WebJShrink: A Web Service for Debloating Java Bytecode

Konner Macias
University of California, Los Angeles
U.S.A
konnermacias@ucla.edu

Mihir Mathur
University of California, Los Angeles
U.S.A
mihirmathur@cs.ucla.edu

Bobby R. Bruce
University of California, Davis
U.S.A
bbruce@ucdavis.edu

Tianyi Zhang
Harvard University
U.S.A
tianyi@seas.harvard.edu

Miryung Kim
University of California, Los Angeles
U.S.A
miryung@cs.ucla.edu

ABSTRACT

As software projects grow in complexity, they come packaged with under-utilized libraries and therefore become *bloated*. Though several software debloating tools exist, none of them help developers gain insights into how under-utilized those libraries are nor help developers build confidence in the behavior preservation of software after debloating. To bridge this gap, we developed WebJShrink, a visual analytics tool for analyzing and pruning bloated software projects. WebJShrink is built on JShrink which uses static and dynamic reachability analysis to determine the extent of software bloat. WebJShrink provides rich visualizations of the bloat lurking within a target project's internal structure. It then removes unused features, and returns a safer, slimmer variant of the software project. To illustrate the target project's behavior preservation, WebJShrink examines the debloated software with its JUnit tests and visualizes the test results. In evaluating WebJShrink against 26 real world systems, we found WebJShrink could reduce software size by up to 42%, 11% on average, while still passing 100% of unit tests after debloating. We provide a video demonstrating WebJShrink at https://youtu.be/yzVzcd-MJ1w.

CCS CONCEPTS

• **Software and its engineering → Automatic programming**; *Object oriented languages.*

KEYWORDS

Java bytecode reduction, debloating, JShrink, software optimization

ACM Reference Format:
Konner Macias, Mihir Mathur, Bobby R. Bruce, Tianyi Zhang, and Miryung Kim. 2020. WebJShrink: A Web Service for Debloating Java Bytecode. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 5 pages.
https://doi.org/10.1145/3368089.3417934

1 INTRODUCTION

With the size and capability of software projects having grown tremendously over recent decades, software bloat such as unused features encumbers large projects. Modern object-oriented projects are especially bloated due to an excessive use of indirection, abstraction, and ease of extendibility [9]. Bloated software poses a direct security threat as unused software components increase security attack surfaces. For instance, Java applications utilizing the popular Apache Commons Collections library, prior to version 4.1, are vulnerable to deserialization attacks, that may result in the execution of arbitrary code. Most applications using this Apache Commons library do not require the unsafe classes related to deserialization attacks; thus, such unnecessary classes should be removed from the imported library to prevent this form of attack. Furthermore, size reduction offers additional benefits such as reduced download times, reduced loading times, and faster serialization time in big data systems such as Apache Spark.

Existing debloating tools such as ProGuard [1] and Jax [10] can automatically detect and remove unused code via static reachability analysis. However, these existing tools have several limitations. Firstly, they do not help developers gain insights into the extent of unused code through interactive visualizations. Secondly, while dynamic language features such as lambda expressions, dynamic proxy, reflection, etc. are prevalent in modern Java applications, these existing tools do not handle dynamic language features as they rely exclusively on static analysis. Thus, they may remove dynamically invoked code and subsequently induce unexpected behavior in debloated software. Lastly, they do not check behavior preservation of debloated software by running existing tests.

This paper presents a user-friendly visual analytics tool called WebJShrink that helps developers gain insights into the extent of software bloat and build trust about the safety of debloated software by running regression tests. WebJShrink is designed to be easy to use for developers and run as a software-as-a-service via a web browser without needing any downloads or installations. WebJShrink builds upon JShrink [3] and extends it with an intuitive GUI to provide rich, interactive visualizations of debloating statistics and behavior preservation. Furthermore, while prior debloating work uses static reachability analysis, JShrink combines pure static analysis with dynamic profiling to account for dynamic language features.

