TSE-2023-02-0066: Major Revision

Dear Editorial Office of *IEEE Transactions on Software Engineering*,

We are happy to submit a major revision of our paper entitled **"Automatic Specialization of Third-Party Java Dependencies"**. We thank the reviewers for their recommendations. We have addressed all their comments, which has improved the quality of our manuscript. We have also consolidated the results in Table 2 after noticing an oversight in the computation of the number of classes removed during the specialization phase of DepTrim.

We have addressed the three major points emphasized by the associate editor as follows:

1. Scope: We have clarified the scope of the paper and its novelty in the Abstract, Introduction, and Related Work sections. DepTrim operates in two steps: 1) removing bloated dependencies, and 2) removing classes from non-bloated dependencies. The second step is completely novel (e.g., code analysis, code transformation, and packaging of specialized dependency `JAR` files). Our experiments with DepTrim on 30 projects are completely new.
2. Reliance on tests: We have refined the assurances provided by unit tests in the paper. We ran representative workloads on the original version and on the specialized version of the projects that have a CLI. The details of this new experiment are discussed in Section 6.2.
3. Comparative study: As far as we know, there is no other tool that specializes the dependencies of Java projects, packages the specialized dependencies and generates a new build file that uses the specialized dependencies. We have extended the comparison with the state-of-the-art in Section 7 in order to emphasize the key novelty of DepTrim w.r.t previous work.

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, Deepika Tiwari, Tim Toady, and Benoit Baudry

Reviewer #1

> **R1.1**: "My main issue with the paper stems from the "guarantees" that are made about the approach.Indeed, line 7 on page 2 best summarizes my issues with the paper "(i.e., the project correctly compiles, and all its tests pass, guaranteeing that the expected behavior of the project is unchanged)". This statement overstates the truth. Just because a project compiles and the tests pass does not in any way guarantee that the behaviour of the project is unchanged. Test suites can never prove the absence of bugs, and must generally concentrate on specific issues, since it is impossible to test everything. It is therefore possible that some dynamic features (that slip by DEPTRIM) are untested and make it into the final trimmed version without causing either a compilation error, or a test failure. Statements like these should be reworded."

We thank the reviewer for this comment. We acknowledge that relying solely on the project's test suite as a mechanism for confirming the preservation of functional integrity does not conclusively guarantee that no changes in project behavior have occurred after specialization. The inherent limitations in comprehensive testing, as mentioned by reviewer, render it unfeasible to evaluate every possible scenario. We mention this point in Section 6.3 as it is an internal threat to validity of our findings.

In response to this feedback, we have have carefully reviewed and edited the paper to clarify this nuance for the reader, thereby reinforcing the limits of our testing methodology and the implications for our results.

> **R1.2**: "For example: "With the creation of a partially specialized tree (PST), DEPTRIM effectively achieves dependency specialization without jeopardizing the success of the build, making it a practical option.", while the build passes, there is no proof that DEPTRIM does not jeopardize the usefulness of the application. This seems disingenuous to me, and more effort should be spent to explain that there are potentially many untested cases in this paper and not all causes of failures have been found."

We appreciate the reviewer's observation and we recognize the potential existence of untested scenarios and unaccounted sources of failure after specialization, which may indeed affect the project's behavior.

We have added clarifications in Section 6.1 of the paper to explain/clarify the limitations of our approach with respect to behavioral preservation and validation. This expanded explanation elucidates the potential limitations of our approach, particularly regarding the preservation of application behavior and the extent of validation. We have also added a completely new section 6.2 where we elaborate what additional validation step shall be carried our before deploying specialized dependencies in production. We have also evaluated the three applications in our dataset (projects that have a user facing interface) and observed that specialization does not jeopardize their key features.

