FixrLeak: GenAI-based Resource Leak Fix for Real-World Java Programs

Zhizhou Zhang
Uber Techonologies Inc.
Sunnyvale, CA
zzzhang@uber.com

Akshay Utture
Uber Technologies Inc.
Sunnyvale, CA
akshay.utture@uber.com

Manu Sridharan
Univeristy of California, Riverside
Riverside, CA
manu@cs.ucr.edu

Jens Palsberg
University of California, Los Angeles
palsberg@ucla.edu
Los Angeles, CA

Abstract

Resource leaks frequently occur in Java applications, which can cause severe performance issues and system malfunctions. To address this, we introduce `FixrLeak`, a Generative AI-based framework that automatically generates fixes for resource leaks in real-world Java programs. `FixrLeak` integrates AST-level analysis to generate correct and idiomatic fixes. The framework automates the entire process and significantly reduces the manual effort required by developers. Our evaluation within Uber's Java codebase demonstrates the effectiveness of `FixrLeak`, achieving a high success rate in fixing resource leaks.

1 Introduction

Resource leaks are a persistent challenge in Java applications. When resources such as files, database connections, or streams are not properly closed, they can lead to resource leaks—elusive bugs that can cause security vulnerabilities [2], performance degradation, outages, and severe failures [7] as they accumulate over time.

At Uber, ensuring high code quality is a priority, and as part of this commitment, we have implemented and deployed advanced tools and processes to tackle resource leaks head-on. Our resource leak detection system, powered by SonarQube,[1] has been deployed for several years, scanning millions of lines of Java code in our internal monorepo [11]. This system has identified many outstanding leaks, some of which had persisted for years.

Recognizing the limitations of relying solely on developer intervention [5], we have developed `FixrLeak`, a framework based on Generative AI (GenAI) designed to automatically generate fixes for resource leaks. `FixrLeak` not only automates the generation of code patches, but also performs sanity checks to ensure the correctness of these fixes before submitting them for review. Our goal is to make Uber's Java codebase as close to being free of resource leaks as possible.

Designing `FixrLeak` presents several challenges. First, the generated patches must effectively resolve the resource leaks. Second, the fixes must not introduce new issues, such as using resources after they have been closed. Finally, the fixes must adhere to Java coding best practices, specifically using the `try`-with-resources statement.

The contributions of this work are as follows:

[1] https://www.sonarsource.com/products/sonarqube/

Machine Learning for Systems Workshop at (NeurIPS 2024).

1. A closed-loop framework that integrates GenAI with program analysis to effectively address resource leaks in Uber's internal codebase.
2. An empirical evaluation demonstrating the high success rate of the framework in an industrial environment.
3. A case study on several interesting GenAI cases suggested.

2 Background

2.1 What is a Resource Leak

When programmers deal with resources in Java, they need to properly close the resources after use to prevent resource leaks. For example, in the original code on the top side of Fig 1, there exists a leak on the `reader` object. The `reader` object is declared on line 3 and holds a file descriptor, which is a limited system resource. Without timely release of such resources, over time, the resource may become exhausted. If file descriptors are exhausted, for example, it may cause applications to fail or become unable to open new files.

There are multiple ways to fix a resource leak, such as using `try/catch/finally` blocks. However, using `try`-with-resources statement has become the recommended way of managing resources [4]. `try`-with-resources statement reduces the amount of boilerplate code introduced by `try/catch/finally`, and safely releases resources even when exceptions occur.

```
1  public void readFile(String filePath) {
2    BufferedReader reader = new BufferedReader(new FileReader(filePath));
3    try {
4      String line;
5      while ((line = reader.readLine()) != null) {
6        System.out.println(line);
7      }
8    } catch (IOException e) {
9      e.printStackTrace();
10   }
11 }
```

```
1  public void readFile(String filePath) {
2    try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
3      String line;
4      while ((line = reader.readLine()) != null) {
5        System.out.println(line);
6      }
7    } catch (IOException e) {
8      e.printStackTrace();
9    }
10 }
```

Figure 1: Example code with resource leak (top) and fixed using `try`-with-resources (bottom).

2.2 Prior Work on Resource Leak Fix

There has been some recent work on automatically fixing resource leaks [12, 13, 8, 9], and we discuss it in two parts: non-GenAI based and GenAI based.