After performing reachability analysis, WebJShrink delivers interactive visualizations detailing used and unused classes and methods sorted by library APIs. WebJShrink then presents a debloating menu letting the user choose (1) how they want unused methods removed: (2) whether application methods should be pruned as opposed to library methods only, (3) whether JShrink's checkpointing should be enabled, etc. Though JShrink currently supports four different kinds of debloating transformations [10], its empirical evaluation finds that *unused method removal* is the most effective, achieving the majority of size reduction with minimal impact on behavioral preservation. Other transformations such as *class hierarchy collapsing* only contribute marginal size reduction with high risks of breaking software functionality [3]. Therefore, WebJShrink focuses on visualizing and analyzing the result of removing unused methods in the web interface. Checkpointing reverts the target application to checkpoints in case the debloating transformation leads to any test failure.

Once debloating concludes, WebJShrink automatically runs the freshly debloated software against its Maven test cases to gather behavior preservation statistics. WebJShrink shows the percentage of successful unit tests and size reduction statistics to the user using a progress-bar visualization and data table, respectively, along with a compressed ZIP file of the debloated project. At this point, the user can visibly witness the concrete existence of bloat lurking within their software, and how the minimized project performs against all tests previously set against it. WebJShrink offers this entire debloat process in a streamlined manner, with all intensive computation offloaded to the cloud, allowing developers to analyze and safely debloat their project repository without any local system requirements. We have made the code of WebJShrink available to public at https://doi.org/10.6084/m9.figshare.12518474

2 MOTIVATING EXAMPLE AND TOOL USAGE

Suppose Alice is a developer shipping dieforfree/qart4j[1], an image to QR code ASCII art generator, and needs to reduce the size of her project prior to release. First, she wishes to analyze the existence and severity of bloat within her Java code base. She visits the WebJShrink website, and is prompted for the project's GitHub identifier and to list relevant call graph analysis options, as shown in Figure 2. She must specify the project's entry points: `main` for all main methods (when the project is an executable application), `public` for all public methods (if the project is a library), and `test` for all JUnit test methods. Any combination of these three are permitted, and are used to determine method reachability via call graph analysis. Alice may also select whether the JShrink library uses either CHA [5] or Spark [7] algorithm to generate the call graph (Spark is more computationally intensive but generates a more exact, smaller call graph). Lastly, she indicates whether to utilize dynamic analysis, in addition to a purely static approach.

Upon completion, Alice is presented visualizations to aid her understanding of what code is, and is not, used. This is shown in Figure 1. WebJShrink displays two pie charts, providing an overview of bloat in terms of the ratio of used and unused Java classes (① in Figure 1) and methods (③ in Figure 1). Moving the cursor over any pie chart slice reflects the actual count of used or unused methods/classes (② in Figure 1).

WebJShrink further breaks down this data by each of the project's library dependencies via a horizontal bar chart (④ in Figure 1). From this, Alice may click on each library and discover which specific classes, within that library, are used or not (⑤ and ⑥ in Figure 1). From this single screen, Alice is presented a clear summary of the extraneous code in her repository and is now better educated for choosing to debloat her project prior to release.

Alice is ready to debloat her application, and proceeds to selecting the debloating options, as shown in ⑦ of Figure 1. The options presented are to "Prune App", which debloats both the application and library code, as opposed to library code only, "use Checkpointing" which reverts any debloat transformations which result in test failures; and whether to remove the entire methods including their header, or to remove just their body, or to leave an exception message in place of the removed method. Removing just the method body may be safer in the case where a call graph is not complete due to the existence of methods executed dynamically. Missing methods result in `MethodNotFoundExceptions`, whereas blank methods will not.

Once the options for debloating are selected, Alice selects "Debloat this Repository" and triggers the unused method removal procedure in JShrink. Once complete, Alice is presented size reduction and test behavior preservation statistics visualized through a before-and-after data table and a unit test success progress bar, respectively (as shown in Figure 3). Alice can now witness the reality of bloat lurking within her project, and how her project has been become minimized in terms of byte size as a result of JShrink. Further, Alice can gain complete confidence in the correctness of this debloating process in viewing how her debloated software performed against all unit tests previously ran against it. From this screen, Alice may download the resulting debloated project (debloated library dependencies included), packaged as a `ZIP` file, ready for delivery.