R1.3: "Moreover, the fact that PST exist is proof that DEPTRIM can make mistakes. In 16/30 applications DEPTRIM produced a compilation error. This is of course discussed in RQ4. However the assumption is made that if a mistake occurs (e.g., because of dynamic loading), compilation will fail, or tests will fail, and then dependencies can be adjusted to reduce specialization. However, what happens if compilation doesn't fail, and tests are not sufficient to catch all issues? Indeed, 3 of the successful TST projects (immutables, scribejava, and tablesaw) had very few tests, hinting that they may not be well tested. This should at the very least be further discussed in the threats. Ideally, a second sample of applications, with known dynamic loading should be used to determine whether the causes of compilation fails that were detected in RQ4 are prevalent or not."

We thank the reviewer for this comment regarding the risk of removing necessary bytecode elements due to incomplete testing. We have added a discussion about the case of `immutables`, `scribejava`, and `tablesaw` in Section 6.4.

Empirical results from our previous work [7] indicate that detecting dynamic bytecode usage in libraries that leverage dynamic Java features poses a significant challenge for code removal transformations. We have made an effort to describe a number of common cases of failures in Section 5.4 (e.g., for dependencies `ommons-beanutils` and `commons-io`). To do so, we manually analyzed the tests' logs, as reported by the `maven-surefire-plugin`. We have discuss this in Section 6.1, in order to highlight the need of a more in-depth investigation to precisely determining to what extent the usage dynamic features could affect dependency specialization.

R1.4: "The fact that only a small percentage of tests fail when dynamic loading is used does not inspire confidence. This means that these features are not thoroughly tested, likely increasing the chance that DEPTRIM can make a mistake that will not be captured. It would be nice to have a warning of the kinds of tests that should exist in applications that will make use of DEPTRIM to prevent these mistakes. These could be inspired from the failing tests from RQ4. "

Determining whether a test makes use of dynamic features (e.g., reflection, dynamic proxies, or custom class loaders) can be quite challenging in Java. In particular, detecting which tests could trigger dynamic features in the code under test requires a deep understanding of Java's dynamic features and a careful analysis of the codebase of the project. The best tool that we known for this task is the GraalVM Tracing Agent [2], which is an advanced tool engineered to detect the utilization of dynamic features within a Java application [3]. When a Java application is run with this agent enabled, it monitors and records the application's usage of dynamic features and generates configuration files that describe this dynamic behavior. These generated files are useful for developers in situations where a priori knowledge of all dynamic behavior is not possible, such as when specializing dependencies which that could be accessed using dynamic features. By examining these files, developers can gain valuable insights into the application's dynamic behavior, enabling them to identify

the specific usage of dynamic features that the application is reliant upon. However, some dynamic behaviors are just too subtle or complex to detect easily, such as the effects of multi-threading, just-in-time (JIT) compilation, etc.

After thorough analysis, and due to the significant complexity associated with the detection of dynamic loading and other dynamic Java features, we believe it is good approach encouraging the DepTrim users to leverage the GraalVM Tracing Agent to detect the tests wich such dynamic behaviour. To facilitate this task, we have added necessary documentation about this limitation of DepTrim in its GitHub `README.md`, and we also suggest using the GraalVM Tracing Agent as a way to capture dynamic behaviors to improve the specialization process: https://github.com/ASSERT-KTH/deptrim/blob/main/README.md#known-limitations.

> **R1.5**: "Indeed it is likely that developers currently do not spend much time testing whether various parts of their dependencies are correctly dynamically loaded, because dependencies weren't expected to change until DEPTRIM came along. It would therefore be good to give empirical guidelines for preventative measures (i.e., how to make useful tests to prevent DEPTRIM from making mistakes)."

We agree with the reviewer, as testing dependencies is not a common practice in the software development workflow. This is because dependencies are generally treated as reliable external components and thus, developers often focus their testing efforts on assessing the functionality of the code they write, rather than the dependencies their code relies upon. Therefore, we cannot expect having a large number of tests in the project covering external functionality when specializing dependencies.

In our experiments, we use the tests for post-validation only. Recap that DepTrim relies on pure static analysis for code analysis. Consequently, there is no way to prevent mistakes due to dynamic Java features used at runtime. However, as previously mentioned by the reviewer and agreed by the author in comment R1.4, it is feasible using tools that alert developers when encountering dynamic features used by the project, as a preventative measure to avoid over-specialization.

