A BROADER STUDY OF THE EFFECTIVENESS OF RESOURCE LEAK CHECKING AND REPAIR FOR JAVA

By

Bryce Hojo

A capstone project submitted for Graduation with University Honors

February 12, 2026

University Honors

University of California, Riverside

APPROVED

Dr. Manu Sridharan

Department of Computer Science and Engineering

Dr. Begoña Echeverria, Howard H Hays, Jr. Chair

University Honors

ABSTRACT

Resource leaks occur when a program utilizes resources (e.g., files, database connections, network sockets), but fails to release them after use. This can result in performance slowdowns, system crashes, and denial-of-service attacks leaving applications vulnerable to security risks. The Checker Framework Resource Leak Checker (RLC) is a static analysis tool that soundly detects potential resource leaks in Java programs but does not automatically repair them. Recent work, Arodnap, combines RLC, RLC's annotation inference component, and RLFixer's repair framework to improve automation of resource leak repair. Arodnap uses NJR-1 Dataset, a normalized benchmark consisting of 293 Java programs, to run its full pipeline but does not test on real-world projects. In this work, we evaluate the effectiveness of resource leak detection and repair using Arodnap on 9 open-source Java applications spanning diverse domains, including game engines, bioinformatics, networking, and embedded database systems. Our evaluation highlights cases where automated repair succeeds through the use of try-with-resources statements, automatically closing the resource after its use, as well as limitations in patch generation and applicability to applying the full pipeline in a real-world setting.

ACKNOWLEDGEMENTS

I would like to extend my gratitude to my faculty mentor, Dr. Manu Sridharan, for his invaluable guidance, support, and expertise throughout the project. I am thankful for the opportunity to work in his research group and to dive deeply into Java static analysis tools for resource leak detection. I would also like to thank my graduate student mentor, Sanjay Malakar, for his advice and assistance. I spend many long nights debugging errors in my code, and I greatly appreciate his unwavering support and knowledge throughout my research. Finally, I would like to thank my family and friends for their constant encouragement and support. I am grateful to have had the opportunity to conduct this research with both my faculty mentor and my graduate student mentor.

TABLE OF CONTENTS

INTRODUCTION

A resource leak occurs when a program uses a resource (ex. files, database connections, network sockets) but fails to release it after it is no longer needed. This can lead to slowdowns in the program and pose security vulnerabilities. Java's built-in type system in the programming language is insufficient to detect and prevent many classes of programming errors, motivating the development of the Checker Framework (CF). CF extends Java's type system with custom, pluggable type-checking annotations that integrate directly into the language and the compiler [1]. These pluggable type-checking systems enable the detection of specific bugs at compile time. We used Checkerframework 3.49.0 for our study. General Java analysis tools such as SpotBugs [2] use bytecode pattern matching to detect general bugs in the code, however it does not model resource ownership, which can lead to false positives.

The Resource Leak Checker (RLC) is a pluggable type system built upon CF. RLC performs static analysis to identify resource leaks in Java programs, allowing these errors to be detected early without executing the program. RLC relies on annotations to keep track of the flow state of resources from initialization to deallocation. For instance, the "@Owning" annotation tracks ownership of the resource and ensures the resource is properly closed with the owning reference [3]. These annotations give more information to the checker to detect possible resource leaks. Figure 1 illustrates a socket resource, skt, that must be properly closed. However, the socket is reassigned to another resource. As a result, the checker reports an error that the owning resource skt is declared final and can not be reassigned. As a developer, it would be cumbersome to include all the annotations manually. As a result, a tool called Inference works with the RLC to automatically infer all of the annotations [4]. Rather than writing annotations, a

developer only focuses on solving the resource leaks given warnings/errors provided by the RLC.

```
1 +@Owning
2 private final Socket skt = new Socket();
3 skt = new Socket();
4 skt.close();
```

Fig. 1: RLC reports this as an error due to reassignment of resource skt.

To provide broader context on resource leak detection in Java, several resource leak detection tools exist beyond the RLC. Recent resource leak detection tools aim to improve the identification of unreleased resources using a variety of techniques. One approach leverages accelerating technologies, Artificial Intelligence (AI), a method that has been implemented to improve static resource leak detection in Java programs. A tool called InferROI uses a Large Language Model, a type of AI, achieving a 59% promising bug detection rate and a false alarm rate of 18.6% [5]. While this tool does not automate fixes of resource leaks, it provides promising results in the detection of leaks through AI. Another tool, MrROK, introduces a novel approach to mine abstract patterns of resource usage called Abstract Resource Acquisition and Release pairs using rule-based expansion to detect more resource leaks [6]. Instead of relying on known API pairs, a set of two API methods, one to obtain a resource and the other to release the same resource, MrROK can identify more cases of resource leaks allocated but not released. Collectively, these tools illustrate the diversity of techniques being explored to improve resource leak detection in Java applications.