The state-of-the-art non-GenAI tool is RLFixer [12]. It uses a resource-escape analysis written in WALA to find fixable leaks, and applies hand-designed fix-templates to generate fixes. Compared to these pre-GenAI tools, GenAI-based approaches have two main advantages from an industrial deployment point of view at Uber. Firstly, they do not need manually hand-crafted templates; this allows the tool to work with newer programming idioms (such as the recommended try-with idiom), different kinds of bugs, and potentially different languages, without requiring a new labor-intensive template design. Secondly, previous tools require analysis frameworks such as WALA, which are tricky to integrate into Uber's Bazel build system [1] and do not scale to Uber's massive codebase size. GenAI-based fixing, on the other hand, requires no build-tool integration since it works on pure text input, and can be fed only relevant code out of the entire codebase, thereby circumventing the scalability issue.

InferFix [8], a proprietary Microsoft tool, is the current-best GenAI-based resource-leak fixing tool, and it targets Infer resource leaks by building an LLM-fixing automation pipeline similar to

FixrLeak. However, it has two major downsides. Firstly, InferFix gets only a 70% fix-correctness; the percentage of developer-accepted fixes will be even lower. This problem is akin to static analysis tools having a high false-positive rate; developers will start distrusting the tool and ignoring it [6]. The root cause for the modest fix-correctness is that InferFix attempts every single leak, including both, hard-to-fix and easy-to-fix leaks. LLMs fail to correctly fix most hard-to-fix leaks because they need object lifetime prediction and inter-procedural data-flow analysis. Hence, what we need is to improve the fix-correctness rate by avoiding the hard-to-fix leaks. The second downside to InferFix is that it relies on a pre-trained version of the Codex model at Microsoft for higher accuracy; custom models are effort-intensive and cannot benefit from ongoing improvements to third-party models like the GPT models. In this paper, we demonstrate how high accuracy can be achieved with a third-party black-box model.

3 Design

The general flow of FixrLeak is plotted in Fig. 3. Currently, we are focusing on fixing leaks where the lifetime of the resource does not exceed the function that allocated the resource.

Input Gathering. FixrLeak will first read the target resource leaks from SonarQube. Specifically, it will get a list of all detected resource leaks, including file name and line number. Since the source code can be changed and the line number of the leak can be different, we compute a deterministic hash using file name and function name to check if a prior fix has been generated or not. Then FixrLeak will use the AST parsing library `tree-sitter` to extract the original source code of the **leaking function**. We use SonarQube instead of other leak-detection tools like Infer because it aggressively prunes potential false-positive leaks; this pruning leaves mostly intra-function leaks which can be correctly fixed by LLMs with high accuracy.

AST-level Analysis Simply performing the resource leak fix without additional analysis can be dangerous. For example, Fig. 2 shows an example where `reader` cannot be closed within `openFile` and a proper fix should occur at the caller site of `openFile`. Otherwise, it can cause `use after close` error. Currently, we use tree-sitter [3] to parse the Java source code and perform the check at the Abstract Syntax Tree (AST) level. As a rough approximation, FixrLeak will not fix a function in which the resource is passed in as a parameter or returned at the end of the function, written to a field and copied to another local variable. For those cases, the resource is very likely to outlive the scope of the leaking function, and a more complex analysis is needed on the caller side.

```
1 public BufferedReader openFile(String filePath) throws IOException {
2   // The BufferedReader is created and returned, so it cannot be closed here.
3   BufferedReader reader = new BufferedReader(new FileReader(filePath));
4   return reader;
5 }
```

Figure 2: An example that cannot be fixed using `try-with-resources` within the function.

Prompt Engineering. After the check, FixrLeak creates a prompt to query the LLM for a fix. Details of the prompt can be found in the Appendix. Once FixrLeak sends the prompt to GenAI such as ChatGPT [10], it will wait for a response. Then it will replace the original leaky function with the GenAI-suggested leak-free function, and generate a pull request.

PR Verifying. To ensure that the generated pull request (PR) is valid, FixrLeak will perform several rounds of tests before sending the PR to reviewers. For example, it will check if the target binary can be built after the code refactoring, it will check if all existing tests pass, and it will also trigger a new run of SonarQube at that specific target to make sure that the detected leak is eliminated.

Code Review. The final step is the code review from the developers to accept the PR. For most cases, all they need to do is one-click accept.