> **R1.6**: "While I understand that DEPTRIM reduces dependency size by trimming classes from dependencies, and it is therefore natural to discuss how many classes are removed, a discussion about "classes" is not very satisfying from a usability perspective. Developers generally do not discuss the size of their deliverable in number of classes. Generally we discuss application size in bytes, or sometimes lines of code as a proxy. Some classes can be very small, some can be very large. A discussion about classes obfuscates a useful measure for deployment, namely the true size of the deployed application. Indeed, one of the stated benefits of DEPTRIM is "Smaller binaries reduce overhead when the JAR files are deployed and shipped over the network". Therefore, it would be nice to have a discussion about the binary size reduction (in bytes) to determine the effective impact of DEPTRIM on the size of applications."

The original Maven build pipeline of the 30 projects under study do not systematically produce one single *JAR* file that includes the project and all its dependencies. This is why we rely on sizes contributed by the bytecode within the dependency JARs rather than the size of the fat JAR in order to assess the ability of DepTrim to reduce the size of third-party code. We have added a new table, Table 4, in Section 5.3 to summarize these results. We have provided the distribution of the number of classes in compile-scope dependencies (CD), as well as the distribution of the size of the bytecode for these dependencies. Moreover, we provide the global reduction of the number of classes and the size of the bytecode for third-party dependencies.

> **R1.7**: "The paper does not currently have enough evidence to conclusively make the following statement: "Specialized dependencies reduce the build time, which increases productivity and reduces maintenance efforts". There is some overhead to doing static analysis, which is not fully measured in the paper (some rough time usage is indeed presented). What is the difference between the savings in build time and the time overhead of running DEPTRIM in the first place? Is that actually a net saving? Further evidence is needed for this statement."

We agree with the reviewer on this point. Our experiments did not encompass measuring build performance. Consequently, we have removed this statement from text.

> **R1.8**: "The exact effect of DEPTRIM on attack vectors is not fully clear. Indeed, the number of potential unsafe classes is reduced. However if the classes were not in use in the first place, were they really problems? Most current attacks appear to be supply chain attacks where correct dependencies are substituted for harmful ones. By making "custom" dependencies, DEPTRIM seems like it would actually make it more difficult to determine if a dependency is correct or harmful in the first place. It would be nice to have some more discussion on why removing unused classes is worth doing from a security perspective (either by citing related works that proves this, or through example use-cases)."

We thank the reviewer for this insightful comment. The impact of DepTrim on security, especially concerning dependency-related attack vectors, indeed merits further analysis. DepTrim primary role is to trim unused classes in dependencies, thereby reducing the overall attack surface. While it is true that classes that are not in use in the first place may not inherently pose a security concern, they may become problematic in certain scenarios [9]. For instance, an unused class could inadvertently become invoked due to changes in the codebase, or potentially be exploited by an attacker who gains access to the application's runtime environment [1]. As expressed by Ponta *et al.* [5]: "Even if some dependency code is not reachable when included in a given application (and thus it can be considered dead code in that context), it can still contribute to extending the attack surface of that application, e.g., because it

includes gadget classes leading to deserial-ization vulnerabilities[1]." In this context, unused classes could be considered as latent vulnerabilities.

In regards to supply chain attacks in which legitimate dependencies are replaced with malicious ones, the output of DepTrim indeed represents custom dependencies. However, the specialization process performed in our experiments is deterministic and repeatable: given the same codebase and set of dependencies, DepTrim will yield the same reduced set of dependencies. Thus, potential discrepancies can be identified through comparison with the expected output.

Still, the reviewer's point about the increased difficulty in determining the authenticity of a dependency is valid, and we acknowledge this as a potential trade-off for using DepTrim. We have added a discussion on this trade-off and the broader context of dependency verification in the Discussion section titled "Specialization and Software Integrity" of the revised paper. We concur that more evidence supporting the removal of unused classes from a security perspective should be provided. We have enhanced our discussion by referencing existing work that underscores the security benefits of minimizing the attack surface in the Related work section.

