

TOSEM-2021-0317: Revision

Dear Editorial Office of *ACM Transactions on Software Engineering and Methodology*,

We are happy to submit a major revision of our paper entitled “**Coverage-Based Debloating for Java Bytecode**” exactly 43 days after the notification. We thank the reviewers for their detailed recommendations. We have addressed all their comments, which has improved the quality of our manuscript.

We performed five major revisions:

- We added a new research question (RQ5) comparing JDBL and JSHRINK. We compare the debloating results and execution time for 17 case studies from the original paper of Bruce et al. [1]. Asked by reviewers in comments #1.7, #2.1, #2.2, and #3.3.
- We added two additional examples (Examples 4 and 5) showing the challenges of collecting accurate and complete coverage information for debloating. Asked by reviewers in comments #1.1, #1.3, #2.5, and #3.4.
- We improved the description of the data collection procedure that we followed. A more detailed description of our dataset is also available in Durieux et al. [3]. Asked by reviewers in comments #1.4, #1.5, #3.11, and #3.12.
- We added a new section (Section 6.1) that quantifies and discusses the complementary contribution of each coverage tool and the benefits of using this approach for debloating. Asked by reviewers in comments #1.16 and #2.6.
- We clarified the impact of the different libraries and their versions w.r.t the results reported in our experiments. We show that the results are not biased due to the different library versions analyzed. Asked by reviewer in comments #3.1.

In the following pages, we give detailed answers to each of the reviewers’ comments. The original text from the reviewers is included in boxes, our answers follow the boxes. All changes are highlighted in blue in the revised version of the manuscript (except typos).

In case of requiring any further information, please do not hesitate to contact us.

Sincerely yours,

César Soto-Valero

On behalf of Thomas Durieux, Nicolas Harrand, and Benoit Baudry

Reviewer #1

Main issues

Comment #1.1: “For me it was not clear why, starting from the title, the discussion was done about bytecode, but the listings presented in the section show the actual Java code. I believe, this was done to make the examples more understandable for a reader. However, it hides the actual changes in the bytecode. So, I would suggest to at least add the corresponding bytecode to the provided listings. For example, Listing 4 could be put on a side of Listing 3. And then the authors could show how the code in Listing 3 corresponds to the bytecode in Listing 4.”

We thank the reviewer for this comment. JDBL debloats bytecode based on the coverage information collected from various code coverage tools. As discussed in Section 3.1, collecting accurate and complete coverage information for debloating is challenging. The listings presented in this section shows some representative examples of the mismatch between the actual code usage (as expressed by the tests) and the coverage information reported by coverage tools.

We show source code only (except for example #3) to facilitate the readers’ understanding of these examples. For further reference to the examples, the source code and coverage reports of all the listings is available in the companion repository along the rest of the paper’s data and experiments: https://github.com/castor-software/jdbl-experiments/tree/master/revision/coverage_examples.

Comment #1.2: “Listings placed in the paper far from the actual examples. I would suggest to place them right inside corresponding examples.”

Following the reviewer’s suggestion, we have put all the listings side-by-side, which improves the reading experience of the reader.

Comment #1.3: “The challenges section 3.1 should contain all the challenges addressed in the paper. For example, I did not find the challenges for the solutions presented in 3.2.2 and 3.2.3.”

We now discuss two additional challenges in Section 4.1 (Examples #4 and #5). There is at least one example of the challenge for each solution presented in Sections 3.2.2 and 3.2.3.

Comment #1.4: “I did not understand the motivation for several library selection steps in the data collection process (section 4.2). For example, why only single-module Java projects? Why the projects that do not declare a fixed release are excluded? Why not use git tags to identify commit hashes of the released versions? I think, the clear motivations for each step is needed.”

In Maven, modules in a multi-module project are built and packaged separately. This significantly increases the technical difficulties of debloating because each module has a separate classpath and a separate test suite. Also, modules may depend on each other (a.k.a. internal dependencies), which complicates the removal of classes in projects with dependent modules. JDBL is engineered with usability in mind: the user only needs to edit the *pom.xml* file when using JDBL. We performed our experiments on single-module projects to facilitate usability and obtain a single debloated JAR file that we can use to assess the effectiveness of debloating.

We did not consider clients that do not declare a fixed dependency version because it was not possible to identify the library version actually needed to compile and run the client, and that information is essential for our experiment.

We did not use the Git tags because they are not widely adopted. Moreover, the Git tag version frequently has a different naming convention than the Maven version, adding complexity. We tried both approaches and kept the extraction of the version number from the number as a more reliable approach.

Comment #1.5: “My other issue about the data collection section - it is presented in a text form, which makes it difficult for reader to quickly understand the library selection steps. I would suggest to improve the presentation of the selection procedure. Maybe a figure can improve the clarity.”

We thank the reviewer for this comment regarding the presentation. We have made an effort to present our data collection pipeline in text form in order to save space for other sections of the paper. We have edited and improved the presentation of the data collection procedure (see Section 4.2). We have enumerated the data selection steps and added more details about the dataset. A more detailed description of the dataset, which includes an illustrative figure, is presented in Durieux et al. [3].

Comment #1.6: “Table 1 shows the coverage according to the JaCOCO tool, however, other tools are then used to generate the coverage. So, I think, the total coverage generated as a result of running all the tools needs to be included in the table.”

We thank the reviewer for this comment. We have edited Table 1 to show the total class coverage of the libraries. We computed these descriptive statistics by aggregating the coverage reports collected by the four coverage tools that we use for debloating (see Section 3.1.2). We have shown class coverage only because, as discussed in Section 6.1, the JVM class loader does not provide more fine-grained coverage information.

