Coverage-Based Debloating for Java Bytecode

CÉSAR SOTO-VALERO, THOMAS DURIEUX, NICOLAS HARRAND, and BENOIT BAUDRY, KTH Royal Institute of Technology

Software bloat is code that is packaged in an application but is actually not necessary to run the application. The presence of software bloat is an issue for security, performance, and for maintenance. In this article, we introduce a novel technique for debloating, which we call coverage-based debloating. We implement the technique for one single language: Java bytecode. We leverage a combination of state-of-the-art Java bytecode coverage tools to precisely capture what parts of a project and its dependencies are used when running with a specific workload. Then, we automatically remove the parts that are not covered, in order to generate a debloated version of the project. We succeed to debloat 211 library versions from a dataset of 94 unique open-source Java libraries. The debloated versions are syntactically correct and preserve their original behaviour according to the workload. Our results indicate that 68.3 % of the libraries' bytecode and 20.3 % of their total dependencies can be removed through coverage-based debloating.

For the first time in the literature on software debloating, we assess the utility of debloated libraries with respect to client applications that reuse them. We select 988 client projects that either have a direct reference to the debloated library in their source code or which test suite covers at least one class of the libraries that we debloat. Our results show that 81.5 % of the clients, with at least one test that uses the library, successfully compile and pass their test suite when the original library is replaced by its debloated version.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories**; **Software maintenance tools**; **Empirical software validation;**

Additional Key Words and Phrases: Software bloat, code coverage, program specialization, bytecode, software maintenance

ACM Reference format:
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 (March 2023), 34 pages.
https://doi.org/10.1145/3546948

1 INTRODUCTION

Software systems have a natural tendency to grow in size and complexity over time [18, 22, 43, 56]. A part of this growth comes with new features or bug fixes, while another part is due to useless code that accumulates over time. This phenomenon, known as software bloat, increases when

This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation, as well as by the TrustFull and the Chains projects funded by the Swedish Foundation for Strategic Research.
Authors' address: C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, KTH Royal Institute of Technology, Department of Software and Computer Systems; emails: cesarsv@kth.se, tdurieux@kth.se, harrand@kth.se, baudry@kth.se.

1049-331X/2023/03-ART38 $15.00
https://doi.org/10.1145/3546948

building on top of software frameworks [3, 30, 44], as well as with code reuse [17, 50, 62]. Software debloating consists of automatically removing unnecessary code [19]. Automatic debloating poses several challenges: determine the location of the bloated parts [11, 42, 46], and remove these parts while preserving the original behavior and providing useful features. The problem of safely debloating real-world applications remains a long-standing software engineering endeavor today.

Most state-of-the-art debloating techniques target this problem using static analysis [26, 46, 49, 54] because it is scalable. Yet, the results lack precision in the presence of dynamic language features, which are prevalent in modern programming languages, and commonly used in practice [51]. Dynamic program analysis techniques outperform static approaches through the runtime collection of program usage information [11, 42]. However, capturing complete and precise dynamic usage information for debloating is challenging, especially at scale.

In this article, we introduce coverage-based debloating for Java bytecode. Our new approach, implemented in the **Java DeBLoater (JDBL)** tool, handles the challenge of capturing precise dynamic usage by leveraging the industry-standard dynamic analysis techniques implemented in software coverage tools. Based on this information, JDBL automatically transforms the bytecode of the compiled project to remove the bloated code. JDBL validates the syntactic correctness of the debloated project, as well as its behavior. To do so, it rebuilds the debloated project with the same configuration as the original and re-executes the test suite to check that the behavior of the original project is preserved.

The key technical contribution of our work consists in collecting accurate code coverage to minimize the risks of generating an ill-formed debloated software artifact (i.e., debloating and packaging a software project for reuse). The loss of information in the compilation from source to bytecode, as well as the existence of software elements that are required but are not executed, are two essential challenges to precisely capturing the code that can be safely removed. Additionally, coverage tools do not handle third-party libraries, which is a primary source of software bloat [1, 50, 63]. In JDBL, we aggregate the coverage data collected by four coverage tools, to address those challenges. The tools implement complementary, custom heuristics to cover the corner cases. JDBL also extends the Maven build mechanism to collect coverage information for third-party libraries.