> **R1.9**: "A concerning issue arises in RQ1: some trimmed dependencies are "dependency license statements and build-related metadata". In some cases it may be illegal to ship a dependency without the appropriate license statement. What guarantees can DEPTRIM make with respect to not removing such licenses? At the very least this should be discussed as something that requires careful attention."

DepTrim only removes unused class files.[2] Therefore, license files are not removed during specialization. This improves the compliance of specialized artefacts regarding licensing, with a negligible impact on bytecode size reduction.

Our implementation of DepTrim prevents removing the text files named `LICENSE` or `LICENSE.txt` located in the root directory of the project. However, we notice that using these files is a convention and not a strict rule, and different projects may use different files alternatives. Often projects include only licenses as a header text statement in the source code. Also, as mentioned before, these license files may not be included in the built Maven artifacts unless explicitly configured to do so.

We also want to mention that Maven artifacts themselves do not necessarily include license files. The Maven Central repository has guidelines requiring projects to specify license information, although this does not guarantee the license file is included in every artifact. There is, however, a Maven plugin called the Maven License Plugin [6], which can be used to add license files to the built artifacts. Therefore, while Maven artifacts don't automatically include license files, they can be configured to include them. The adoption of these kind of tools would greatly facilitate avoiding the removal of licenses during program transformation tasks, as with DepTrim.

[1] https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data

[2] See https://github.com/ASSERT-KTH/deptrim/blob/dfb538d09a05f1126d1af72963d9940e0ca6f5a3/src/main/java/se/kth/deptrim/core/Specializer.java#L109-L123

Reviewer #2

> **R2.1**: "Running the test suite of a project is a good first step, and indeed is a critical component of the tool, but I'm curious as to the experience of clients of a project that uses DepTrim. For example, say we have a project P whose dependencies were specialized by DepTrim, yielding project P'. I think it worthwhile to find clients of P, configure them to depend on P' rather than P, and run their tests to see if using P' over P introduced any failing tests. (If failing tests were introduced, this could be indicative that clients use a project in ways not covered by the project's own test suite.) Perhaps this could be investigated in the context of RQ4, as an additional validation of the approach."

Running the complete build script of each project, including their test suite, is an important step to validate the specialization of third-party dependencies. Putting projects with specialized dependency trees in production can require additional validation. We discuss this in a new section (Section 6.2), where we distinguish between projects which have an interface, and projects that are solely used as libraries. As suggested by yourself, extra validation for the latter case would consist in curating a list of clients for the specialized library, which have a test suite that exercises this library. However, finding clients of libraries which 1) build and 2) have a test suite that exercises the third-party library is extremely challenging. This practice called reverse dependency compatibility testing is currently emerging, and is based on a careful selection of relevant clients by the developers of the library.

On the other hand, it is possible to deploy and exercise specialized projects that offer an interface, either graphical or command-line. Our dataset includes three such projects. For each of them, we have designed a representative workload that we ran on the original version and on the specialized version of the project. In all three cases, we managed to run typical scenarios on the specialized projects. The details of this new experiment are discussed in Section 6.2.

> **R2.2**: "Would it be possible to obtain the code coverage of the tests for all of the applications considered in the evaluation? Since the test suites are an integral part of the validation process, knowing how well (or how poorly!) they cover the code would be enlightening."

We have now updated Table 1 in the paper to include the test coverage of the 30 study subjects. The test coverage ranges from 18% to 93%. The median coverage is 64%.

> **R2.3**: "Would it be possible to reword RQ1 to something like "How well does bloated dependency removal work in real-world projects?" The current wording, mentioning dependency specialization, had me expecting to see DepTrim evaluated in RQ1, and not RQ2."

We thank the reviewer for this valid suggestion. We have reworded RQ1 accordingly, which now is as follows:

RQ1. *What is the impact of removing bloated dependencies on reducing the ratio of third-party code in real-world projects?*

> **R2.4**: "Could you report the code size in addition to the number of classes? E.g., report the size of bytecode before and after specialization? This would help contextualize the effectiveness of the approach."