Comment #1.7: “The analysis of results are presented for the ecosystem. However, the view of a single library is missing. I believe, presenting the impact of debloating for an individual library will significantly improve the paper.”

We thank the reviewer for this comment. We have presented the impact of debloating for a benchmark of 17 individual Java libraries in a new research question (RQ5), which was added in the revised version of the paper. In this research question we have assessed the impact of coverage-based debloating for these 17 case studies in the benchmark of Bruce et al. [1] w.r.t the percentage of size reduction of the debloated artifact and the behavior preservation after debloating based on the execution of the original tests (see Section 5.2.3). Moreover, we present a discussion of the debloating execution time in this benchmark of individual projects in a new section (see Section 6.2).

Comment #1.8: “The analysis of errors presented in the paper is done according to the black-box approach. While this provides some ideas on the types of the errors, such analysis does not provide insights on how JDBL could be improved, or which part of JDBL likely to cause such errors, or for which libraries JDBL is likely to generate such kind of errors. Hence, I believe, the analysis of errors needs to be improved.”

We did a manual analysis of the errors for most of the cases. This manual analysis was supported by a custom-made dashboard (accessible at <http://130.237.222.185:8881>) to identify the problems that JDBL has faced. We did not notice any particular pattern from our analysis of the errors. The remaining problems are due to custom configurations and specificities of some projects, which are hard to deal with, such as classpath incompatibility between different Maven plugins. We decided not to present a deeper analysis because we did not identify any specific take-away from our manual analysis.

Other issues

Comment #1.9: “The summary of contributions in the Introduction section is not clear. In particular, at this moment the reader does not know why there were different sets of 395 and 220 libraries for the third and fourth contributions. I would suggest the authors to clarify the contributions and base them on the information present in the Introduction section.”

We thank the reviewer for this remark. We have edited the list of contributions in the Introduction section. We have clarified and removed the unnecessary information regarding the experimental results to avoid confusing the reader at this point of the paper.

Comment #1.10: “Figure 1 - What about within library dependencies? I.e. when one class/method depends on the other class/method of the same library. I believe, such details in the example will be useful for the reader to completely understand the situation with the code reuse in Java projects.”

We have modified Figure 1 to illustrate the situation pointed out by the reviewer. We added inter-library code reuse examples within JPROJECT, dependency **A**, and dependency **D**. We edited the caption of the figure to reflect this change. We also improved the description regarding the colors, so that impaired people can understand the figure, see Comment #3.24.

Comment #1.11: “In the definition of the coverage-based debloating (lines 176 - 180), the authors are using the term “correct program”. I believe, the clarification is needed what is understood by the correct program.”

By “correct program” we mean a “compilable program.” We have clarified this by editing the definition. We changed the term from “correct program” to “syntactically correct program.”

Comment #1.12: “Footnote 7: I would suggest to use the direct link to the Maven Repository: <https://repo.maven.apache.org/maven2/>.”

We have changed the footnote to link directly to the Maven Repository <https://repo.maven.apache.org/maven2/>, as suggested by the reviewer.

Comment #1.13: “The authors start analysis from the source code. And then their first step becomes the compilation of the projects. For me this was not clear. Why not to start directly from the compiled distributed Jar archives (e.g. available from Maven Central).”

We do not debloat the compiled JAR files directly because they do not include the tests. We use the tests as the workload that exercises the project. Our coverage-based debloating technique, implemented in JDBL, debloats a Java project during its build life-cycle. JDBL relies on the Maven’s compilation and testing phases to perform the coverage collection, bytecode transformation, and validation of syntactic and semantic correctness of the debloated artifact (see Section 3.3).

This approach has several advantages compared with existing debloating techniques:

- It facilitates the integration of JDBL: developers only need to add a few lines in the *pom.xml* file of the project (Listing 1 shows an example).
- The debloating procedure occurs within the regular project build life-cycle.
- The debloating procedure is reproducible as long as the project do not have flaky tests.
- A new debloated artifact with all its debloated dependencies (*jar-with-dependencies*) is generated after the project’s build.

```
1 <plugin>
2   <groupId>se.kth.castor</groupId>
3   <artifactId>jdbl-maven-plugin</artifactId>
4   <version>1.0.0</version>
5   <executions>
6     <execution>
7       <goals>
8         <goal>trace-based-debloat</goal>
9       </goals>
10      </configuration>
11    </execution>
12  </executions>
13 </plugin>
```

Listing 1: Using JDBL as a Maven plugin that executes within the *package* phase. This facilitates its execution and integration into the project’s build life-cycle.

Comment #1.14: “The authors several time say that they disable surefire plugin. However, no motivation is presented for this action. I believe, this motivation would help the reader to follow the decisions made by the authors of the paper.”

We thank the reviewer for this remark. We have edited Section 4.3.2 of the paper with additional clarifications about our motivation to set up the *maven-surefire-plugin* to its default

configuration.

Note that we explain why we disabled the `maven-surefire-plugin` in Section 4.3.1 (Coverage-Based Debloating Execution):

“The second step adds JDBL as a plugin inside the Maven configuration (pom.xml) and resets the configuration of the `maven-surefire-plugin`. This reset ensures that its original configuration is not in conflict with the execution of the coverage collection phase of JDBL. A manual configuration of JDBL could prevent this problem, but in order to scale up the evaluation, we decided to standardize the execution for all the libraries.”