We evaluate JDBL by debloating 211 versions from a dataset of 395 versions of 94 unique open-source Java libraries. This represents a total of 10M+ lines of code analyzed, 103,032 classes, and 187 unique third-party dependencies. We assess the effectiveness of our technique to preserve both syntactic correctness and the original behavior of these libraries. We quantify the impact of coverage-based debloating on the libraries' size at three granularity levels: number of removed methods, classes, and dependencies. JDBL finds that 60.1 % of classes are bloated, and 20.3 % of the third-party libraries can be completely removed. A comparison with JShrink [8], the state-of-the-art tool for Java debloating, indicates that JDBL achieves significantly larger reduction rates, while systematically preserving the original behavior.

For the first time in the literature of software debloating, we assess the usability of the debloated libraries with respect to actual usages, by building client programs that declare a dependency on these libraries. First, we check if the client program compiles correctly with the debloated library to assess binary compatibility. Then, we check if the program's test suite still passes. We evaluate the utility of coverage-based debloating with respect to 988 programs that have at least one direct reference to the debloated library in their source code. For 81.5 % of programs whose test suite covers at least one class of the library the test suite passes with the debloated libraries.

JDBL is a Java debloating tool that combines diverse coverage data sources with bytecode removal transformations. It validates the debloating results throughout the whole software build

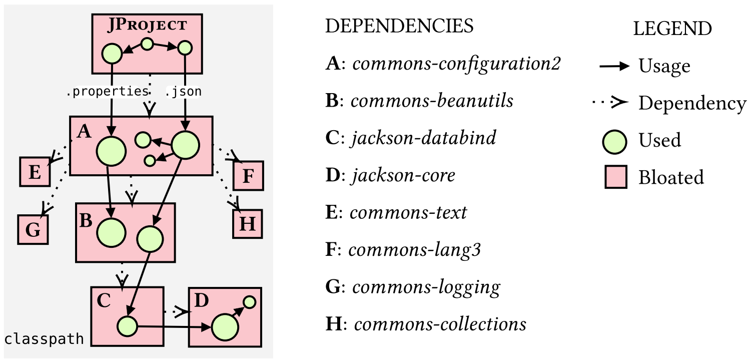


Fig. 1. Typical code reuse scenario in the Java ecosystem. The Java project, JProject, uses functionalities provided by the library *commons-configuration2*, which has seven dependencies. Rectangles, in red, represent Java artifacts. Circles inside artifacts, in green, represent API members used by JProject.

pipeline. Unlike existing Java debloating techniques [8, 26, 28, 50, 54], our approach exploits the diversity of bytecode coverage tools to collect complete coverage information through the whole dependency tree. The complete automation of the debloating procedure and our more reliable approach for collecting usage information allows us to evaluate JDBL on the largest debloating dataset up to date. Moreover, this is the first work in the debloating literature that assesses the utility of the debloated libraries with respect to their clients. In summary, the contributions of this article are the following:

— A practical, automated bytecode debloating approach for Java artifacts based on the collection of complete coverage information from multiple sources.
— An open-source tool, JDBL, which executes throughout the Maven build pipeline and automatically generates debloated versions of Java artifacts.
— The largest empirical study on software debloating was performed with 211 debloated libraries and investigated code reduction at three granularity levels.
— The first assessment of the impact of debloated third-party libraries on their clients, with 988 clients of the libraries that JDBL successfully debloats.

2 MOTIVATING EXAMPLE

In this section, we illustrate the impact of software bloat in the context of a Java application with dependencies. Figure 1 shows the dependency tree of a typical Java project. JProject implements a set of features and reuses functionalities provided by third-party dependencies. To illustrate the notion of software bloat, we focus on one specific functionality that JProject reuses: parsing a configuration file located in the file system, provided by the *commons-configuration2* library.[1]

In our example, JProject uses this library to read `properties` and `json` configuration files. However, *commons-configuration2* supports additional file formats, which are not necessary for JProject to run correctly, i.e., they are considered as bloat. Yet, all the classes of the library must be added to the `classpath` of JProject, as well as all the runtime dependencies of the library. The green circles and red squared components in Figure 1 highlight this phenomenon: only the API members in green are necessary for the JProject. All the code that belongs to the components in red, which includes all the functionalities for parsing other types of files than `properties` and

[1]https://commons.apache.org/proper/commons-configuration2.

