014, Proceedings, May 31 - June 1, 2014, Hyderabad, India*, Premkumar T. Devanbu, Sung Kim, and Martin Pinzger (Eds.). ACM, 92–101. https://doi.org/10.1145/2597073.2597074

[28] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2022. Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction. *CoRR* abs/2209.11515 (2022). https://doi.org/10.48550/arXiv.2209.11515 arXiv:2209.11515

[29] Triet Huynh Minh Le, David Hin, Roland Croft, and Muhammad Ali Babar. 2021. DeepCVA: Automated Commit-level Vulnerability Assessment with Deep Multi-task Learning. In *36th IEEE/ACM International Conference on Automated Software Engineering, ASE 2021, Melbourne, Australia, November 15-19, 2021*. IEEE, 717–729. https://doi.org/10.1109/ASE51524.2021.9678622

[30] Chunhua Liao, Pei-Hung Lin, Joshua Asplund, Markus Schordan, and Ian Karlin. 2017. DataRaceBench: a benchmark suite for systematic evaluation of data race detection tools. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2017, Denver, CO, USA, November 12 - 17, 2017*, Bernd Mohr and Padma Raghavan (Eds.). ACM, 11:1–11:14. https://doi.org/10.1145/3126908.3126958

[31] Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, SPLASH 2017, Vancouver, BC, Canada, October 23 - 27, 2017*, Gail C. Murphy (Ed.). ACM, 55–56. https://doi.org/10.1145/3135932.3135941

[32] Ziyi Lin, Darko Marinov, Hao Zhong, Yuting Chen, and Jianjun Zhao. 2015. JaConTeBe: A Benchmark Suite of Real-World Java Concurrency Bugs (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, Myra B. Cohen, Lars Grunske, and Michael Whalen (Eds.). IEEE Computer Society, 178–189. https://doi.org/10.1109/ASE.2015.87

[33] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. 2019. DroidLeaks: a comprehensive database of resource leaks in Android apps. *Empir. Softw. Eng.* 24, 6 (2019), 3435–3483. https://doi.org/10.1007/s10664-019-09715-8

[34] Fumio Machida, Jianwen Xiang, Kumiko Tadano, and Yoshiharu Maeno. 2012. Aging-Related Bugs in Cloud Computing Software. In *23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Dallas, TX, USA, November 27-30, 2012*. IEEE Computer Society, 287–292. https://doi.org/10.1109/ISSREW.2012.97

[35] Fernanda Madeiral, Simon Urli, Marcelo de Almeida Maia, and Martin Monperrus. 2019. BEARS: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, Xinyu Wang, David Lo, and Emad Shihab (Eds.). IEEE, 468–478. https://doi.org/10.1109/SANER.2019.8667991

[36] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net. https://openreview.net/pdf?id=iaYcJKpY2B_

[37] Oracle. 2020. FilterInputStream (Java Platform SE 7). https://docs.oracle.com/javase/7/docs/api/java/io/FilterInputStream.html. (Accessed on 03/30/2023).

[38] Ivan Pashchenko, Stanislav Dashevskyi, and Fabio Massacci. 2017. DeltaBench: Differential Benchmark for Static Analysis Security Testing Tools. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2017, Toronto, ON, Canada, November 9-10, 2017*, Ayse Bener, Burak Turhan, and Stefan Biffl (Eds.). IEEE Computer Society, 163–168. https://doi.org/10.1109/ESEM.2017.24

[39] Peter Pickerill, Heiko Joshua Jungen, Miroslaw Ochodek, Michal Mackowiak, and Miroslaw Staron. 2020. PHANTOM: Curating GitHub for engineered software projects using time-series clustering. *Empir. Softw. Eng.* 25, 4 (2020), 2897–2929. https://doi.org/10.1007/s10664-020-09825-8

[40] pmd. 2023. GitHub - pmd/pmd: An extensible multilanguage static code analyzer. https://github.com/pmd/pmd. (Accessed on 03/30/2023).