In addition, we further stress the motivation of this modification in Section 6.3 (Internal Validity):

“JDBL relies on the official Maven plugins for dependency management and test execution. Still, due to the variety of existing Maven configurations and plugins, JDBL may crash at some of its phases due to conflicts with other plugins. For example, the Maven `shade` plugin allows developers to change the produced JAR by including/excluding custom dependencies; or the `maven-surefire-plugin` allows excluding custom test cases, thus altering the validation phase of JDBL. To overcome this threat and to automate our experiments, we set the `maven-surefire-plugin` to its default configuration, and use the `maven-assembly-plugin` to construct the JAR of all the study subjects.”

Comment #1.15: “In the RQ1, the authors analyse the validation errors. However, no insights about the crash category errors are provided. Also, there were 13 time-out errors, but no information provided on how long was the time-out set. I would suggest to complete the paper with this information.”

As mentioned in Comment #1.8, the causes of crashing builds are diverse and difficult to diagnose in general. This is due to the aggressive nature of the debloating transformations, which may cause the build to stop at some phase and terminate abruptly, i.e., due to accessing invalid memory addresses, using an illegal opcode due to JVM incompatibility, or triggering an unhandled exception. We have performed an additional manual investigation of the build crashes. We implemented a dashboard to facilitate the filtering and analysis of the complete results of our debloating experiments, which is accessible for the interested reviewer here: <http://130.237.222.185:8881>.

Regarding the execution time-out, we set this value to 1h in our experiments. We have added this information in Section 4.3.1 (Coverage-Based Debloating Execution) to make the description of our experiments more accurate.

Comment #1.16: “lines 1282 - 1285 “the failures of one specific tool are likely to be corrected by other tools”. I believe, a more precise analysis is needed to support this statement. The complete solution would be to generate a table that presents all the cases when one tool helps to fix the failures of another tool. But at least manual investigation of some of the failures has to be presented.”

We have added a new Section 6.1 that discusses the complementarity of the code-coverage tools for debloating. The Venn diagram presented in this section shows that there is not one single tool that can capture all the code used at runtime and that each tool captures specific corner cases.

Comment #1.17: “line 1299 ‘which cover projects from different domains’. I would suggest the authors to expand the domains covered by their selected libraries.”

We have expanded this sentence in Section 6.3.2 by explicitly listing the most relevant domains covered by the selected libraries. We also added a footnote there with the explicit list of libraries used in our experiments.

Reviewer #2

Comment #2.1: “My major concern with the paper, however, is that it does not compare to the state of the art. Table 7 gives some insight into the different categories of approaches and implementations, but there is no evaluation performed with the five competing tools. While I see that all of them target different parts of the programs to be reduced, I think that a comparison at least against JSRINK should be possible here. From my comparison of the numbers in the evaluation of JDBL and JSRINK reported in the respective papers, JDBL should be superior to JSRINK. If this is actually the case in a controlled comparative experiment, this would be a major contribution to the field.”

We thank the reviewer for this constructive comment. We have reproduced the JSRINK experiments on the same benchmark reported in the companion paper by Bruce et al. [1]. To do so, we forked the JSRINK project on GitHub and updated its functionalities to execute the original experiments in a Docker container, see <https://github.com/castor-software/jshrink>.

We performed a comparison between JSRINK and JDBL in terms of JAR size reduction and behavior preservation. The comparison is based on the 17 single-module Maven projects of Bruce’s benchmark (JDBL is designed to work on single-module projects). The comparison is discussed a new section 5.2.3, which answers a a new research question (RQ5):

“RQ5: How does coverage-based debloating compares with the state-of-the-art of Java debloating regarding the size of the packaged artifacts?”

Our results show that JDBL outperforms JSRINK in both average size reductions while passing all the tests with the debloated artifacts. Moreover, we present an additional comparison of the execution time of both tools in a new Section 6.2. The benchmark of projects debloated with JDBL is accessible on the companion repository of our paper at https://github.com/castor-software/jdbl-experiments/tree/master/revision/benchmark/projects_debloated_with_jdbl. We believe the results reported in our paper w.r.t this benchmark is a comparison point that will benefit future researchers working on Java bytecode debloating.

Comment #2.2: “A medium concern currently is reporting of runtimes of the approach. The authors report a total runtime of their experiments of 4.5 days and debloat times of 1.5 days. It would be interesting to see descriptive statistics on the runtime of the approach (avg, median, quartiles) to see if it is feasible for every-day software development. I would appreciate to see such statistics in a future version of the paper. Also the comparison to JSRINK would be very interesting.”

We thank the reviewer for this insightful comment. We have added a new discussion section (see Section 6.2) comparing the execution time of JDBL and JSHRINK. To facilitate further research in this direction, we have made available on GitHub the complete results of the benchmark of projects of Bruce et al. [1] debloated with JDBL: https://github.com/castor-software/jdbl-experiments/tree/master/revision/benchmark/projects_debloated_with_jdbl.

Comment #2.3: “line 141: You mention that your objective here is attack surface reduction, but you do not evaluate that. I would suggest removing this.”

We have removed “attack surface reduction” from the objectives of this paper. Attack surface reduction is part of our future work on assessing the impact of debloating in Java applications.

Comment #2.4: “line 221: Please introduce the term “artifact” before usage. It also might be misleading here.”

We have clarified the term “artifact” in paragraph 4 of the Introduction section (where it is first mentioned). We have also edited the sentence pointed out by the reviewer to make clear there that by artifact we meant “the bytecode of a compiled project.”

Comment #2.5: “lines 262ff: I found the three examples a bit hard to follow. Is there a systematic to follow (from [21] or [30]) maybe? In the current presentation is is hard to see for the reader that the authors address all problematic cases here. As this is key to the presented approach, I think this should be addressed.”