4 Evaluation

In this experiment, we test FixrLeak with ChatGPT-4O [10]. We chose the first 124 bugs detected by SonarQube for our evaluation. 12 of these are in deprecated code, and we discard them. Of the

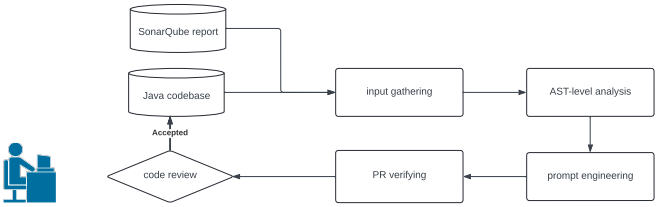

Figure 3: Resource leak fixing workflow using ChatGPT.

remaining 112 bugs, 102 passed the AST-level analysis check, which ensures that the lifetime of the resource does not exceed the leaky function. As a result, 93 out of 102 leaks can be fixed using GenAI automatically. The rest of the leaks have different bazel configurations for validation and testing that require a certain manual involvement. Among these 93 generated PRs, 63 have already been accepted by developers and the rest is pending approval. In addition, 61 of the leaks can be fixed during the first attempt. In general, we can see that the success rate of GenAI-based resource leak repair is very high. The high success rate stems from focusing on intra-function leaks; the intra-function level fixing task fits the strength of LLMs.

4.1 Other Interesting Findings

We also noticed a couple of interesting cases in which GenAI performs adjustments in addition to pure code refactoring on `try`-with-resources to get the correct fix. In both cases, we did not give an extra command in the prompt.

Switching to Proper Class. We found that LLM can be creative when the type of variable holding the leaked object is not a subtype of `AutoCloseable`. A simplified version of the code is shown in Fig. 4 (Appendix). The original resource `response` on line 3 is assigned to a variable of type `org.apache.http.HttpResponse`, which does not implement the `AutoCloseable` interface. As a result, it cannot be simply fixed by the `try`-with-resources block. Interestingly, we found that GenAI changed the type of `response` from `HttpResponse` into `CloseableHttpResponse`, where `CloseableHttpResponse` implements `AutoCloseable` and hence enables a fix via the `try`-with-resources block. The suggested fix itself is not compileable since `CloseableHttpResponse` class is not imported into the source file. Otherwise, the fix is correct. Therefore, we also ask LLM to return the new potential imports for the suggested fix and automatically add the missing imports in `FixrLeak`.

Change Exception Type. Another interesting case that we observed is that GenAI can also change the type of exception in the enclosing method signature through the fix. The code in Fig. 5 (Appendix) shows the simplified example. In the original code, when the FileInputStream constructor cannot find the file, it will throw `FileNotFoundException` exception at line 2. However, when the refactored code has `try`-with-resources, the Java compiler will call the `close()` method under the hood and expect the method to handle or declare all exceptions with `IOException` at line 3. `FileNotFoundException` is a subclass of `IOException`. Changing the throws clause to IOException still covers the possibility of a `FileNotFoundException` being thrown, but it also accounts for other IO-related issues that might arise within the `try`-with-resources block.

5 Conclusion and Future Work

Resource leaks have been extensively explored in previous studies. However, there has been a notable gap in the literature between detection and fixing. There are several practical roadblocks that prevent the application of the latest fixing techniques to industry code bases, such as complex build systems [1]. Generative AI provides a unique advantage in bridging the gap and offering correct and idiomatic fixes. With the help of GenAI and AST-level analysis, `FixrLeak` can repair resource leaks within Uber's Java codebase effectively and efficiently. In the future, we would like to improve: a) support for inter-procedural fixes, b) the incorporation of GenAI-based leak detection, and c) more source code analysis to accurately identify user-defined resource classes.

References

[1] Bazel. https://bazel.build/. (Accessed on 08/14/2024).

[2] Cwe - cwe-400: Uncontrolled resource consumption (4.15). https://cwe.mitre.org/data/definitions/400.html. (Accessed on 08/13/2024).

[3] Tree-sitter: Introduction. https://tree-sitter.github.io/tree-sitter/. (Accessed on 08/14/2024).

[4] The try-with-resources statement (the java™ tutorials > essential java classes > exceptions). https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html. (Accessed on 08/13/2024).

[5] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*, pages 332–343, 2016.

[6] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 332–343, New York, NY, USA, 2016. Association for Computing Machinery.

[7] Mohammadreza Ghanavati, Diego Costa, Janos Seboek, David Lo, and Artur Andrzejak. Memory and resource leak defects and their repairs in java projects. *Empirical Software Engineering*, 25:678–718, 2020.

[8] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1646–1656, 2023.