Another tool, RLFixer, generates repairs for resource leaks given any resource-leak detector. It can provide low overhead and fix 66% of the warnings, of which 95% are correct [7]. RLFixer uses a data-flow analysis of resource objects to separate feasible leaks from infeasible leaks and classifies resource leaks, helping to improve repairability. RLFixer identifies repair

templates, such as a try-finally block, which guarantees a specific line of code will execute. This technique is used to apply a close method to the resource to deallocate it. Despite this approach, RLFixer is limited to only fixing leaks of Java library resources and can not repair leaks in resource wrappers, a programmer-written class that acts as the resource and must be closed.

Arodnap, a Java resource leak repair tool, addresses RLFixer's limitations by utilizing RLC, RLC Inference (RLCI), and RLFixer. In addition, Arodnap extends upon RLFixer by adding a code transformation stage to enable precise reasoning of resource leaks, a novel field containment analysis (tracks if a resource escapes to a field of an object), and new repair patterns to determine if a new repair can be applied without introducing new resource leaks [8]. Both RLFixer and Arodnap were evaluated using the NJR-1 dataset [9], a normalized benchmark of projects that are compiled and run successfully on version Java-8. This study evaluates the effectiveness of resource leak detection and repair on real-world Java applications, using the full pipeline of Arodnap (RLC, RLCI, RLFixer).

METHODS

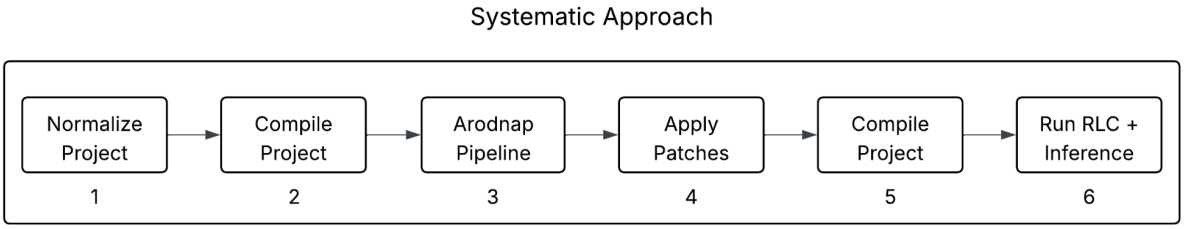

Fig. 2: Full process of study

During the initial phase of study, we selected a list of 9 real-world Java applications ranging from various domains to best suit running Arodnap on. We used Chat-GPT to curate a large list of open-source Java applications that utilize resources such as databases, streams, and sockets. From the list, we narrowed down our search to select projects that are compatible with Java 11 and are active with contributions from developers. In addition, we only selected projects

that use Maven or Gradle, common Java build systems in modern applications, due to ease of dependency resolution and having a standard project layout. As a requirement to run Arodnap, each project must be normalized. For a project to be normalized, it must have all dependencies collected, a standard Java source structure, relative paths to Java source files, and fully qualified Java class names to analyze [10]. To achieve this, we developed and executed a project-specific build script that constructs an independent project conforming to Arodnap's required project structure. We removed all test folders and files from each project to ensure only production files are considered for resource leaks. Next, we ran the Javac command, the Java compiler, on the normalized project, checking for correctness and ensuring it compiled successfully with no errors. This step confirms the success of normalization for each project in step 1. During the normalization process, common Java errors arose from missing library dependencies due to variations in project build scripts which were handled accordingly.