We have edited the description of the three examples in Section 3.1. We have added two additional examples (Examples #4 and #5) to facilitate the comprehension of the reader. These examples cover all the problematic cases addressed in Section 3.2. The source code of all the listings presented in the paper is available here: https://github.com/castor-software/jdbl-experiments/tree/master/revision/coverage_examples.

Comment #2.6: “lines 334ff: From the description it sounds as if JCov is the best choice here and does not have any problematic cases. Why is the approach not using this solely, if correct?”

We have added a discussion about the complementarity of the code-coverage tools in new section (Section 6.1). From the new Venn diagram presented in this section, it can be observed that JCov covers 39,596 (51.7%) of the classes.

Comment #2.7: “lines 353ff: I agree with the authors here, but t would be good to have some numbers later in the paper for this, so see how minimal this is.”

We have computed the size of all the `class` files in the original bundled JAR files of our dataset. We compared the bytecode size of interfaces, enums, annotations, and exceptions with respect to the bytecode size of the rest of the `class` file. According to our results, these constructs represent

15.8% of the size on disk of the JAR files. We have reported this finding in Section 5.2.2. The Java project of the tool that we implemented to perform this particular analysis is available at: <https://github.com/castor-software/jdbl-experiments/tree/master/revision/TypeAnalyzer>.

Comment #2.8: “Algorithm 1: The caption says “bytecode”, but the algorithm compiles from source code. This is a bit confusing.”

We thank the reviewer for this comment. We have edited the caption of Algorithm 1 to reflect the fact that the debloating procedure receives a Java project as input.

Comment #2.9: “Algorithm 1: I was confused about \mathcal{W} . What are its members? What does it mean to execute a complete workload?”

The *workload* \mathcal{W} is a set of valid inputs belonging to the *input space* of a compiled Maven. The *input space* of a compiled Maven project is the set of all valid inputs of its public API. Workloads are particularly useful for performing dynamic analysis. For example, to detect distinct execution paths in software applications through profiling and observability tasks, or using monitoring tools to analyze how the application reacts to different workloads at runtime. For our debloating experiments, we consider \mathcal{W} as the complete test suite of a Maven project, where each $w \in \mathcal{W}$ is an individual JUnit test executed by Maven. We have added this explanation in Section 3.3.1 (Coverage Collection).

Comment #2.10: “RQ2: The question currently implies completeness which the approach does not provide by design. I suggest to rephrase and add “in the inspected context” or something similar.”

We thank the reviewer for this suggestion. We have rephrased RQ2 to explicitly reflect the fact that we assess behavior preservation of the debloated libraries based on the workload used for debloating.

RQ2 now reads as follows:

“RQ2: To what extent do the debloated library versions preserve their original behavior w.r.t the debloating workload?”

Comment #2.11: “Table 1: Client coverage is VERY low. This introduces a lot of uncertainty in the inspection of RQ6 and is not mentioned in the threats to validity section.”

We agree with the reviewer in this regard. The clients’ coverage is not very high (20.24% on average), which may cause that some used functionalities in the debloated libraries are not executed by the clients’ tests. We have added text in Section 6.3 (Construct Validity) to point out the low coverage of the clients as a threat to the validity of our results for RQ6.

Comment #2.12: “RQ5/Section 4.3.4/Section 5.3.1: I think this inspection is not necessary. As the debloated libraries are created in the build process of the application it does not matter if a client can be compiled against these libraries. They need to be binary compatible to the client, of course, but I think a developer will probably not pull these debloated libraries out of the target directory and compile against them again. Maybe I missed the point here, but to me this feels unnecessary. If not, then this use case needs to be explained in more detail. Also, the number of 44 out of 1,001 applications was surprisingly high to me. Would this be linked to the low test coverage observed?”

We agree with the reviewer. The purpose of RQ5 is indeed to check for binary compatibility. In this research question, we check whether the client still compiles when using a debloated library as a dependency. Our objective is to quantify the impact of debloating a library for its clients. From the client’s perspective, missing used classes in a dependency causes errors at compilation time. For example, as shown in Table 6 in the revised version of the paper, the most common error is “Cannot find class.” Thus, we report these figures because it is critical for libraries to know the repercussions of debloating.

As pointed out by the reviewer, the lower number of compilation errors observed in the clients when using a debloated library may be related to the low test coverage of the clients and, in particular, in the libraries used as dependencies. Our debloating results are influenced by the coverage of the libraries and clients used as study subjects. We have added a sentence in Section 6.3.2 (External Validity) that refers to this point. A more in-depth investigation of the causes of low coverage between clients and their dependencies is out of the scope of this paper.

Comment #2.13: “line 1276: While valid, I think the “flakiness” of the tests could be checked in the dataset before the experiment is performed.”

We executed three times the test suite of all the library versions and all clients, as a sanity check to filter out libraries with flaky tests. We have explicitly referred to this in Section 4.2 of the paper (Data Collection):

Comment #2.14: “line 1281: I think also the version requirements could be checked before executing the implementation to the approach or before adding these projects to the dataset. It is a fair requirement.”

We thank the reviewer for this comment. We agree with the reviewer that adding Java version requirements in our data filtering is a fair experimental protocol. We did not check the Java version used by the debloated libraries in our dataset, but we checked that the library compiles successfully in our experimental environment before executing the debloating. For our experiments, we use Java version 1.8.

Comment #2.15: “The novelty of the work is a bit “oversold here”: I could 6 times “novel”, 9 times “first” in the sense of “we are the first” here. I can certainly see the novelty of the work even when this would be toned down a little.”

We thank the reviewer for this comment. We have lowered the tone and removed unnecessary novelty claims in the revised version of the paper.