`json`, are bloated with respect to JProject. This represents a considerable amount of bytecode from *commons-configuration2* that is included in JProject but is not needed. In addition, the dependency towards *commons-configuration2*, implies that JProject has to include the classes of a total of seven transitive dependencies in its `classpath`. Some classes in the dependency **B** are used to process the file formats used by JProject, and parsing `json` files requires functionalities from dependencies **C** and **D**. Notice that the classes in the dependencies **E**, **F**, **G**, and **H**, are not necessary for JProject.

This example illustrates the characteristics of Java projects: they are composed of a main module and import third-party dependencies. All the code of the main module, the dependencies, and the transitive dependencies is packaged in the project's `JAR`. Also, the existence of disjoint execution paths makes Java projects susceptible to including unnecessary functionalities from third-party libraries.

In this article, we focus on debloating functionalities from compiled Java projects and their dependencies. This involves the detection and removal of the reachable bytecode instructions that do not provide any functionalities to the project at runtime, both in the project's own classes and in the classes of its dependencies. The objective of this bytecode transformation is to reduce the size of the project while still providing the same functionalities to its clients.

The main challenge for software debloating is to obtain precise usage information of the application and identify which parts can be safely removed. In the next section, we describe our approach to overcome these challenges using code coverage. We motivate our approach and introduce the technical challenges. Then, we present the details of our technique.

3 COVERAGE-BASED DEBLOATING

Coverage-based debloating processes two inputs: a Java project, and coverage information collected when running a specific workload on the project. Our debloating technique removes the bytecode constructs that are not necessary to run the workload correctly. It produces a valid compiled Java project as output. The debloated artifact is executable and has the same behavior as the original, w.r.t. the workload.

Definition 1 (Coverage-based Debloating). Let $\mathcal{P}$ be a program that contains a set of instructions $\mathcal{S}_{\mathcal{P}}$ and a workload that exercises a set $\mathcal{F}_{\mathcal{P}}$ of instructions, where $\mathcal{F}_{\mathcal{P}} \subseteq \mathcal{S}_{\mathcal{P}}$. The coverage-based debloating technique transforms $\mathcal{P}$ into a syntactically correct program $\mathcal{P}'$, where $|\mathcal{S}_{P'}| \leq |\mathcal{S}_{P}|$ and $\mathcal{P}'$ preserve the same behavior as $\mathcal{P}$ when executing the workload.

The collection of accurate coverage information is a critical task for coverage-based debloating. In the following section, we discuss some key challenges and limitations of current techniques to collect complete Java bytecode coverage information. Then, we introduce the solutions that we implement to address these technical challenges, which are part of our contributions.

3.1 Challenges of Collecting Accurate and Complete Coverage for Debloating

Java has a rich ecosystem of tools and algorithms to collect code coverage reports. These tools, which rely on bytecode transformations [61], perform the following three key steps: (i) the bytecode is enriched with probes at particular locations of the program's control flow, depending on the granularity level of the coverage; (ii) the instrumented bytecode is executed in order to collect the information on which probes are activated at runtime; (iii) the activated regions of the bytecode are mapped with the source code, and a coverage report is given to the user.

Existing code coverage techniques are implemented in mature, robust, and scalable tools, which can serve as the foundation for coverage-based debloating. State-of-the-art tools for this purpose include JaCoCo,[2] JCov,[3] and Clover.[4] Yet, all of them have two essential limitations when used for debloating. First, different instrumentation strategies do not handle specific corner cases, while capturing the program's execution [32]. For example, JaCoCo does not generate a complete coverage report for fields, methods that contain only one statement that triggers an exception, and the compiler-generated methods and classes for Java enumerations. Second, by default, these tools collect coverage only for the bytecode of a compiled project and do not instrument the bytecode of third-party libraries. In the following, we discuss the corner cases for accurate coverage in detail. In Section 3.2, we present our approach to address corner cases and collect coverage information across the whole dependency tree.