We have reported the code size as well as the code size reduction by DepTrim in a new table (Table 4) in Section 5.3. This new table summarizes the key metrics about the impact of DepTrim on dependency trees. We provide the distribution of the number of classes in compile-scope dependencies (CD), as well as the distribution of the size of the bytecode for these dependencies. Additionally, we provide the global reduction of the number of classes and the size of the bytecode for third-party dependencies. Our findings are that DepTrim reduces the size of the third-party bytecode by 35.7%.

Reviewer #3

> **R3.1**: "Trimming down the dependencies of a software package (especially in the Java/Maven context) is a recently extensively worked on field. Usually these approaches use the term "debloating" to describe their process (pretty accurately actually). Here an approach for "specialization" is presented. This is curious as the authors have themselves published multiple approaches under the label "debloating" themselves. So the most obvious (and possibly easy to fix) shortcoming of the current manuscript is a lack of definition in the introduction what "specialization" is and how is it distinguished from tree shaking, debloating, and other terminology in the field. Later in the manuscript (Definition 4 and 5) the authors provide such a definition, but I found it to be very unclear. Especially Definition 4 reads like a definition of a debloated dependency and appears like a distinction without a difference. That very definition being the distinguishing factor the manuscript builds its case upon makes me question the actual contribution (and thus novelty) here."

We thank the reviewer for this valuable comment. We understand the confusion that may arise due to the seemingly overlapping terminology, as these terms are sometimes used interchangeably within the field. As such, we appreciate the opportunity to clarify the distinction between dependency debloating and dependency specialization.

Dependency debloating, as the reviewer rightly mentioned, involves trimming down the size of the packaged software by removing unnecessary or unused parts of the project's code and its included dependencies. This process is often driven by a general aim to reduce software bloat and does not necessarily take into account the specific usage of a dependency by a particular project under certain conditions.

Dependency specialization, on the other hand, is a more targeted process. While it shares the debloating's goal of reducing the size of dependencies, it does so with a specific project's usage in mind. In other words, a dependency is specialized with respect to a project when all the classes within that dependency are used by the project, and all unused classes have been identified and removed. In this case, DepTrim creates specialize variants of the specialized dependencies and distinct dependency trees for which the specialization is achievable. Given a project, DepTrim creates a specialized set of dependencies for which there is no class in the API of a specialized dependency that is unused, directly or indirectly, by the project or any other dependency in its dependency tree.

In essence, while debloating is more about size reduction in a broader sense, specialization is about size reduction tailored to the specific needs of a project. The two processes are certainly related, but they have different emphases and may therefore produce different results.

To make this distinction more evident, we have consolidated the definition of of dependency specialization in Section 3.1. We have revised the manuscript to clarify these definitions and distinctions in the introduction section to avoid confusion between dependency debloating and dependency specialization. We are believe that these revisions have enhanced the clarity and comprehensibility of the unique contributions and novel aspects presented in this paper.

> **R3.2**: "My second critique here is that all the heavy lifting (debloating) is done using the authors own prior work so I am uncertain that the offset from prior work justifies a different publication here. Being uneasy with that I gave the manuscript multiple re-reads to find the actual offset. It seems that the only difference seems to be the approach to only partially debloat/specialize dependencies based on their compilability with the project. To me this seems to be more an issue in unsoundness in the debloating and thus a workaround for the actual problem. I hope the authors can correct me on this one and it is a misunderstanding. Other parts of the proposed DepTrim approach also seem to be preexisting components. (See Section 3.3)."

The main novelty of our paper is not in the technique to detect and remove bloated code *per se*, but rather in the proposed output: "specialized variants of dependency trees." Indeed, we use existing software debloating techniques, including part of our prior work [8], to perform the actual detection of unused classes through static bytecode analysis. However, the key contribution of this paper is the unique way in which we apply those techniques: instead of removing the whole unused dependencies (as in our prior work), we do so in a manner that is sensitive to the project's specific usage of each dependency type, which we refer to as "dependency specialization." The decision to only partially specialize dependencies based on their compilability is not merely a workaround for the problems of debloating. Rather, it is an integral part of our approach to dependency specialization, which aims to create a more efficient, lean, and tailored dependency set that meets the specific needs of a project while maintaining its functionality.