Comment #2.16: “line 33: How do you identify and address useless code in this paper?”

We have noticed that there are subtle differences among different types of code bloat [2]. In this paper, consider *useless code* as code that can be removed from a software project without affecting its expected functionality. The removal of useless code in our paper is two-fold. First, we remove useless code in libraries based on the coverage information collected after executing their test suite. For the purposes of our experiments, we use the test suite as a proxy of the expected functionalities and consider non-tested code as useless. We proceed to remove code if it does not affect the compilation and test execution of the library. Second, we identify and quantify the amount of useless code in the clients of these libraries that exist as a result of declaring the library as a third-party dependency. Here, we analyze the syntactic correctness of the clients when using the debloated version of the library (i.e., if the clients compile), as well as the semantic correctness (i.e., if all the client’s tests pass).

Comment #2.17: “line 1258: I think the experiment itself is not novel.”

We have removed the word “novel” from the sentence. It is still important to notice that there is no previous work in the literature that has analyzed the impact of debloated libraries on their clients.

Reviewer #3

Major comments

Comment #3.1: “Treating different library versions as different libraries, the paper mixes different versions of libraries so the results may be biased for specific libraries with many included versions.”

We appreciate this concern of the reviewer regarding the soundness of our dataset. To describe the number of library versions, we have added a new row in Table 1. This row shows the distribution of library versions for the 94 unique libraries in our dataset. According to these descriptive statistics, the number of versions in our dataset is not unbalanced in favor of a set of specific libraries (1st Qu. = 1, Mean = 4.2, 3rd Qu. = 5.0, SD = 4.29). We performed a D’Agostino skewness test using the R `moments` package [6] to check whether the distribution of successfully debloated versions of libraries is skewed. The test results in a p-value = 0.3231, so we can conclude that the distribution of the number of successfully debloated versions is not statistically skewed towards a particular set of libraries (p-value > 0.05). Therefore, the debloating results are not biased as a consequence of one library having many versions with the same debloating result.

This can be observed more easily by plotting the debloating results of library versions per library. The stacked bar plot of Figure 1 shows this visualization. Each bar in the x-axis is one of the 94 libraries, sorted in decreasing order according to the number of versions. The y-axis is the number of versions of each library. The colors red in the legend represent versions that JDBL fails to debloat (either because no debloated JAR was produced or because not all the tests pass

when using the debloated JAR), whereas the green accounts for the successes. As we observe, the number of versions for which the debloating fails is scattered across many libraries, and the same occurs with the successes. Therefore, the debloating our results are not affected by the different library versions analyzed.

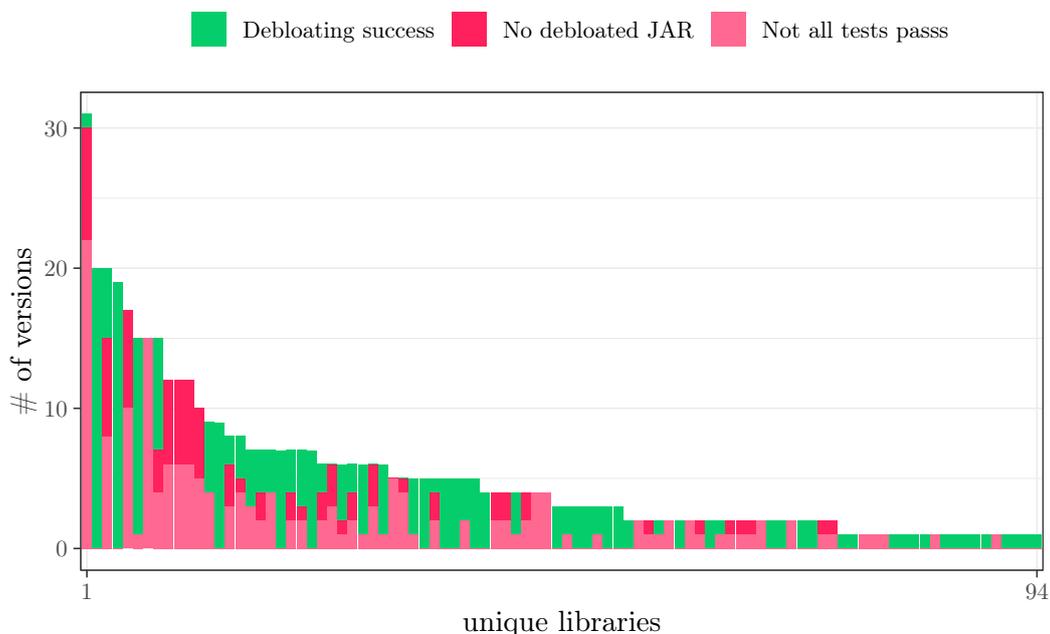


Figure 1: Debloating results of the versions of the 94 unique libraries used in our experiments.

Comment #3.2: “65% of the libraries where JDBL worked do not have dependencies. JDBL seems to work mostly on libraries w/o dependencies whereas from the introduction/motivation example it is supposed to be used to remove dependencies for an application. The paper is not clear about the percentage of libraries w/o dependencies to which JDBL was applied”

We have edited the motivating example in the figure presented in Section 2 to show more clearly that JDBL removes bloated code in the application as well as in its dependencies. The total number of distinct libraries with and without dependencies is 29 and 36, respectively.

Comment #3.3: “Related work: look into dead code elimination and compare your approach with historical approaches”

JDBL debloats Java bytecode dynamically using the runtime coverage collected when executing a workload. This means that, in contrast with only removing dead code, JDBL is able to debloat code that is reachable in the control flow from an entry point of an application code [5]. We have added a new research question (RQ5), comparing JDBL with JSRINK [1], which is the most recent published research tool for Java bytecode debloating.