3 IMPLEMENTATION

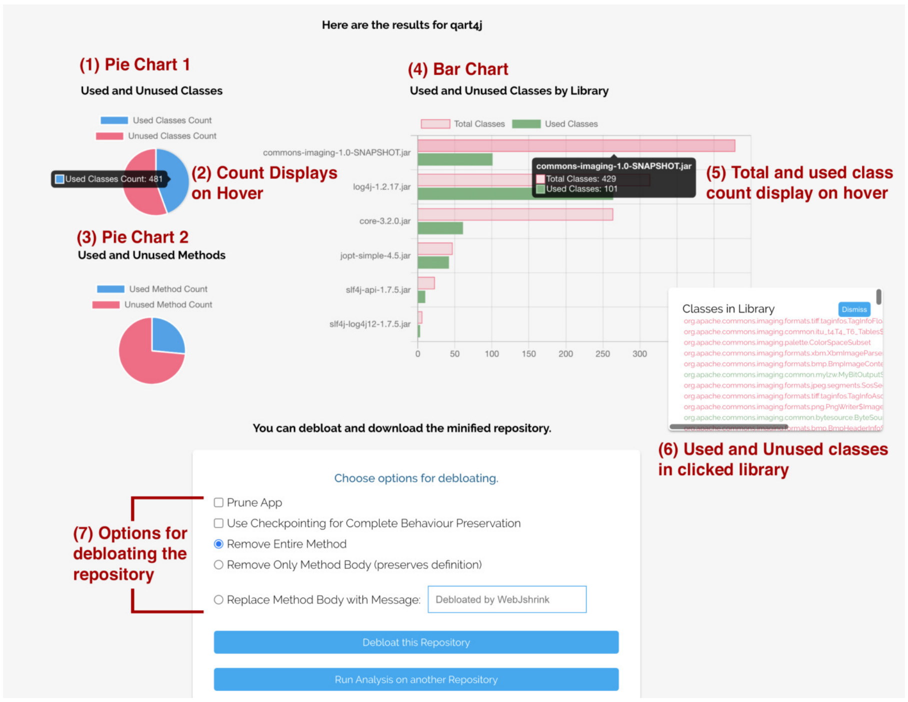
This section describes the implementation details of WebJShrink. WebJShrink is built using a client/server architecture; the client is a `React.js` single-page web application and the server is a Python application implemented using the Flask microframework. There are four main components of the application: (1) Preparation, (2) Call Graph Analysis, (3) Data Extraction and Visualization, (4) Debloating Transformation and Delivery.

Preparation. Given a GitHub repository and necessary input parameters, the WebJShrink server clones the repository and builds the project. To reduce the effort of building a project, we focus on Java projects built by the popular build system, Maven [8]. We leverage the Maven dependency plugin[2] to resolve all external library dependencies of a project, a necessary task in building an accurate call-graph. After a successful Maven build, the server executes JShrink to perform reachability analysis.

Call Graph Analysis. JShrink determines the reachability of methods and classes via call graph analysis. The entry points specified on WebJShrink's landing page are the starting points for