In steps 3 and 4, we ran the full Arodnap pipeline on each project and applied the generated patches from Arodnap on the normalized project. When running Arodnap, the tool has multiple stages consisting of running RLCI + RLC, performing code transformations, re-running RLCI + RLC, running an enhanced RLFixer, and lastly performing patch validation to ensure the patches do not create any new compilation errors. Our baseline of resource leak warnings is recorded after the re-run of RLCI + RLC. After Arodnap performs code transformations and re-runs RLC with inference, shifted resource leaks may arise as a result of the origin of the leak reporting to a different line of code due to code transformation [8]. To prevent inaccurate resource leak errors, this study focuses on the analysis of RLFixer enhancements and patch generation of Arodnap. After Arodnap is completed, it generates a folder containing the project name along with the generated patches outlining the text-based representation of the changes

with the original code and the suggested additions/deletions of Arodnap. A script is used to apply the patches to the code base of the normalized project. If there are multiple patches targeting different resource leaks on the same lines of code, only the first patch will be applied, while the others are not. Manual intervention is needed to combine patches if they affect similar lines of code.

In step 5, after the patches have been applied, we run the Javac command again to ensure that the applied patches do not introduce new compilation errors. In the final step, we run RLC with inference to determine if the patches fix the resource leaks. Overall, this systematic approach ensures the reproducibility and robustness of our findings.

RESULTS

We ran Arodnap on 9 benchmarks consisting of JGit [11], H2 [12], SootUp [13], jMonkeyEngine [14], BioJava [15], Netty-SocketIO [16], Jackson Databind [17], Checkstyle [18], and JaCoCo [19]. Arodnap runs an enhanced RLFixer to produce concrete patches. Figure 3 outlines the baseline total of resource leaks after applying initial code transformations. The next category, fixable, represents the number of resource leaks that RLFixer claims it can fix based on improved warnings and specifications from RLC and RLCI. Patches generated represent the number of patches created from Arodnap. Figure 3 does not show whether the target was resolved. We treat each generated patch as a candidate fix. We later validate each patch by checking whether it eliminates the corresponding RLC warning.

Project	Version	# RLC Warnings	Fixable	Patches Generated
JGit	6.5.0	294	N/A	N/A
H2	2.4.240	512	23	17
SootUp	1.3.1	21	21	19
jMonkeyEngine	3.8.1-stable	43	14	10
BioJava	7.2.4	162	87	81
Netty-SocketIO	2.0.13	8	8	4
Jackson Databind	2.20.1	321	20	7
Checkstyle	10.26.1	16	3	1
JaCoCo	0.8.10	133	15	0

Fig 3: Graph highlighting number of RLC warnings, patches deemed fixable by Arodnap, and patches successfully generated across 9 Java benchmarks.

DISCUSSION

In Figure 3, H2 has over 512 resource leaks. Out of these leaks, RLFixer analyzed 23 that can be fixed, and only 17 patches were generated. Out of the 489 resource leaks from H2 that are not fixable by RLFixer, 238 resource leaks are unfixable due to field escapes, 238 leaks could not be classified, and 13 leaks escaped via an array. H2 had leaks that could not be classified due to RLCI timing out after 1 hour. Arodnap is designed to be usable as a developer, thus attempting to provide acceptable runtimes of the entire pipeline. As a result, if Arodnap's analysis on the benchmark takes longer than an hour to run, Arodnap will stop its current stage and continue to the next stage. Since inference was not fully finished, RLFixer had difficulty trying to determine if the leaks were fixable or not, and hence could not classify the resource leaks.

A field escape is when a resource gets stored in a field of an object. The lifetime of the resource extends beyond the current method, and the resource has no control over when it should be closed. Although Arodnap provides improved field containment analysis to determine resource wrappers making them safe to deallocate, field escapes are not safe to repair.

When patching the leaks, Arodnap used a template-guided Abstract Syntax Tree, such as wrapping a block in try-with-resources statement or a try-finally block [8]. A try-with-resources statement does automatic invocation of the close method on resources, eliminating the need to use a finally block solely to deallocate the resource. However, the class that owns the resource must implement AutoCloseable to encapsulate the resource, allowing it to be explicitly released via the “close()” command. Arodnap analyzes the classes that contain a resource and implements AutoCloseable to be able to use the try-with-resources for wrapped resources. Arodnap prioritizes solving all leaks with the try-with-resources template, but if it is not valid, it would resort to using a try-finally block. Of the 17 patches generated for H2, 13 patches used try-with-resources statements and 4 patches used a try-finally block. After running Arodnap on all benchmarks and applying all patches to each benchmark, all corresponding patches resolved the targeted resource leaks.