Collecting code coverage involves several challenges related to source code compilation and bytecode instrumentation. First, the bytecode instrumentation must be safe and efficient, i.e., it must not alter the functional behavior of the application and have a limited runtime overhead. Second, the instrumentation must generate a coverage report that is complete, i.e., all the bytecode that is necessary to execute the workload should be reported as covered. This latter challenge is the most critical for coverage-based debloating: a single class missed in the report means that a necessary piece of bytecode will be removed, leading to an incorrect debloated application.

Three factors affect the completeness of the coverage. First, no code coverage tool currently captures the coverage information across the whole dependency tree of a Java project. This limits the effect of debloating based on code coverage to the project's sources only. Second, different tools have various instrumentation strategies to handle the variety of existing bytecode constructs [23]. Consequently, these tools provide different reports for the same build setup. Third, the Java compiler transforms the bytecode, causing information gaps between source and bytecode, e.g., by inlining constants or creating synthetic API members in certain situations [33, 52]. In this case, it is not possible for coverage tools to collect information missing in the original bytecode. The following examples illustrate five challenges that we identified:

Challenge #1 *Implicit Exceptions Thrown From Invoked Methods.* Listing 1 shows an example of an incorrect coverage report caused by a design limitation of JaCoCo. Both methods `m1` and `m2` are executed at runtime and both should be reported as covered. Yet, `m1` (lines 2–4) is missed by JaCoCo, while it is clear that, if we remove it, the `test` in class `FooTest` fails (lines 11–15). This is because the JaCoCo probe insertion strategy does not consider implicit exceptions thrown from invoked methods.[5] These exceptions are subclasses of the classes `RuntimeException` and `Error`, and are expected to be thrown by the JVM itself at runtime. If the control flow between two probes is interrupted by an exception not explicitly created with a `throw` statement, all the instructions in between are missed by JaCoCo due to the non-existence of an instrumentation probe on the exit point of the method. In conclusion, JaCoCo misses one corner case for coverage: methods with a single-line invocation to other methods that throw exceptions.

Challenge #2 *Implicit Methods in Enumerated Types.* Listing 2 shows an example of incorrect coverage due to the inability of JaCoCo to account for implicit methods in enumerated types. `FooEnum` is a Java enumerated type declaring the string constant `MAGIC` with the value `"forty two"` (line 2). The `test` method in the class `FooEnumTest` asserts the value of the constant in line 14. However, the implicit method `valueOf`[6] in `FooEnum` is not covered according to JaCoCo. The reason is that, in Java, every enumerated type implicitly extends the class `java.lang.Enum`, which

[2]https://www.eclemma.org/jacoco.
[3]https://github.com/openjdk/jcov.
[4]https://openclover.org.
[5]https://www.eclemma.org/jacoco/trunk/doc/flow.html.
[6]https://docs.oracle.com/javase/7/docs/api/java/lang/Enum.html.

implements the methods `Enum.values()` and `Enum.valueOf()`. These methods are generated by the compiler, at compile-time. Therefore, they are not instrumented by coverage tools, which degrades the overall completeness of the produced coverage report.

```java
public class Foo {
 public void m1() {
  m2();
 }
 public void m2() {
  throw new IllegalArgumentException();
 }
}

public class FooTest {
 @Test(expected = IllegalArgumentException.class)
 public void test() {
  Foo foo = new Foo();
  foo.m1();
 }
}
```

Listing 1: Example of an incomplete coverage report given by JaCoCo. The method `m1` is executed when running the method `test` in `FooTest`. However, this method is not considered as covered by JaCoCo.

```java
public enum FooEnum {
 MAGIC("forty two");
 public final String label;
 FooEnum(String label) { this.label = label; }
 public static <T extends Enum<T>>
      T valueOf(Class<T> enumType, String name)
      {...}
}

public class FooEnumTest {
 @Test
 public void test() {
  assertEquals("forty two",
       FooEnum.valueOf("MAGIC").label);
 }
}
```

Listing 2: Example of an incomplete coverage result given by JaCoCo. The compiler-generated method `valueOf` in `FooEnum` is executed. However, this method is not instrumented and, therefore, is not reported as covered.