In Section 3.3, while we do employ existing components, they are utilized in a

novel way that differentiates DepTrim from DepClean (our previous debloating tool). Therefore, it is the particular combination and application of these components in the context of our broader specialization approach that forms the innovative contribution of this work.

To make the difference more evident, we have added a new table in the revised version of the paper (Table 5) showing a comparison between the outputs of existing Java debloating tools and DepTrim. The comparison is based on the Java debloating tools studied in [5]. The new table is described in a paragraph which we have added to Section 7 of the revised paper.

> **R3.3**: "I am also surprised the author do not discuss their other (very interesting) prior work in the related work section: César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2023. Coverage-Based Debloating for Java Bytecode. ACM Trans. Softw. Eng. Methodol. 32, 2, Article 38 (April 2023), 34 pages. https://doi.org/10.1145/3546948 It would be interesting how DepTrim would change if it was built with JDBL instead of DepClean."

We have included JDBL into the new comparative table added, which responds to comment R3.2. Note that JDBL yields a singular specialized JAR file, while DepTrim outputs numerous specialized JAR files, each corresponding to a specialized `pom.xml` file. We have clarified this distinction in the appropriate paragraph within the Related Work section.

> **R3.4**: "The evaluation overall looks well-prepared, but again I was struggling with the scope of the manuscript in order to assess this correctly. In the definition of the study subjects I have to remark on the second dataset the authors use and procure for this manuscript that stars are indeed a proxy for popularity of a project on GitHub, but (as shown before) not a good selection criteria as for representative software projects. Coming back to the general framing of the manuscript, I also would have expected a comparative evaluation with debloating approaches to better understand the improvement made here, but no such comparison is made at the moment."

We thank the reviewer for this comment. These are valid points regarding the selection criteria of our second dataset and the need for a comparative evaluation with existing debloating approaches. Regarding the selection of our study subjects, we agree that GitHub stars are an imperfect measure of project representativeness. Note that, initially, we do not use popularity and stars to select representative study subjects; instead we rely on a set of representative projects previously curated by Durieux *et al.* [4]. However, as we mention in Section 4.1 (Study Subjects), this dataset is not enough for our research purposes. The reason is that our selection criteria entails not only stars, but also other constraints that filter out a large number of GitHub projects (e.g., we need Maven projects with at least one compile scope dependency, that we were able to build, which have a deterministic test suite, etc.). We want to mention

that to reach the 30 projects used in this study, we attempted to built tens of thousands of GitHub projects when preparing our experiments. We have edited Section 4.1 to make our methodology for selecting study subjects more clear.

As for the comparison with other debloating techniques, we understand the importance of such an evaluation to better illustrate the improvements brought by our approach. While our main focus was to introduce and assess the concept of "specialization" in this work, we agree that adding a comparative dimension would enrich our evaluation. As indicated in R3.2, we have provided a comparative table as well as additional discussion in Section 7.

Minor Comments

> **R3.5**: "Figure 2 implies that DepTrim builds a call graph, but I understood that DepClean does that. Please clarify and if DepTrim computes a call graph what precision does it have?"

DepTrim builds a static call graph of all used types in the project and its direct and transitive dependencies. As mentioned in section 3.3 (Implementation Details), this call graph reuses the core functionalities from DepClean. Measuring the precision of the call graph is out of the scope of this paper.

> **R3.6**: "p.5, left column, last paragraph: What does unsafe mean here? / p.10, left column, last paragraph: Why are they discarded as "unsafe"?"

In the context of our work, the term "unsafe" refers to those specialized dependencies that cause the build to break when included in the depenency tree, while "safe" dependencies are those that do not cause any issues during the build process. We have revised our manuscript and changed these terms in order to avoid any potential confusion.

> **R3.7**: "p.12, right column, second full paragraph: "130 do not pass [...] a few test cases" Are both referring to the same amount here? Is the second reference a subset of the first?"

Yes, only 130 out of $27,844$ unique tests executed across all projects fail. Table 4 shows the number of unique failing tests in the 9 projects with at least one test failure.