Comment #3.4: “Listing 1 and L263,L269: exception in m2 is explicitly created by a throw statement, so listing says something different than explanation -> does the paper mean m1 is not shown as covered? If yes, the listing should highlight m1 instead of m2 to be consistent”

We thank the reviewer for this correct observation. Listing 1, which shows an example of an incomplete coverage report given by JaCoCo, was incorrect. We have checked the coverage report of JaCoCo by running this example and observed the coverage reported. The source code of all the examples presented in the listings is available: https://github.com/castor-software/jdbl-experiments/tree/master/revision/coverage_examples

Figure 2 shows a screenshot of HTML coverage report of JaCoCo corresponding to the code presented in Listing 1. In this example, the methods `m1` and `m2` are executed at runtime, but `m1` is not reported as covered (represented in red in the listing) because JaCoCo does not cover methods with implicit exceptions thrown from invoked methods [4]. We have fixed the listing and expanded the explanation of this example to facilitate its understanding for the reader. Furthermore, we have included two additional examples: Examples #4 and #5, to facilitate the reader’s comprehension of the challenges.

Foo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
<code>m1()</code>		0%		n/a	1	1	2	2	1	1
<code>m2()</code>		100%		n/a	0	1	0	1	0	1
<code>Foo()</code>		100%		n/a	0	1	0	1	0	1
Total	3 of 10	70%	0 of 0	n/a	1	3	2	4	1	3

Created with JaCoCo 0.8.5.201910111838

Figure 2: JaCoCo report with the overage results of the example presented in Listing 1.

Comment #3.5: “L339: Listing 3 describes const folding, so how does a list of dynamically loaded classes lead to errors described in Listing 3?”

We have expanded the description of the Example 3 in the revised paper. Figure 3 shows a screenshot of the coverage report of JaCoCo for the example presented in Listing 3. As we can observe from the figure, JaCoCo reports that the class `Foo` has 0% of code coverage. However, this class is compiled (i.e., a file `Foo.class` is created), and if we remove the bytecode file, then the compilation of the class `FooTest` will fail. We believe that this example exposes the existing mismatch between the Java source code and its compiled representation, at the same time that presents the limitations of debloating based solely on the output of code coverage tools.

Comment #3.6: “L370: Factual wrong: Also compile dependencies are packaged by Maven (<https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>) Runtime scope means, this dependencies are not needed to compile please clarify what you use”

We thank the reviewer for catching this glitch. JDBL debloats all the dependencies that end

Foo

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
• Foo()		0%		n/a	1	1	1	1	1	1
Total	3 of 3	0%	0 of 0	n/a	1	1	1	1	1	1

Created with JaCoCo 0.8.5.201910111838

Figure 3: JaCoCo report with the overage results of the example presented in Listing 3.

up in the fat JAR of the packaged Maven project. Therefore, we refer here to compile scope dependencies. We have fixed this statement in the text of the paper.

Comment #3.7: “L421: How are unused classes removed from the classpath (AFAIK one can only remove directories from the classpath), how is it verified, that a unused class does not appear somewhere else in the classpath?”

We have edited this sentence to make the explanation of the removal procedure more understandable. JDBL performs the debloating transformations during the project’s Maven build. It removes the unused classes just before the Maven **package** phase. The removal is performed in the file system with class-level granularity. This way, unused classes are not packaged by Maven after debloating; and for the packaged classes, all unused methods are removed.

Comment #3.8: “L432: How are bloated dependencies removed exactly? Again, how is it ensured, the removed dependency does not appear somewhere else in the classpath?”

JDBL unzips the JAR files of all the dependencies and removes all their unused classes. As with the project’s own classes, the bloated classes in the dependencies do not end up in the packaged project. A dependency is considered as bloated if all their unused classes were removed during this process. We have added more explanations of this bytecode removal process in Section 3.3.2.

Comment #3.9: “L485: It is important to be very explicit what Maven validates and what guarantees Maven gives; how does Maven check that no dependencies and especially other resources were incorrectly removed?”

During the build, Maven checks that the Java source code is syntactically correct through the compilation of the project. This requires checking that the bytecode in the dependencies is available and in the project modules and Maven plugins used during the build. Maven also checks the syntactic correctness of the *pom.xml*. Maven then uses the test suite to check for semantic correctness. If a class in the project or in its dependencies is incorrectly removed, then we observe that there are many different types of error that are logged by Maven, e.g., “Cannot find class,” or “UnsupportedOperationException.” We provide a complete descriptive and quantitative analysis of these errors in Table 6 of our paper.

Comment #3.10: “L530: Does RQ1 make sense? -> debloating tool which just removes all classes (100%) (still makes sense to report failures here, but the question itself seems 'useless') -> Rephrase/Combine with RQ2”

In RQ1, we assess to what extent can JDBL produce a debloated version of the Maven projects used in our study. This involves building the original project, and if the build passes, then we configure the project to perform the debloating using JDBL (see Figure 2). RQ1 evaluates the feasibility of debloating with this approach automatically, at scale, in real-world Java projects. To answer this question, we discuss the results and failures we encountered in producing the debloated JAR of the project. As far as we know, no previous work has assessed the ability of a debloating tool at producing a debloated JAR at the end of the build pipeline.

Comment #3.11: “L572: How many of the 147991 Java projects where Maven projects? How many multi-module projects where dropped at this step?”