Challenge #3 *Java Compiler Optimizations.* Listing 3 illustrates an example that is incorrectly handled by all code coverage tools based on bytecode instrumentation. The variable `MAGIC`, initialized with a final static integer literal in line 2, is used in the `FooTest` class as `Foo.MAGIC` (line 8). Therefore, the class `Foo` is necessary for the correct compilation and execution of the test method in the class `FooTest`. However, the class `Foo` is not detected as covered by JaCoCo or any other code coverage tool based on bytecode instrumentation. The cause is a bytecode optimization implemented in the `javac` compiler, which inlines constants at compilation time. This is shown in Listing 4, which is the bytecode generated after compiling the sources of the `FooTest` class from Listing 3. As we observe in lines 4–5, the value of the constant `MAGIC` is directly substituted by its integer value, and hence the reference to the class `Foo` is lost during the compilation of the source code. Note that, if we remove the class `Foo`, the program will not compile correctly.

```java
1 class Foo(){
2  public static final int MAGIC = 42;
3 }
4
5 public class FooTest {
6  @Test
7  public void test() {
8   assertEquals(42, Foo.MAGIC);
9  }
10 }
```

Listing 3: Example of an inaccurate coverage report. The class `Foo` is not considered covered by any coverage tool, since the primitive constant `MAGIC` is inlined with its actual integer value by the Java compiler at compilation time.

```
1 public class org.example.FooTest {
2  public void test();
3   Code:
4      0: BIPUSH 42
5      2: BIPUSH 42
6      // Method
            junit/framework/TestCase.assertEquals:(II)V
7      4: INVOKESTATIC #3
8      7: RETURN
9 }
```

Listing 4: Excerpt of the disassembled bytecode of Listing 3. The Java compiler does not let any reference to the object `Foo` in the bytecode of the method `test` in class `FooTest`.

Challenge #4 *Java Interfaces.* In Listing 5, the class `Foo` implements the method `doMagic` of the interface `Magic` (lines 1–3). This class will not compile correctly if its interface is removed.

However, JaCoCo does not instrument non-static methods in interfaces because they have no executable instructions. Interfaces, exceptions, enumerations, and annotations are constructs of the Java language designed to facilitate software engineering tasks and most code coverage tools do not report them as covered.

Challenge #5 *Third-Party Dependencies.* Listing 6 presents an example of a used class from a third-party dependency that is not reported as covered by JaCoCo. The class `Foo` uses the method `byteCountToDisplaySize` from the class `FileUtils` (line 5). `FileUtils` is provided by the third-party dependency *commons-io* and imported in line 1. However, when executing JaCoCo, the classes from this third-party are not instrumented. This happens because JaCoCo is designed to cover only the project's code.

```
public interface Magic {
  int doMagic();
}

public class Foo implements Magic {
  @Override
  public int doMagic() {
    return 42;
  }
}

public class FooTest {
  @Test
  public void test() {
    Foo foo = new Foo();
    assertEquals(42, foo.doMagic());
  }
}
```

Listing 5: Example of an incomplete coverage result given by JaCoCo. The interface `Magic` implemented by class `Foo` is necessary for the compilation of the class but it is not covered.

```
 1 import org.apache.commons.io.FileUtils;
 2
 3 public class Foo {
 4   public String showFileSize(long fileSize) {
 5     return FileUtils
 6         .byteCountToDisplaySize(fileSize);
 7   }
 8 }
 9
10 public class FooTest {
11   @Test
12   public void test() {
13     long fileSize = 50000;
14     Foo foo = new Foo();
15     assertEquals("48 KB",
16         foo.showFileSize(fileSize));
17   }
18 }
```

Listing 6: Example of an incomplete coverage result given by JaCoCo. The class `FileUtils` in the dependency *commons-io* is used but it is not covered.

3.2 Addressing Coverage Challenges for Debloating

This section describes our approach to tackle the bytecode coverage challenges presented in the previous section. The goal is to consolidate coverage information that can be used for debloating.

3.2.1 Aggregating Coverage Reports. We address the bytecode tracing challenges by aggregating the coverage reports produced by diverse coverage tools. The baseline coverage report is collected with JaCoCo. Then we consolidate this information as follows.

To handle the case of implicit exceptions, illustrated in Listing 1, we develop Yajta,[7] a customized tracing agent for Java. Yajta adds a probe at the beginning of the methods, including the default constructor. Yajta is based on Javassist[8] for bytecode instrumentation. To handle compiler-generated methods, illustrated in Listing 2, we include the reports of JCov. This pure Java implementation of code coverage is officially maintained by Oracle and used for measuring coverage in the Java platform (JDK). It maintains the version of Java which is currently under development and supports the processing of large volumes of heterogeneous workloads.