> **R3.8**: "p.12, second bullet point, right column: I understand that the test fails because DepTrim removes the class, but the interesting question is why does it do that?"

We agree with the reviewer regarding this point. We have made an effort in Section 5.4 to provide technical answer to this question. Investigating the test failures in this

context was an arduous task, as it required an understanding of intricate interactions between test, dependencies, and the impacts of DepTrim bytecode removal. As described, we performed an analysis of the project logs through manual examination. Three primary causes were identified, each adding a unique layer of complexity to the investigation.

The first identified cause was the dynamic loading of dependency classes, where tests failed when DepTrim removed classes loaded via reflection. The second cause was related to Java serialization, where the necessary classes for closing input streams were removed, causing failures. Third, dependencies using the Java Native Interface (JNI) for executing machine code at runtime were also removed by DepTrim, leading to test failures.

As mentioned in the paper, each identified cause highlighted the complexities involved in understanding these failures. From dynamically loaded classes to unresolved Java serialization and native code execution, every failure required a deep dive into the specifics of each of the 9 projects affected. Our rigorous investigation underscores the substantial effort we invested to provide a plausible explanation for these unexpected test failures.

R3.9: " p.13, Sec 6.1: I do not agree that it is likely that Java will embrace a full closed-world constraint and invite the authors to strengthen their argument here as currently the manuscript does not provide sufficient evidence for that."

We understand that this assertion may appear speculative without a more comprehensive justification. Our intention is not predicting a future direction for Java, but rather suggesting AOT as a potential path that could be beneficial for addressing the issues with bloated Java applications. Given the increasing complexity and size of software projects, it is conceivable that further mechanisms might be incorporated to control the scope of dependencies and reduce the potential for unused or unnecessary code.

Nonetheless, we agree that our argument could be better articulated to reflect the speculative nature of this point, and we have revised the text accordingly. We have included more context on the challenges and trade-offs associated with implementing such a constraint in Section 6.1, making clear that it is one of several possible solutions, and its adoption would depend on a variety of factors within the broader Java ecosystem.

R3.10: "p.14, Sec 6.2: Please explain why rehashing or re-signing could/couldn't be done here."

We have added two sentences in the paper to explain the issues with rehashing specialized dependencies.

R3.11: "p.15, Sec 8, last paragraph: Please have a look at the following approach presented at ICSE'23. It seems to go into the direction you sketched out here: https://conf.researchr.org/details/icse-2023/icse-2023-technical-track/72/UpCy-Safely-Updating-Outdated-Dependencies"

We thank the reviewer for bringing this recent and highly relevant paper to our attention. We have included a reference to this work in Section 8 of our revised paper.

References

[1] Pietro Borrello, Emilio Coppa, Daniele Cono D'Elia, and Camil Demetrescu. The rop needle: Hiding trigger-based injection vectors via code reuse. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1962–1970, 2019.

[2] GraalVM Compiler. Assisted configuration with tracing agent, Jun 2023. URL: https://www.graalvm.org/22.0/reference-manual/native-image/Agent/.

[3] GraalVM Compiler. Dynamic features of java, Jun 2023. URL: https://www.graalvm.org/latest/reference-manual/native-image/dynamic-features/.

[4] Thomas Durieux, César Soto-Valero, and Benoit Baudry. Duets: A dataset of reproducible pairs of java library-clients. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, pages 545–549. IEEE, 2021.

[5] Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *Proc. of ICSME*, pages 555–558. IEEE, 2021.

[6] Maven Central Repository. License maven plugin, Jun 2023. URL: https://mvnrepository.com/artifact/org.codehaus.mojo/license-maven-plugin.

[7] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. Coverage-based debloating for java bytecode. *ACM Transactions on Software Engineering and Methodology*, 32(2):1–34, 2023.

[8] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering*, 26(3):45, 2021.

[9] Renjun Ye, Liang Liu, Simin Hu, Fangzhou Zhu, Jingxiu Yang, and Feng Wang. Jslim: Reducing the known vulnerabilities of javascript application by debloating. In *International Symposium on Emerging Information Security and Applications*, pages 128–143. Springer, 2021.