JGit and JaCoCo were the two benchmarks that failed to produce any patches. When running Arodnap on JGit, the RLFixer stage failed due to an array out of bounds exception. This failure is caused by an internal WALA call graph construction error. During propagation-based call graph building, WALA encountered a method invocation that attempted to compute possible call targets. The resulting target set was empty, which led to the array out of bounds error. This result is likely due to JGit’s extensive use of Java 8 features such as lambdas, an expression used to write concise functional-style code. WALA’s lambda support is partial, resulting in unresolved call targets during static analysis [20]. Due to this internal failure, RLFixer is unable to generate any fixable leaks. In JacCoCo, RLFixer attempted to produce patches. Out of 133 leaks reported from RLC, RLFixer analyzed 15 leaks with potential fixes. All 15 leaks were field escapes resulting in Arodnap producing zero patches.

Arodnap successfully fixed nearly half of the resource leaks in BioJava. The majority of the leaks detected involved the InputStream and FileInputStream resources. These resources are in the set of Java streams recognizable in static analysis tools. Arodnap is able to quickly identify these resources and fix them, provided the resources do not escape into a field. However, in Jackson Databind, some resource leaks involve more complex lifetimes posing challenges for automated fix generation.

In past work, I ran RLC on SootUp, and from the leaks reported, I attempted to manually fix them and make contributions to the project. I used the same version of SootUp from our prior work for running Arodnap on SootUp and analyzed the difference between manual fixes and patches generated by Arodnap. From the leaks reported, I fixed 16 of those leaks using try-with-resources statements and try-finally blocks. Figures 4(a) and 4(b) show a FileInputStream resource being closed with a try-with-resource statement, both manually and in Arodnap's patch. Note in Figure 4(a), the BufferedReader and InputStreamReader are also declared in the try-with-resource statement. However, this is not strictly necessary since closing the underlying FileInputStream is sufficient, and the BufferedReader does not own the underlying stream. In Figure 4(b), Arodnap instead declares only the FileInputStream as a resource. It introduces a temporary variable called "_arodnap_temp0" and replaces the subsequent uses of the stream with this variable. Although the two patches differ in structure, both eliminate the resource leak in TamiflexModel.java. Arodnap replaces all instances of the resource with the temporary variable. Both of these try-with-resource statements fixed the resource leak in TamiflexModel.java. In addition to Arodnap successfully generating patches, Arodnap enforces a compilation check on every suggested patch and will not produce the patch if it fails to compile. In SootUp, one of the resource leaks identified a case where the resource,

Stream, escaped inside a lambda. Arodnap was able to detect this case and while it attempted to produce a patch for it, the compilation check failed, and the patch was not produced. In addition to running RLC on SootUp, prior work involved attempts to run just RLFixer on SootUp, which was not the main focus of the research. RLFixer only provides repair hints that are purely textual. This does not simplify the process for developers to apply the hints in their codebase. Arodnap addresses this by creating a concrete patch that visually shows the changes and provides a script to apply them directly to the project.

```
1  private void parseTamiflexLog(String logFile, boolean verbose) {
2  -   try {
3  -     BufferedReader reader =
4  -         new BufferedReader(new InputStreamReader(new
       FileInputStream(logFile)));
5  +   try(FileInputStream fis= new FileInputStream(logFile);
6  +       InputStreamReader isr = new InputStreamReader(fis);
7  +       BufferedReader reader = new BufferedReader(isr); ) {
8       String line;
9       while ((line = reader.readLine()) != null) {
10        String[] portions = line.split(";", -1);
```

Fig. 4(a): Manual Resource Leak Fix. Adapted from benchmark SootUp, file TamiflexModel.java.

```
1     private void parseTamiflexLog(String logFile, boolean verbose)
          {
2  -    try {
3  +    try (java.io.FileInputStream __arodnap_temp0 = new
       FileInputStream(logFile)) {
4        BufferedReader reader =
5  -         new BufferedReader(new InputStreamReader(new
       FileInputStream(logFile)));
6  +         new BufferedReader(new InputStreamReader(
       __arodnap_temp0));
7        String line;
8        while ((line = reader.readLine()) != null) {
9          String[] portions = line.split(";", -1);
```

Fig. 4(b): Arodnap's generated patch. Adapted from benchmark SootUp, file TamiflexModel.java.