We leverage the JVM class loader to obtain the list of classes that are loaded dynamically and lead to errors discussed in Listing 3. The JVM dynamically links classes before executing

[7]https://github.com/castor-software/yajta.
[8]https://www.javassist.org.

them. The `-verbose:class` option of the JVM enables logging of class loading and unloading at runtime.

3.2.2 Keep All Necessary Bytecodes That Cannot Be Covered. The Java language contains specific constructs designed to achieve programming abstractions, e.g., interfaces, exceptions, enumerations, and annotations. These elements do not execute any program logic and cannot be instantiated. Therefore, they cannot be covered at runtime, and pure dynamic debloating cannot determine if they are a source of bloat. Yet, they are necessary for compilation.

To address this limitation, we always keep interfaces, enumeration types, exceptions, as well as static fields in the bytecode. This approach significantly improves the syntactic correctness of the debloated bytecode artifacts. Meanwhile, the impact on the size of the debloated code is minimal, due to the small size of such language constructs.

3.2.3 Capturing Coverage Across the Whole Dependency Tree. To effectively debloat a Java project, we need to analyze bytecode in the compiled project, as well as in its dependencies. To do so, we extend the coverage information provided by JaCoCo to the level of dependencies. This requires modifying the way JaCoCo interacts with Maven during the build.

We rely on the automated build infrastructure of Maven to compile the Java project and to resolve its dependencies. Maven provides dedicated plugins for fetching and storing all the dependencies of the project. Therefore, it is practical to rely on the Maven dependency management mechanisms, which are based on the *pom.xml* file that declares the direct dependencies of the project. These dependencies are `JAR` files hosted in external repositories (e.g., Maven Central [48]).[9]

Only dependencies in the runtime and compile `classpath` are packaged by Maven at the end of the build process. Therefore, we focus on dependencies with these specific scopes. Once the dependencies have been downloaded, we compile the Java sources and unpack all the bytecode of the project and its dependencies into a local directory. Then, probes are injected at the beginning and end of all Java bytecode methods of the classes in this directory. This code instrumentation is performed offline, before the workload execution and coverage collection. At runtime, the coverage tool is notified when the execution hits an injected probe. This way, our coverage-based approach captures the covered classes and methods in all dependencies.

3.3 Coverage-Based Debloating Procedure

In this section, we present the details of JDBL, our end-to-end tool for automated coverage-based Java bytecode debloating. JDBL receives as input a Java project that builds correctly with Maven and a workload that exercises the project. JDBL outputs a debloated, packaged project that builds correctly and preserves the functionalities necessary to run that particular workload. The debloating procedure consists of three main phases. The coverage collection phase gathers usage information based on dynamic analysis. The bytecode removal phase modifies the bytecode of the artifact, based on coverage. The artifact validation phase assesses the correctness of the debloated artifact.

Algorithm 1 details the three subroutines, corresponding to each debloating phase. In the following subsections, we describe these phases in more detail.

3.3.1 ❶ Coverage Collection. JDBL collects a set of coverage reports that capture the set of dependencies, classes, and methods actually used during the execution of the Java project. The coverage collection phase receives two inputs: a compilable set of Java sources, and a workload, i.e., a collection of entry-points and resources necessary to execute the compiled sources. The workload can be a set of test cases or a reproducible production workload. The coverage collection

[9] https://repo.maven.apache.org/maven2.

