Poster: Memory and Resource Leak Defects in Java Projects: An Empirical Study

Mohammadreza Ghanavati
Heidelberg University
mohammadreza.ghanavati@informatik.uni-heidelberg.de

Diego Costa
Heidelberg University
diego.costa@informatik.uni-heidelberg.de

Artur Andrzejak
Heidelberg University
artur.andrzejak@informatik.uni-heidelberg.de

Janos Seboek
Heidelberg University
seboek@cl.uni-heidelberg.de

ABSTRACT

Despite many software engineering efforts and programming language support, resource and memory leaks remain a troublesome issue in managed languages such as Java. Understanding the properties of leak-related issues, such as their type distribution, how they are found, and which defects induce them is an essential prerequisite for designing better approaches for avoidance, diagnosis, and repair of leak-related bugs. To answer these questions, we conduct an empirical study on 452 issues found in repositories of 10 mature Apache Java projects.

CCS CONCEPTS

• **General and reference → Empirical studies**; • **Software and its engineering → Memory management**; **Software defect analysis**;

KEYWORDS

Empirical study, memory leak, resource leak, leak detection, fault localization

ACM Reference Format:
Mohammadreza Ghanavati, Diego Costa, Artur Andrzejak, and Janos Seboek. 2018. Poster: Memory and Resource Leak Defects in Java Projects: An Empirical Study. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195032

1 INTRODUCTION

Resource or memory leaks are caused by software defects related to mismanagement of system resources. Newer programming languages offer support for avoiding leaks, particularly memory leaks. In managed languages such as Java, C#, or Go a Garbage Collector (GC) is responsible for memory management. However, objects which are not used but are still reachable by a chain of references cannot be released by a GC. For example, obsolete object references from collections are examples of frequent defects causing memory leaks in Java [1]. Even in managed languages, the programmer is responsible to release finite system resources such as file handles, threads, or database connections after their use. A typical defect pattern observed here is lack of releasing these resources if exceptions occur.

Diagnosis of leak-related defects is an important problem for both researchers and practitioners [2]. Various approaches have been introduced for leak detection and diagnosis based on static and or dynamic analysis [4–9]. Other recent efforts target prevention of leaks via programming language support such as new language construct introduced in Java 7, the *try-with-resources* statement or Smart pointers in C++.

The impact of these research activities, tools and language enhancements critically depends on whether they target prevalent or rare types of leak defects, whether they can handle difficult cases, and whether their assumptions are realistic enough to apply in practice. For instance, Xu [8] proposes a method for detecting leaks caused by obsolete references from within object containers, but provides only a limited evidence that this is a frequent cause of leak-related bugs. To this end it is important to understand the properties of leak-related defects, their detection methods, and repair activities in a quantitatively representative way.

To our knowledge, available studies in this field (e.g., [3]) are still limited in terms of sample size and depth of analysis. We believe that researchers and practitioners would benefit from a systematic empirical study of a large sample of leak-related defects found in real-world applications.

Table 1: Studied projects with statistics on their size and number of issues.

Project	#Issues	#Bugs	#Fixed	# Leak Issues
AMQ	123	116	88	80
CASSANDRA	77	65	45	35
CXF	62	61	44	37
DERBY	50	36	23	16
HADOOP	236	201	132	119
HBASE	92	65	44	40
HIVE	78	69	47	44
HTTPCOMP.	31	28	24	20
LUCENE	77	65	42	34
SOLR	74	60	33	27
Total	**900**	**766**	**522**	**452**

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195032

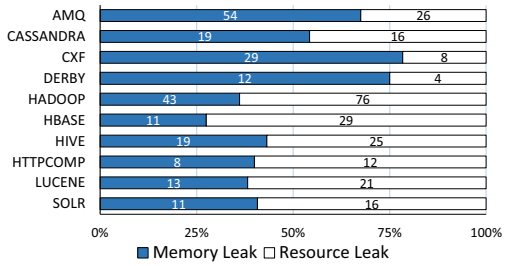

Figure 1: Frequency of leak types per project.

2 METHODOLOGY

Our study uses ten open source large-scale Java projects from Apache repository (Table 1). The investigated projects use a centralized bug tracker called JIRA to track the issue reports. We obtain the dataset for our study via the four steps described below. Our methodology yields a dataset with **452** leak-related issues, each representing a unique leak bug.

Keyword search. We select only issues that contain the keyword "leak" in the issue title or issue description.

Issue type. As we are only interested in leak-related bugs, we subsequently keep only the issues of type *Bug*.

Issue resolution. We restrict our analysis to bug issues for which there is at least one associated repair patch.