[1]https://github.com/dieforfree/qart4j

[2]https://maven.apache.org/plugins/maven-dependency-plugin/analyze-mojo.html


Here are the results for qart4j
(1) Pie Chart 1
Used and Unused Classes
Used Classes Count
Unused Classes Count
Used Classes Count: 481
(2) Count Displays on Hover
(3) Pie Chart 2
Used and Unused Methods
Used Method Count
Unused Method Count
(4) Bar Chart
Used and Unused Classes by Library
Total Classes
Used Classes
commons-imaging-1.0-SNAPSHOT.jar
log4j-1.2.17.jar
core-3.2.0.jar
jopt-simple-4.5.jar
slf4j-api-1.7.5.jar
slf4j-log4j12-1.7.5.jar
0
50
100
150
200
250
300
commons-imaging-1.0-SNAPSHOT.jar
Total Classes: 429
Used Classes: 101
(5) Total and used class count display on hover
Classes in Library
Dismiss
org.apache.commons.imaging.formats.tiff.taginfos.TagInfoFlo
org.apache.commons.imaging.common.itu_t4.T4_T6_Tables$
org.apache.commons.imaging.palette.ColorSpaceSubset
org.apache.commons.imaging.formats.xbm.XbmImageParser
org.apache.commons.imaging.formats.bmp.BmpImageConte
org.apache.commons.imaging.common.mylzw.MyBitOutput$
org.apache.commons.imaging.formats.jpeg.segments.SosSe
org.apache.commons.imaging.formats.tiff.taginfos.TagInfoAsc
org.apache.commons.imaging.formats.png.PngWriter$Image
org.apache.commons.imaging.common.bytesource.ByteSou
(6) Used and Unused classes in clicked library
You can debloat and download the minified repository.
(7) Options for debloating the repository

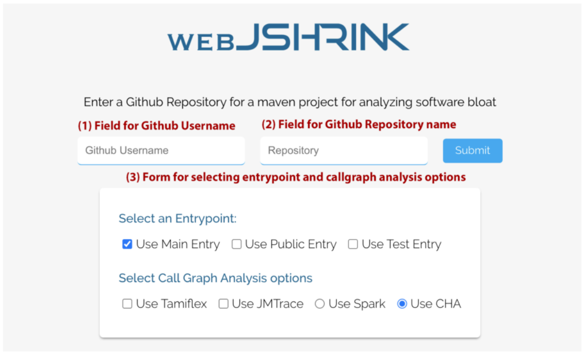
Choose options for debloating.

- ☐ Prune App
- ☐ Use Checkpointing for Complete Behaviour Preservation
- ● Remove Entire Method
- ○ Remove Only Method Body (preserves definition)
- ○ Replace Method Body with Message: Debloated by WebJshrink

Debloat this Repository

Run Analysis on another Repository

Figure 1: Visualizations showing used and unused classes, methods and classes by library for DIEFORFREE/QART4J.


WEBJSHRINK
(1) Field for Github Username
(2) Field for Github Repository name
(3) Form for selecting entrypoint and callgraph analysis options
Submit

Enter a Github Repository for a maven project for analyzing software bloat

Github Username | Repository

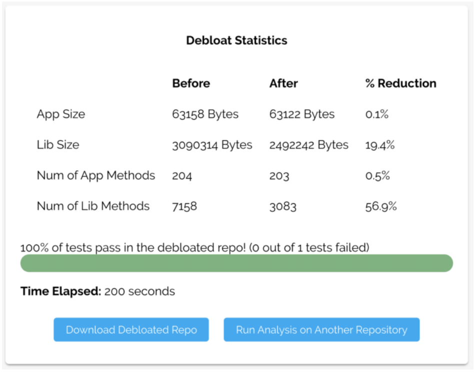
Select an Entrypoint:

☑ Use Main Entry ☐ Use Public Entry ☐ Use Test Entry

Select Call Graph Analysis options

☐ Use Tamiflex ☐ Use JMTrace ○ Use Spark ● Use CHA

Figure 2: Form for specifying repository and options for entry-point and call graph analysis.



Debloat Statistics

	Before	After	% Reduction
App Size	63158 Bytes	63122 Bytes	0.1%
Lib Size	3090314 Bytes	2492242 Bytes	19.4%
Num of App Methods	204	203	0.5%
Num of Lib Methods	7158	3083	56.9%

100% of tests pass in the debloated repo! (0 out of 1 tests failed)

Time Elapsed: 200 seconds

Figure 3: Code reduction statistics and test behavior examination results presented to the user after debloat completes.

generating a call graph such that all methods found to be reachable from these entry points are recorded within the graph. JSHRINK extends Soot [11], a bytecode analysis and modification framework to carry out this debloating bytecode transformation. Working at the bytecode level is necessary, as we debloat libraries which we cannot expect to be delivered to the user in any other form (Maven, for example, downloads dependencies as Java Jar archives).