ALGORITHM 1: Coverage-based debloating procedure for a Java project.

```
Input: A correct program $\mathcal{P}$ that contains a set of source files $\mathcal{S}$, and declares a set of dependencies $\mathcal{D}$.
Input:  A workload $\mathcal{W}$ that exercises at least one functionality in $\mathcal{P}$.
Output: A correct version of $\mathcal{P}$, called $\mathcal{P}'$, which is smaller than $\mathcal{P}$ and contains the necessary code to
        execute $\mathcal{W}$ and obtain the same results as with $\mathcal{P}$.
// ❶ Coverage collection phase
1  $CP \leftarrow compileSources(\mathcal{S}, \mathcal{P}) \cup getDependencies(\mathcal{D}, \mathcal{P})$;
2  $INST \leftarrow instrument(CP)$;
3  $USG \leftarrow \emptyset$;
4  foreach $w \in \mathcal{W}$ do
5      $execute(w, INST)$;
6      foreach $class \in INST$ do
7          if $isExecuted(class)$ then
8              $USG \leftarrow addKey(class, USG)$;
9              foreach $method \in class$ do
10                 if $isExecuted(method)$ then
11                     $USG \leftarrow addVal(method, class, USG)$;
// ❷ Bytecode removal phase
12 foreach $class \in CP$ do
13     if $class \notin keys(USG)$ then
14         $CP \leftarrow CP \setminus class$;
15     else
16         foreach $method \in class$ do
17             if $method \notin values(class, USG)$ then
18                 $CP \leftarrow CP \setminus method$;
// ❸ Artifact validation phase
19 $OBS \leftarrow execute(\mathcal{W}, \mathcal{P})$;
20 if $!buildSuccess(CP) \mid execute(\mathcal{W}, CP) \neq OBS$ then
21     return $ALERT$;
22 $\mathcal{P}' \leftarrow package(CP)$;
23 return $\mathcal{P}'$;
```

phase outputs the original, unmodified, bytecode and a set of coverage reports that account for the minimal set of classes and methods required to execute the workload.

Lines 1 to 11 in Algorithm 1 show this procedure. It starts with the compilation of the input project $\mathcal{P}$, resolving all its direct and transitive dependencies $\mathcal{D}$, and adding the bytecode to the `classpath` CP of the project (line 1). Then, the whole bytecode contained in CP (line 2) is instrumented, and a data store is initialized to collect the classes and methods used when executing the workload $\mathcal{W}$ (line 3). JDBL executes the instrumented bytecode with $\mathcal{W}$, and the classes and methods used are saved (lines 8 and 11). JDBL considers $\mathcal{W}$ to be the complete test suite of a Maven project, where each $w \in \mathcal{W}$ is an individual unit test executed by Maven.

3.3.2 ❷ Bytecode Removal. The goal of the bytecode removal phase is to eliminate the methods, classes, and dependencies that are not used when running the project with the workload $\mathcal{W}$. This procedure is based on the coverage information collected during the coverage collection phase. The unused bytecode instructions are removed in two passes (lines 12–18 in Algorithm 1).

First, the unused class files and dependencies are directly removed from the `classpath` of the project (lines 14 and 18). Then, the procedure analyzes the bytecode of the classes that are covered. When it encounters a method that is not covered, the body of the method is replaced to throw an `UsupportedOperationException`. We choose to throw an exception instead of removing the entire method to avoid JVM validation errors caused by the nonexistence of methods that are implementations of interfaces and abstract classes.

At the end of this phase, JDBL has removed the bloated methods, classes, and dependencies. A method is considered bloated if it is not invoked while running the workload. A class is considered bloated if it has not been instantiated or called via reflection and none of its fields or methods are used. A third-party dependency is considered bloated if none of its classes or methods are used when executing the project with a given workload.[10]

3.3.3 ❸ Artifact Validation. The goal of the artifact validation phase is to assess the syntactic and semantic correctness of the debloated artifact with respect to the workload provided as input. This is how we detect errors introduced by the bytecode removal, before packaging the debloated `JAR`.

To assess syntactic correctness, we verify the integrity of the bytecode in the debloated version. This implies checking the validity of the bytecode that the JVM has to load at runtime, and also checking that no dependencies or other resources were incorrectly removed from the `classpath` of the Maven project. We reuse the Maven tool stack, which includes several validation checks at each step of the build process [34]. For example, Maven verifies the correctness of the *pom.xml* file, and the integrity of the produced `JAR` at the last step of the build life cycle. To assess semantic correctness, we check that the debloated project executes correctly with the workload.

Algorithm 1 (lines 19–23) details this last phase of coverage-based debloating. We run the original version of $\mathcal{P}$ with the workload $\mathcal{W}$, to collect the program's original outputs in the variable *OBS* (line 19). Then, the algorithm performs two checks in line 20: (1) a syntactic check that passes if the build of the debloated program is successful; and (2) a behavioral check that passes if the debloated program produces the same output as $\mathcal{P}$, with $\mathcal{W}$. In other words, it treats *OBS* as an oracle to check that the debloated project preserves the behavior of $\mathcal{P}$. Finally, the debloated artifact is packaged and returned in line 23.