We have computed the number of Java projects that used Maven from the initial dataset of 147,991 projects. We identified that 54,745 projects have at least one pom.xml file declared in their repository (34,560 single-module projects, 20,185 multi-module projects). We have added this information in the paper, in section 4.2.

Comment #3.12: “L586: Considering different versions of one library unique makes all results based on this unique libraries biased, as an example Table 2 shows entries where different (minor) versions of one library produce the same errors. I would suggest to group all data based on the 'unique' libraries used. For example averaging the results for each of the libraries before averaging the averages to mitigate biased results. Moreover please better specify the libraries in each step. The results would look less biased if for each major version of a library exactly one version would have been selected.”

We thank the reviewer for this comment. As we explained previously, when answering Comment #3.1, our results are not biased by the different library versions analyzed. The number of versions for which the debloating fails is scattered across distinct libraries, and the same occurs with the successes.

We study different versions of the same libraries in order to collect exact usage data from the clients (see RQ6 and RQ7). For example, if considering only one single version per library, then we could miss all the clients that use a different version. Furthermore, it is known that developers do not always stick to semantic versioning polices [9], which may cause significant differences among the versions.

Comment #3.13: “L610: 'no previous work uses test suite for debloating? -> please check for JSRINK usage of unit tests”

We thank the reviewer for this remark. We have removed this sentence in Section 4.3.

Comment #3.14: “L634: What is packaged in the jar file with all its dependencies -> L369 speaks about runtime dependencies”

We have fixed the error pointed by the reviewer (see the related Comment #3.6). We use the official `maven-assembly-plugin` with the `descriptorRef` set to `jar-with-dependencies` to package all the project class files and its dependencies [8]. This means that the bytecode of all the dependencies required for compilation will be added to the JAR.

Comment #3.15: “L706: Please provide insights how many really distinct libraries are there, and how many dependencies are there on average?”

The total number of distinct libraries (i.e., not considering the different versions) with dependencies presented in Figure 6 is 29. The total number of distinct libraries without dependencies is 36.

Comment #3.16: “L721: How do you check statically if a library is used in the source code?”

We implemented a utility tool that analyzed the source and search for class usages: <https://github.com/castor-software/jdbl-experiments/tree/master/script/extractClassUsage>. Our tool uses the integration of the Spoon library [7] with Maven (via the class `MavenLauncher`) to extract the types used by a client. If there is at least one type from the library used by the client, then we consider the library as *statically used* by the client. To the best of our knowledge, no previous work has validated the debloating considering the static usages in the source code of a debloated program. We have edited Section 4.3.4 to clarify this procedure.

Comment #3.17: “L722: How is the debloated library injected? Is there a chance, that a removed class from the debloated library can be found in another dependency?”

We replace the original JAR used by the client with the debloated JAR produced after debloating the library with JDBL. The library usage is assessed from the perspective of the client that uses the library. Therefore, if another third-party dependency uses the library as a dependency (i.e., becoming a transitive dependency of the client), it will end up in the client’s classpath only if it is actually used by the client.

Comment #3.18: “L752-L755: How can test succeed for RQ1 here and at the same time fail for RQ2? How was maven executed?”

In RQ1, we report the number of projects that were debloated with JDBL and for which Maven produced a debloated JAR at the end of the build pipeline. For our experiments, we configure Maven to continue its execution in case of a test failure. We then analyze the Maven logs outputs of the build and report the results in RQ1 and R2. Notice that the libraries that pass all the test in RQ2 are a subset of the 302 libraries that compile and are packaged correctly from RQ1.

Comment #3.19: “L781: Elaborate how JDBL alters the behavior of the project build besides changing the type to JAR”

We have added more details regarding the validation errors discussed in Section 5.1.1.

Comment #3.20: “L978: 99.1% debloating looks far too good to be true. Can you please elaborate and explain how such good numbers are created.”

Thank you for noticing this. We have investigated the results for this project in detail, and we found an omission in the coverage data. Consequently, we manually checked the results for all the projects that we reported as successfully debloated. We also ran a script that automatically checks the consistency of the coverage data. This thorough verification of our results revealed that only 1 out of 220 libraries misses coverage data. We also observed that 8 out of 220 libraries encountered issues during the validation phase. We updated the results to take these observations into account. We now have 302 libraries where we succeed to generate a debloated JAR (RQ1) and 211 that pass the test-suite after the debloating (RQ2). The impact on the global results is marginal, as evidenced by the updated figures and tables in the paper.

Comment #3.21: “L1126: How exactly does the static analysis work? False positives possible?”

We have edited the text in Section 4.3.4 and described how we used static analysis to check for library types used by the clients. Our approach does not permit false positives. False negatives are possible, but we believe that considering the size of our dataset, the results obtained are representative of real-world library-clients usages.

Comment #3.22: “Table 5. How is it ensured we have multiple separate errors vs one original error and then related 'follow' errors”

We manually counted and classified the errors in the 44 clients that do not compile using the debloated version of the library. Table 5 summarizes the errors that we found based on our analysis of the Maven logs. A client may be impacted by different errors. We decided to keep the total number of errors to reflect his fact and also to show the most common errors (e.g., “Cannot find the class” and “Package does not exist”). We believe that this information regarding the error that we encountered could be of interest to developers and practitioners using debloating in their codebases.

Comment #3.23: “L1298: External Validity: Is this representative for Java project after filtering for single POM and JUnit?”

We thank the reviewer for this remark. We believe that single POM (i.e., single-module) Maven projects using JUnit are representative of the Java ecosystem, but unfortunately, we do not have the data to support this claim. Therefore, we have slightly edited the second sentence in Section

6.2 (External Validity) to restrict our findings to the projects with these characteristics.