Previous work, such as JRed [6], focused on debloating by creating call graphs in a completely static manner. However, the prevalent use of dynamic language features in Java such as reflection, ambiguous refraction, dynamic class loading, dynamic proxy, invokedynamic (lambda expression), JNI, and serialization makes debloating unsafe, as dynamically invoked code is not captured in static call graph analysis. To solve this problem, we design and implement our own dynamic profiling component, called JMTRACE. JMTRACE creates a log of the dynamically invoked call targets when running the target project’s JUnit test cases. We then use this log to extend our call graph; including methods that are accessed via reflective calls and, in turn, any methods that may thereby be reached from them. JMTRACE injects logging statements at the entry and exit of each method in a class during class loading. Similar to TamiFlex [2], JMTRACE instruments Java reflection call sites in bytecode to log which methods are invoked via reflection at runtime. In doing so, it obtains a log of all call targets executed via a test run.

We use the ASM bytecode manipulation and analysis library [4] to parse the target project's bytecode to develop a set of all classes and methods within the project. From this, we can quickly determine the set of methods and classes that are "unreachable" or "unused".

Data Extraction and Visualization. After running JShrink, the WebJShrink server creates a map of each class to a library, and labels the methods within each class as being either used or unused, as determined by JShrink. This map is then relayed to the browser for data visualization. Using the JavaScript library, `react-chartjs-2`[3], WebJShrink delivers pie charts of used and unused classes, along with an interactive horizontal bar chart detailing usage of each API in terms of class count.

Project Debloating and Delivery. JShrink eliminates unused methods by extending Soot's bytecode transformation APIs [11] to erase methods entirely, their method bodies, or by replacing their method bodies with an exception throw to indicate the removal of a method. While JShrink support four kinds of debloating transformations ('method removal', 'method inlining', 'field removal', and 'class hierarchy collapsing'), WebJShrink's visual interface focuses on 'method removal' as it is the most effective in terms of size reduction [3]. If 'checkpointing' is enabled in WebJShrink, test-failure inducing transformations are reversed to ensure debloating safety. Upon completion of debloating transformation, WebJShrink automatically runs the debloated project against its Maven unit tests and reports test behavior preservation statistics back to the user. A compressed deliverable (ZIP file) is then returned back to the user available for download. This compressed deliverable contains both the debloated application code and the debloated library dependencies.

Table 1: Code Size Reduction and Test Failures of 26 Popular GitHub Repositories

Application	Code Size Reduction	Test Failures	Tests
aragozin/jvm-tools	1.7%	0	102
bukkit/bukkit	15.4%	0	906
dieforfree/qart4j	42.2%	0	1
dubboclub/dubbokeeper	13.8%	0	1
eirslett/frontend-maven-plugin	18.7%	0	6
google/gson	0.3%	0	1050
JakeWharton/DiskLruCache	0.1%	0	61
JakeWharton/retrofit1-okhttp3-client	8.4%	0	9
JakeWharton/rxrelay	15.7%	0	58
JakeWharton/rxreplayingshare	20.1%	0	20
junit-team/junit4	1.7%	0	1081
kevinsawicki/http-request	0.2%	0	163
mabe02/lanterna	0.2%	0	34
notnoop/java-apns	13.8%	0	111
pagehelper/Mybatis-PageHelper	20.1%	0	106
pedrovgs/algorithms	0.0%	0	493
sockeqwe/fragmentargs	8.9%	0	15
square/moshi	0.2%	0	835
tomighty/tomighty	16.5%	0	26
zeroturnaround/zt-zip	5.4%	0	121
esoco/gwt-cal	16.5%	0	92
peter-lawrey/Java-Chronicle	0.0%	0	8
lehphyro/maven-config-processor-plugin	25.4%	0	77
jboss-logging/jboss-logmanager	11.1%	0	42
qiujiayu/AutoLoadCache	16.5%	0	11
alibaba/TProfiler	4.7%	0	3
Total	-	0	5432
Mean	11.0%	-	-
Median	9.9%	-	-

4 EVALUATION RESULTS

Our technical research paper on JShrink [3] presents comprehensive comparative evaluation against existing debloating tools, JRed [6], Jax [10], and ProGuard [1]. Here, we summarize our evaluation results briefly.