Mentioned in the methodology, manual intervention was required to apply some of the Arodnap patches to the benchmarks. Arodnap treats each patch independent of each other and reported resource leaks that are intertwined with each other fail to get applied. The first of the intertwined leaks would get patched and the other would fail. In some cases where leaks appear in the same file, Arodnap attempts to fix the patch through a try-with-resources statement, initializing "_arodnap_temp0" as a temporary variable. However, when Arodnap attempts to fix another leak in the same file using a temporary variable, it will use the same variable name "_arodnap_temp0." When compiling the entire project with the added patches, Javac will flag an error due to " _arodnap_temp0" being initialized twice. Arodnap is unable to prevent duplicate variables due to fixing each leak independent of each other. The entire benchmark would need to be re-analyzed in order for Arodnap to recognize this error, resulting in potential slower runtime of the entire pipeline. This approach is inefficient, and Arodnap expects the developer to analyze the patches and fix compilation errors.

The results from Figure 3 highlight a contrast between fixable leaks versus leaks that are fixed through Arodnap's patches. Fixable leaks should provide a corresponding patch, but not all benchmarks have patches for every fixable leak. RLFixer provides a debug file for each benchmark explaining its analysis of the resource leaks detected. For example, RLFixer's debug file for jMonkeyEngine reports one of the resource leaks as fixable and attempts to solve it with a return fix, but it fails. A return fix is when the resource is deallocated right before the method is returned. Figure 5 presents a resource, InputStream "in", that is not being deallocated and is being returned in a field. While RLFixer presents this as a plausible leak to be fixed, the InputStream is being returned in a method inside a field escape, making this leak unfixable. RLFixer's internal implementation struggles to identify field escapes as not solvable.

```
1  public static UrlAssetInfo create(AssetManager assetManager,
       AssetKey key, URL url) throws IOException {
2          URLConnection conn = url.openConnection();
3          conn.setUseCaches(false);
4          InputStream in = conn.getInputStream();
5
6          if (in == null){
7              return null;
8          }else{
9              return new UrlAssetInfo(assetManager, key, url, in);
10         }
11     }
```

Fig. 5: InputStream, in, is being escaped into a field and being returned in the method. Adapted from benchmark jMonkeyEngine, file UrlAssetInfo.java.

Running Arodnap on all 9 benchmarks took strenuous effort to ensure it met Arodnap's expected project structure. An individualized script was used to normalize each benchmark. Many projects had submodules with their own source directories and differing build structures, making it difficult to create a single automated script that would work for every project. One challenge encountered was gathering all the required library dependencies. Some libraries were missing because certain projects failed to compile. When a project failed to compile, Maven, a Java-based build and dependency management tool, was unable to download and upload the dependent libraries to the ".m2" directory during the build process. Another observation was the time it took to run the Arodnap pipeline. While most benchmarks finished running the entire Arodnap pipeline in roughly 20-30 minutes, projects such as H2 and Checkstyle took over an hour to finish. Different machines might require a shorter or longer time to finish running Arodnap. During the compilation validation of Arodnap's generated patches, there were Java errors presented from applying the patches. After diagnosis of the problem, we determined a configuration error where not all of the jars were being added to the class path, generating incorrect patches. In addition, various benchmarks contained multiple submodules, each having source folders, messing up the logic of Arodnap's patching. This issue did not occur in the NJR-1

benchmark since all projects did not contain multiple submodules with their own source folder. Due to this fix, the Arodnap patch compilation successfully worked and rejected incorrect patches. The entire process of making each benchmark into a normalized structure and running Arodnap with applying patches is not fully automated. Automating this process would increase the usability of this repair tool and prevent user configuration mistakes from happening.

ADDITIONAL WORK

In addition to past work done on SootUp, I worked on the "Field Can Be Final" plugin, which analyzed programs in the NJR-1 benchmark to determine if resource leaks should be made final. Setting resources to final prevents the resource from being initialized, and the compiler will flag an error if this happens. I parsed over 50 benchmarks to test the validity of the plugin and create unit tests to debug and fix errors in the plugin. This early work preceded the inclusion of setting resource fields to final in Arodnap, allowing for more actionable leak warnings.

During the initial phase of testing Arodnap on real-world Java applications, I selected Recaf [21], a modern Java bytecode editor, due to potential resource leaks of file streams. When running Arodnap, I received two different errors from Javaparser, a library that parses Java code into an Abstract Syntax Tree, stating that switch expressions and patterns of instanceof, a Java keyword, are not supported. I determined that switch expressions and instanceof became a permanent feature in Java 14 and Java 16, respectively. Since RLFixer is compatible with up to Java 11 programs and Recaf is compatible with Java version 22 and higher, running Java applications that use exclusive Java 12 and higher syntax will result in compilation errors from the javaparser. Due to time constraints, we were forced to focus on running Java 11 applications. As a future direction, RLFixer's code base should support the most recent Java versions.