[41] Serena Elisa 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, MSR 2019, 26-27 May 2019, Montreal, Canada*, Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc (Eds.). IEEE / ACM, 383–387. https://doi.org/10.1109/MSR.2019.00064

[42] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, and Julian Dolby et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, Joaquin Vanschoren and Sai-Kit Yeung (Eds.). https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html

[43] Rebecca L. Russell, Louis Y. Kim, Lei H. Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. In *17th IEEE International Conference on Machine Learning and Applications, ICMLA 2018, Orlando, FL, USA, December 17-20, 2018*, M. Arif Wani, Mehmed M. Kantardzic, Moamar Sayed Mouchaweh, João Gama, and Edwin Lughofer (Eds.). IEEE, 757–762. https://doi.org/10.1109/ICMLA.2018.00120

[44] Antonino Sabetta and Michele Bezzi. 2018. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 579–582. https://doi.org/10.1109/ICSME.2018.00058

[45] Elvis Saravia. 2022. Prompt Engineering Guide. *https://github.com/dair-ai/Prompt-Engineering-Guide* (12 2022).

[46] Ran Shaham, Elliot K. Kolodner, and Shmuel Sagiv. 2000. On the Effectiveness of GC in Java. In *ISMM 2000, International Symposium on Memory Management, Minneapolis, Minnesota, USA, October 15-16, 2000 (in conjunction with OOPSLA 2000), Conference Proceedings*, Craig Chambers and Antony L. Hosking (Eds.). ACM, 12–17. https://doi.org/10.1145/362422.362430

[47] Jiri Slaby, Jan Strejcek, and Marek Trtík. 2013. ClabureDB: Classified Bug-Reports Database. In *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7737)*, Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni (Eds.). Springer, 268–274. https://doi.org/10.1007/978-3-642-35873-9_17

[48] SpotBugs. 2023. SpotBugs. https://spotbugs.github.io/. (Accessed on 03/30/2023).

[49] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*. IEEE Computer Society, 48–62. https://doi.org/10.1109/SP.2013.13

[50] Shin Hwei Tan, Jooyong Yi, Yulis, Sergey Mechtaev, and Abhik Roychoudhury. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 180–182. https://doi.org/10.1109/ICSE-C.2017.76

[51] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. 2023. Is ChatGPT the Ultimate Programming Assistant - How far is it? *CoRR* abs/2304.11938 (2023). https://doi.org/10.48550/arXiv.2304.11938 arXiv:2304.11938

[52] David A. Tomassi, Naji Dmeiri, Yichen Wang, Antara Bhowmick, Yen-Chuan Liu, Premkumar T. Devanbu, Bogdan Vasilescu, and Cindy Rubio-González. 2019. BugSwarm: mining and continuously growing a dataset of reproducible failures and fixes. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 339–349. https://doi.org/10.1109/ICSE.2019.00048

[53] Jie Wang, Wensheng Dou, Yu Gao, Chushu Gao, Feng Qin, Kang Yin, and Jun Wei. 2017. A comprehensive study on real world concurrency bugs in Node.js. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, Grigore Rosu, Massimiliano Di Penta, and Tien N. Nguyen (Eds.). IEEE Computer Society, 520–531. https://doi.org/10.1109/ASE.2017.8115663

[54] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. 2021. PatchDB: A Large-Scale Security Patch Dataset. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2021, Taipei, Taiwan, June 21-24, 2021*. IEEE, 149–160. https://doi.org/10.1109/DSN48987.2021.00030

[55] Jialu Zhang, José Cambronero, Sumit Gulwani, Vu Le, Ruzica Piskac, Gustavo Soares, and Gust Verbruggen. 2022. Repairing Bugs in Python Assignments Using Large Language Models. *CoRR* abs/2209.14876 (2022). https://doi.org/10.48550/arXiv.2209.14876 arXiv:2209.14876

[56] Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 10197–10207. https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html