Minor comments

Comment #3.24: “Fig. 1 and L123: Using red/green colors is a barrier for some people”

We thank the reviewer for this comment. We have edited the description of Figure 1 and modified the legend. It now refers to the “green circles” and “red rectangles” so that visually impaired people can understand the figure.

Comment #3.25: “L176: A workload could in theory also exercise all of S_P , so that $F_P == S_P$ and $S_{P'} \leq S_P$, so I don’t get why the workload is defined to strictly exercise a subset of S_P ”

We thank the reviewer for spotting this mathematical imprecision. We have changed the expressions in the Definition 1 from $\mathcal{F}_P \subset \mathcal{S}_P$ to $\mathcal{F}_P \subseteq \mathcal{S}_P$, and from $|\mathcal{S}_{P'}| < |\mathcal{S}_P|$ to $|\mathcal{S}_{P'}| \leq |\mathcal{S}_P|$ to account for the particular case mentioned by the reviewer.

Comment #3.26: “L349, L353: First annotations are mentioned, than there is no discussion why they are not always kept like the other results”

Performing an in-depth discussion of Java annotations and how they compile to bytecode is out of the scope of our paper. Section 3.2.2 mentions that we do not remove annotations and other Java constructs such as interfaces, exceptions, and enumerations because these elements are necessary for compilation.

Comment #3.27: “L575: Why is there no check if the module type is Jar at the step, where we already check if JUnit is a dependency?”

Our dataset comprises Java libraries that contain public APIs and are designed to be reused by client applications via dependency declaration. The way Maven facilitates reuse is by packaging dependencies as JAR files. Maven considers the JAR format as the default for packaging Java applications (i.e., `<packaging>jar</packaging>` is present in the parent *pom.xml*). We checked that the packaging mode of all the libraries in our dataset is the default JAR format.

Comment #3.28: “L639: How exactly is the configuration of Surefire reseted, how do ensure this does not introduce conflicts? -> L670 if the number of tests change -> Is this enough”

We reset the configuration of the `maven-surefire-plugin` to its default parameters to execute the whole test suite and collect the complete coverage information of the libraries. This is also done to fairly compare the coverage-based debloating results obtained with all the libraries. This change could introduce some mismatches in the number of executed tests for the libraries with specific configurations.

Comment #3.29: “L629: Again, why was there no check for the package type, if the paper points out that this step might introduce conflicts?”

In this sentence, we refer to the conflicts that may when configuring Maven plugins. We rely on the `maven-assembly-plugin` with its default configuration to produce the fat JAR at the end of the Maven build pipeline. However, this plugin is highly configurable, and in some cases, the libraries may already use it with particular customizations. Our dataset includes libraries with very large build pipelines, which reflects the complexity of everyday software development. This may cause the build to break for different reasons in some situations. We present a detailed analysis of the build errors in our answers to RQ1 and RQ2.

Comment #3.30: “L665: ‘semantic preservation’ is a good term and I would like if it was used consistently during the text”

In RQ2 and RQ7, we refer to “preservation of the original behavior” of the libraries and clients, respectively. As explained in Section 4.3, preserving the original behavior means that all the original tests in the test suite used for debloating pass successfully:

“We consider that the test suite has the same behavior on both versions if the number of executed tests is the same for both versions, and if the number of passing tests is also the same.”

We have revised the terminology across the whole paper to make it consistent with this statement.

Comment #3.31: “L729: Is the set of dynamically used classes disjunct to the set classes directly referenced the source code?”

No, the set of dynamically used classes is not disjunct to the set of classes statically referenced in the source code. The union of both sets is the classes used.

Comment #3.32: “L923: Why are failing test cases accepted within JDBL, after we already verified, that all tests should work?”

Our experiments assess the challenges of coverage-based debloating for real-world Java applications when using the test suite as workload. To have a complete picture of these challenges, we configure JDBL so that it continues its execution with a different coverage tool in case of existing test failures. We consider a library to be successfully debloated only if all its tests pass. This is mentioned in the answer to RQ2, where we show that 69.9% of the debloated libraries preserve their original behavior.

Comment #3.33: “L1291: For me it would have been better to mention the Maven assembly plugin is used to build the fat jar -> L634”

We have edited the text to mention that we use the `maven-assembly-plugin` to build the fat JAR of the debloated libraries.

References

- [1] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 135–146, 2020.
- [2] César Soto-Valero. Unnecessary code. Author’s blog post, <https://www.cesarsotovalero.net/blog/unnecessary-code.html>, 2018.
- [3] Thomas Durieux, César Soto-Valero, and Benoit Baudry. DUETS: A Dataset of Reproducible Pairs of Java Library-Clients. In *Proceedings of the 2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, MSR’21, pages 545–549, New York, NY, USA, 2021. Association for Computing Machinery.
- [4] ECLEMMMA. Jacoco. Official Online Documentation, <https://www.eclemma.org/jacoco/trunk/doc/flow.html>, 2022.
- [5] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. Is static analysis able to identify unnecessary source code? *ACM Transactions on Software Engineering and Methodology*, 29(1), jan 2020.
- [6] Frederick Novomestky Lukasz Komsta. moments: Moments, cumulants, skewness, kurtosis and related tests. CRAN, <https://cran.r-project.org/web/packages/moments/index.html>, 2015.
- [7] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Software: Practice and Experience*, 46(9):1155–1179, 2016.
- [8] Apache Maven Project. Apache maven assembly plugin. Official Online Documentation, <https://maven.apache.org/plugins/maven-assembly-plugin/usage.html>, 2022.
- [9] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the maven repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224, 2014.