LIMITATIONS

Arodnap uses RLFixer, whose tool and benchmark projects use Java 11. As a result, it was difficult to find many real-world projects that still use older versions of Java. To satisfy Arodnap's Java 11 requirement, most of the benchmarks use older releases of the projects. The benchmarks tested in this study cover only a limited number of projects and do not span all domains or code styles, which may affect the generalizability of the results. In addition, most of the benchmarks contained resource leaks that escaped to a field, which resulted in fewer patches being generated relative to the total number of resource leaks. This represents a limitation in improving resource leak detection when resources escape. In these cases, automated tools are unable to reliably determine ownership to properly close the resource.

FUTURE RESEARCH DIRECTIONS

RLFixer uses outdated open-source library dependencies such as WALA, a Java framework for static program analysis of Java bytecode, and Javaparser that supports analyzing code up to Java 11. In our study, we used WALA 1.5.7 and Javaparser 3.24.7. This prevents the use of Java 17 or higher benchmarks from being run on Arodnap. Prior work has been conducted to address this limitation by upgrading both WALA and Javaparser versions, which would be able to parse modern Java code. However, despite upgrading the versions, RLFixer uses old WALA functionality throughout the codebase, which is not acceptable in newer WALA versions. After further diagnosis, the outdated WALA version that was used relied on assumptions from older Java versions, which packaged the standard library differently. While this was true in older versions of Java, newer versions organize the library differently. As a result of this mismatch, RLFixer is having trouble locating the necessary runtime classes, and adapting it to the newer

format has introduced additional challenges. Further work needs to be done to successfully update WALA properly into RLFixer to be used in Arodnap.

While this study runs Arodnap with an enhanced implementation of RLFixer, we did not run RLFixer by itself on the 9 benchmarks. Arodnap identifies issues with RLFixer's escape reasoning, noting that it is overly conservative in the presence of resource accessor objects that use a resource but do not own it [8]. A resource accessor object is an object that stores resources in a field, but does not close it. RLFixer treats this assignment as a field escape and therefore avoids fixing the leak. However, Arodnap can detect cases where the resource can be safely closed, provided the accessor does not own the resource and does not outlive it. Running RLFixer separately on each benchmark can help highlight and support Arodnap's claim.

With Arodnap's current implementation, developers must manually normalize their projects or rely on custom scripts to meet Arodnap's requirements. Future work includes developing an automated script that conforms projects to Arodnap's expected structure, thereby improving its usability.

CONCLUSION

Arodnap is a novel approach to improve resource leak repair and enhance an existing version of RLFixer to generate better reasoning upon resource leaks, especially wrappers. Prior work involved Arodnap being tested on the NJR-1 benchmark, but there has been no study on real-world applications. This study examined 9 open source Java applications on Arodnap to test its usability and identify the effectiveness of the tool. Arodnap uses RLC with inference to apply annotations to track the flow state of resources from allocation to deallocation. It performs code transformation, such as adding final qualifiers to eligible fields, and uses a novel field containment analysis inside RLFixer to enable repair of leaks involving resources stored in

fields. Arodnap produces concrete patches that fix all of the targeted resource leaks in each of the benchmarks. While there were many resource leaks present in the applications, due to the allocation and usage of resources in various projects, the majority of the leaks failed to be solved due to field escapes. However, the leaks that Arodnap generated patches were successfully deallocated and resolved through the use of try-with-resources statements and try-finally blocks. While manual intervention was involved for patches targeting different resources around the same lines of code, no compilation errors were introduced by the patches. In order for Arodnap to be effective for Java developers, automated processes to normalize the applications and apply the patches are the next steps to improve resource leak checking and repair for Java programs.

REFERENCES

[1] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst, "Practical pluggable types for Java," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA '08)*, 2008, pp. 201–212, https://doi.org/10.1145/1390630.1390656.

[2] D. Hovemeyer and W. Pugh, "Finding bugs is easy," in *Companion to the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '04)*, 2004, pp. 132–136. [Online]. Available: https://doi.org/10.1145/1028664.1028717

[3] M. Kellogg, N. Shadab, M. Sridharan, and M. D. Ernst, "Lightweight and modular resource leak verification," in *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2021)*, 2021, pp. 181–192, https://doi.org/10.1145/3468264.3468576.