We debloated 26 popular GitHub Repositories as displayed in Table 1. For each experiment, we built the call graph by specifying main, public and test methods as entry points, used CHA and JMTrace as our call graph reachability analysis, pruned both application and library code, removed both the header and body of unused methods entirely, and utilized checkpointing. We found reductions of up to 42.2%, 11.0% on average, in bytecode size. Furthermore, we achieved complete software correctness modulo the target projects' JUnit test cases, with 100% passing successfully. This is a significant improvement over prior work. Jax and Proguard cause 3174 (58%) and 496 (9%) of tests to fail respectively, demonstrating the necessity of handling dynamic features and ensuring type safety.

5 CONCLUSION AND FUTURE WORK

In this paper, we introduce WebJShrink, a Java debloating visualization tool. WebJShrink shows the extent of bloat to be removed, allows the user to select available debloating options, and visualizes test behavior preservation before and after debloating transformation. WebJShrink runs as a software-as-a-service and provides an intuitive interface for the user to specify the target repository and to download the debloated software as a package, upon viewing debloating statistics. The evaluation of JShrink on 26 Java applications shows size reductions up to 42.2% (mean: 11.0%) in bytecode size is achievable with no degradation in software correctness—100% unit test success when running the debloated software against its Maven unit tests.

To our knowledge, WebJShrink is the first end-to-end debloating software tool and visualization service that handles dynamic language features safely and helps the user to examine behavior preservation using existing tests. Future work should focus on conducting user studies, and performing stress testing to ensure scalability across multiple concurrent users. We also see potential in integrating our visualizations into an IDE.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments. The participants of this research are in part supported by NSF grants CCF-1764077, CCF-1527923, CCF-1723773; ONR grant N00014-18-1-2037; an Intel CAPA grant; a Samsung grant; the Google PhD Fellowship program; and the Alexander von Humboldt Foundation.

[3]`react-chartjs-2` and is available from http://jerairrest.github.io/react-chartjs-2/

REFERENCES

[1] [n. d.]. ProGuard: Java and Android Apps Optimizer. https://www.guardsquare.com/en/products/proguard. ([n. d.]). Accessed: 2019-12-13.

[2] Eric Bodden, Andres Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. [n. d.]. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 2011 International Conference on Software Engineering — ICSE '11*. ACM, 241–250. https://doi.org/10.1145/1985793.1985827

[3] Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. JShrink: In-depth Investigation into Debloating modern Java Applications. In *Proceedings of the 2020 ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering — ESEC/FSE '20*. ACM. https://doi.org/10.1145/3368089.3409738

[4] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A Code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*.

[5] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 1995 European Conference on Object-Oriented Programming — ECOOP '95*. Springer, 77–101. https://doi.org/10.1007/3-540-49538-X_5

[6] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 2016 Computer Software and Applications Conference — COMPSAC '16*, Vol. 1. IEEE, 12–21. https://doi.org/10.1109/COMPSAC.2016.146

[7] Ondrej Lhotak. 2002. Spark: A flexible points-to analysis framework for Java.

[8] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.

[9] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2020. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *arXiv preprint arXiv:2001.07808* (2020).

[10] Frank Tip, Peter F Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. 2002. Practical extraction techniques for Java. *ACM Transactions on Programming Languages and Systems — TOPLAS '02* 24, 6 (2002), 625–666. https://doi.org/10.1145/586088.586090

[11] Raja Vallee-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Sundaresan Vijay. 1999. Soot — A Java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research — CASCON '99*. IBM Press, 13–23.