[9] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. Skipanalyzer: An embodied agent for code analysis with large language models. *arXiv preprint arXiv:2310.18532*, 2023.

[10] OpenAI. Gpt-4 technical report, 2023.

[11] Rachel Potvin and Josh Levenberg. Why google stores billions of lines of code in a single repository. *Communications of the ACM*, 59(7):78–87, 2016.

[12] Akshay Utture and Jens Palsberg. From leaks to fixes: Automated repairs for resource leak warnings. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 159–171, 2023.

[13] Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, page 151–162, New York, NY, USA, 2018. Association for Computing Machinery.

Appendix

A Context-setting message to OpenAI (role: "system")

In this section, we provide the prompts used to interact with OpenAI.

The try-with-resources statement is a try statement that declares one or more resources. A resource is an object that must be closed after the program is finished with it. The try-with-resources statement ensures that each resource is closed at the end of the statement. Any object that implements java.lang.AutoCloseable, which includes all objects that implement java.io.Closeable, can be used as a resource. The following example reads the first line of a file. It uses an instance of FileReader and BufferedReader to read data from the file. FileReader and BufferedReader are resources that must be closed after the program is finished with it:

```
static String readFirstLineFromFile(String path) throws IOException {
      try (FileReader fr = new FileReader(path);
          BufferedReader br = new BufferedReader(fr)) {
          return br.readLine();
      }
}
```

In this example, the resources declared in the try-with-resources statement are a FileReader and a BufferedReader. The declaration statements of these resources appear within parentheses immediately after the try keyword. The classes FileReader and BufferedReader, in Java SE 7 and later, implement the interface java.lang.AutoCloseable. Because the FileReader and BufferedReader instances are declared in a try-with-resource statement, they will be closed regardless of whether the try statement completes normally or abruptly.

The code below contains a resource leak, please fix it by closing the resource properly. Please only return the fixed code. Do not remove comments or change the code structure. Show the complete code; do not skip or omit any lines. Don't add redundant resource closing and please keep the original code structure (i.e. resource declaration in try-with-resources) in place if possible. Don't change any formatting and indentation. Keep using the original number space or tab. If the resource is non-local, meaning that it is used as a parameter in a method call, you can ignore it and return the original code.

Please only return the compilabe code, don't include "'java etc, don't add any explanation. Don't remove unnecessary qualifier. If the resource outlive the method, you could ignore it and return the original code. Specifically, if the resource is passed in as function parameter, or the resource is used in return, or the resource is assigned to a class fields, just return the original code. Otherwise, please fix the resource leak.

Output format: Put the fixed code between <STARTCODE> and <ENDCODE>, put the import statements needed by the fixed code between <STARTIMPORT> and <ENDIMPORT>, each import is in a new line; do not include the "import " prefix.

Only add the imports from the standard library. Don't miss any potential imports you used in the fixed code, including the function call, variable declaration, template type etc. Please don't put import statements between <STARTCODE> and <ENDCODE>. Return the imports in the <STARTIMPORT> and <ENDIMPORT> section.

B More Code Examples

In this section, we show more examples of the code snippet of the original leak and suggested fixes from LLM.

B.1 Example on Switiching Proper Class

```
public Map postQueryResponse(String query, String uri) throws URISyntaxException,
     IOException {
  HttpPost request = new HttpPost(uri);
- try {
-   HttpResponse response = httpClient.execute(request);
+ try (CloseableHttpResponse response = httpClient.execute(request)) {
    ...
  } catch (Exception e) {
    return Collections.emptyMap();
  }
}
```

Figure 4: Code snippet where GenAI changed imported class from `Closable` into `AutoCloseable`.

B.2 Change of Exception Type Example

```
 public static PublicKey fromPath(String publicKeyFilePath, String encryptionInstance)
-    throws FileNotFoundException, CertificateException {
+    throws IOException, CertificateException {
-  FileInputStream fin = new FileInputStream(publicKeyFilePath);
-  CertificateFactory f = CertificateFactory.getInstance(encryptionInstance);
-  X509Certificate certificate = (X509Certificate) f.generateCertificate(fin);
+  try (FileInputStream fin = new FileInputStream(publicKeyFilePath)) {
+    CertificateFactory f = CertificateFactory.getInstance(encryptionInstance);
+    X509Certificate certificate = (X509Certificate) f.generateCertificate(fin);
    return certificate.getPublicKey();
  }
```

Figure 5: Simplified code showing changes of exception type during refactoring.