[4] N. Shadab, M. Kellogg, A. Utture, M. Sridharan, and M. D. Ernst, "Inference of resource management specifications," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA2, Art. no. 282, pp. 1–24, Oct. 2023, https://doi.org/10.1145/3622858.

[5] C. Wang, J. Liu, X. Peng, Y. Liu, and Y. Lou, "Boosting static resource leak detection via LLM-based resource-oriented intention inference," *arXiv*, 2024, arXiv:2311.04448. [Online]. Available: https://doi.org/10.48550/arXiv.2311.04448

[6] C. Wang, Y. Lou, X. Peng, J. Liu, and B. Zou, "Mining resource-operation knowledge to support resource leak detection," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2023)*, 2023, pp. 986–998, https://doi.org/10.1145/3611643.3616315.

.

[7] A. Utture and J. 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 (ESEC/FSE 2023)*, 2023, pp. 159–171, https://doi.org/10.1145/3611643.3616267.

[8] S. Malakar, M. D. Ernst, M. Kellogg and M. Sridharan, "Repairing Leaks in Resource Wrappers," *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Seoul, Korea, Republic of, 2025, pp. 828-840, https://doi.org/10.1109/ASE63991.2025.00074.

[9] J. Palsberg and C. V. Lopes, “NJR: A normalized Java resource,” in *Proceedings of the 2018 ACM SIGPLAN International Conference on Software Engineering Companion (ISSTA Companion/ECOOP)*, 2018, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3236454.3236501

[10] “Arodnap,” https://github.com/typetools/arodnap

[11] Eclipse JGit Contributors, “JGit,” GitHub repository, 2026. [Online]. Available: https://github.com/eclipse-jgit/jgit

[12] H2 Database Contributors, “h2database: H2 is an embeddable RDBMS written in Java,” GitHub repository, 2026. [Online]. Available: https://github.com/h2database/h2database. Accessed: Feb. 12, 2026.

[13] K. Karakaya, S. Schott, J. Klauke, E. Bodden, M. Schmidt, L. Luo, and D. He, “SootUp: A redesign of the Soot static analysis framework,” in *Tools and Algorithms for the Construction and Analysis of Systems*, B. Finkbeiner and L. Kovács, Eds., Cham:

Springer Nature Switzerland, 2024, pp. 229–247. [Online]. Available: https://doi.org/10.1007/978-3-031-57246-3_13

[14] jMonkeyEngine Contributors, "jMonkeyEngine: A complete 3-D game development suite written in Java," GitHub repository, 2026. [Online]. Available: https://github.com/jMonkeyEngine/jmonkeyengine. Accessed: Feb. 12, 2026.

[15] BioJava Contributors, "BioJava: An open-source Java library for processing biological data," GitHub repository, 2026. [Online]. Available: https://github.com/biojava/biojava. Accessed: Feb. 12, 2026.

[16] Netty-SocketIO Contributors, "netty-socketio: Socket.IO server implemented on Java," GitHub repository, 2026. [Online]. Available: https://github.com/mrniko/netty-socketio. Accessed: Feb. 12, 2026.

[17] Jackson Databind Contributors, "jackson-databind: General data-binding package for Jackson (2.x)," GitHub repository, 2026. [Online]. Available: https://github.com/FasterXML/jackson-databind. Accessed: Feb. 12, 2026.

[18] Checkstyle Contributors, "checkstyle: A development tool to help programmers write Java code that adheres to a coding standard," GitHub repository, 2026. [Online]. Available: https://github.com/checkstyle/checkstyle. Accessed: Feb. 12, 2026.

[19] JaCoCo Contributors, "jacoco: Java Code Coverage Library," GitHub repository, 2026. [Online]. Available: https://github.com/jacoco/jacoco. Accessed: Feb. 12, 2026.

[20] G. Fourtounis and Y. Smaragdakis, "Deep static modeling of invokedynamic," in *Proceedings of the 33rd European Conference on Object-Oriented Programming (ECOOP 2019)*, 2019, Art. no. 15. https://doi.org/10.4230/LIPIcs.ECOOP.2019.15.

[21] Col-E Con, *Recaf: The modern Java bytecode editor* [Online]. Available: https://github.com/Col-E/Recaf. Accessed: Feb. 12, 2026.