Manual investigation. In the final step we manually inspect and remove the false positives from our dataset by filtering out the following issues: 1) non-leaky bugs retrieved by our keyword search heuristic, and 2) wrongly reported leaks by developers.

3 PRELIMINARY RESULTS

We provide here preliminary results on the leak types, the detection types, and the root causes of the leak-inducing defects.

What are the types of leak defects in studied applications? Leaks can be triggered by unreleased references (memory leak) or mismanagement of finite system resources (resource leak). Figure 1 shows the distribution of these two leak types across the studied projects. In projects *AMQ*, *CXF*, and *DERBY* memory leak issues are more reported, while in projects *HADOOP* and *HBASE* more than 60% of the issues are identified as resource leak. In the other projects, the proportion of the reported resource and memory leak issues are about equal.

How are leak-related defects detected? Understanding how the leaks manifest themselves as a defect and how developers detect the leak in the application can provide valuable insights on the leak diagnosis. Table 2 shows the distribution of major detection approaches used by the developers of the considered projects. Interestingly, code analysis (33%) and runtime analysis (23%) are still the dominant methods.

Table 2: Distribution of defect detection methods.

Detection method	# Leak Issues
Code analysis	151 (33.4%)
Runtime analysis	106 (23.5%)
Failed test	55 (12.2%)
Out-of-memory error	51 (11.3%)
Warning	15 (3.3%)
Combined	74 (16.4%)
Total	**452 (100%)**

Table 3: Leak-triggering defect types and their distribution.

Description of a defect type	# Leak Issues
Non-closed resource at error-free execution	135
Object not disposed of if exception is thrown	93
Dead objects referenced by a collection	89
Unreleased reference at error-free execution	29
A race condition defect (concurrency bug)	17
Wrong call schedule of dispose of function	15
Strong reference instead of a weak reference	14
Over-sized cache or buffer	12
Incorrect API usage (wrong method or parameter)	10
Bugs not belonging to the above categories	38

What are the common root causes of leak-inducing defects? To uncover any potential common patterns among the leak-triggering defects we studied in-depth each issue and the code of the corresponding patches. Table 3 shows that there are indeed dominant types of defects, with top 3 accounting for 70% of all issues. Our findings can support development of automated leak detection approaches by pointing to most common defect types.

4 FUTURE WORK

In our future work we intend to refine and extend the results from Section 3 by focusing on the following research questions: How are the leak-related defects repaired? How complex are the repair patches? To what degree are all these taxonomies related to each other? We believe that answering these questions can be beneficial for improving and possibly automatizing the process of localizing leak-related faults, their root cause analysis, and manual or automated bug repair. Our quantitative findings will be used as a basis for drawing implications for practitioners and researchers on topics related to leak-related issues.

REFERENCES

[1] Mohammadreza Ghanavati and Artur Andrzejak. 2015. Automated memory leak diagnosis by regression testing. In *IEEE SCAM*. 191–200.
[2] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. How Practitioners Perceive the Relevance of Software Engineering Research. In *ESEC/FSE*. 415–425.
[3] Fumio Machida, Jianwen Xiang, Kumiko Tadano, and Yoshiharu Maeno. 2012. Aging-Related Bugs in Cloud Computing Software. In *ISSREW*. 287–292.
[4] Rivalino Matias, Artur Andrzejak, Fumio Machida, Diego Elias, and Kishor S. Trivedi. 2014. A Systematic Differential Analysis for Fast and Robust Detection of Software Aging. In *SRDS*. 311–320. https://doi.org/10.1109/SRDS.2014.38
[5] N. Mitchell and G. Sevitsky. 2003. LeakBot: An Automated and Lightweight Tool for Diagnosing Memory Leaks in Large Java Applications. *LNCS* 2743 (2003), 351–377.
[6] V. Sor, P. Oü, T. Treier, and S. N. Srirama. 2013. Improving Statistical Approach for Memory Leak Detection Using Machine Learning. In *ICSM*. 544–547.
[7] Guoqing Xu, Michael D. Bond, Feng Qin, and Atanas Rountev. 2011. LeakChaser: Helping Programmers Narrow Down Causes of Memory Leaks. In *PLDI*. 270–282.
[8] Guoqing Xu and Atanas Rountev. 2008. Precise Memory Leak Detection for Java Software Using Container Profiling. In *ICSE*. 151–160.
[9] Dacong Yan, Guoqing Xu, Shengqian Yang, and Atanas Rountev. 2014. LeakChecker: Practical Static Memory Leak Detection for Managed Languages. In *CGO*. Article 87, 11 pages.