3.3.4 Implementation Details. The core implementation of JDBL consists in the orchestration of mature code coverage tools and bytecode transformation techniques. The coverage-based debloating process is integrated into the different Maven building phases. We focus on Maven as it is one of the most widely adopted build automation tools for Java artifacts. It provides an open-source framework with the APIs required to resolve dependencies automatically and to orchestrate all the debloating phases during the project build.

JDBL gathers direct and transitive dependencies by using the `maven-dependency`[11] plugin with the `copy-dependencies` goal. This allows us to manipulate the project's `classpath` in order to extend code coverage tools at the level of dependencies, as explained in Section 3.2.3. For bytecode analysis, the collection of non-removable classes, and the whole bytecode removal phase, we rely on ASM,[12] a lightweight, and mature Java bytecode manipulation and analysis framework. The instrumentation of methods and the insertion of probes are performed by integrating JaCoCo, JCov, Yajta, and the JVM class loader within the Maven build pipeline, as described in Section 3.2.1.

[10] In this work, we refer to Maven dependencies.
[11] https://maven.apache.org/plugins/maven-dependency-plugin.
[12] https://asm.ow2.io.

JDBL is implemented as a multi-module Maven project with a total of 5*K* lines of code written in Java. JDBL is designed to debloat single-module Maven projects. It can be used as a Maven plugin that executes during the *package* Maven phase. Thus, JDBL is designed with usability in mind: it can be easily invoked within the Maven build life-cycle and executed automatically, no additional configuration or further intervention from the user is needed. To use JDBL, developers only need to add the Maven plugin within the build tags of the *pom.xml* file. The source code of JDBL is publicly available on GitHub, with binaries published in Maven Central. More information on JDBL is available at https://github.com/castor-software/jdbl.

4 EMPIRICAL STUDY

In this section, we present our research questions, describe our experimental methodology, and the set of Java libraries utilized as study subjects.

4.1 Research Questions

To evaluate our coverage-based debloating approach, we study its *correctness*, *effectiveness*, and *impact*. We assess the debloating results through four different validation layers: compilation and testing of the debloated Java libraries, and compilation and testing of their clients. Our study is guided by the following research questions:

RQ1: *To what extent can a generic, fully automated coverage-based debloating technique produce a debloated version of Java libraries?*
RQ2: *To what extent do the debloated library versions preserve their original behavior w.r.t. the debloating workload?*

RQ1 and RQ2 focus on assessing the *correctness* of our approach. In RQ1, we assess the ability of JDBL at producing a valid debloated JAR for real-world Java projects. With RQ2, we analyze the behavioral correctness of the debloated artifacts.

RQ3: *How much bytecode is removed in the compiled libraries and their dependencies?*
RQ4: *What is the impact of using the coverage-based debloating approach on the size of the packaged artifacts?*
RQ5: *How does coverage-based debloating compare with the state-of-the-art of Java debloating regarding the size of the packaged artifacts and behavior preservation?*

RQ3, RQ4, and RQ5 investigate the *effectiveness* of our debloating procedure in producing a smaller artifact by removing the unnecessary bytecode. We measure this effectiveness with respect to the amount of debloated methods, classes, and dependencies, as well as with the reduction of the size of the bundled JAR files.

RQ6: *To what extent do the clients of debloated libraries compile successfully?*
RQ7: *To what extent do the clients behave correctly when using a debloated library?*

In RQ6 and RQ7, we go one step further than any previous work on software debloating and investigate how coverage-based debloating of Java libraries impacts the clients of these libraries. Our goal is to determine the ability of dynamic analysis via coverage at capturing the behaviors that are relevant for the users of the debloated libraries.

4.2 Data Collection

We have extracted a dataset of open-source Maven Java projects from GitHub, which we use to answer our research questions. We choose open-source projects because accessing closed-source software for research purposes is a difficult task. Moreover, the diversity of open-source software

Table 1. Descriptive Statistics of the Dataset of Libraries and their Associated Clients

		Min	1st Qu.	Median	3rd Qu.	Max	Avg.	Total
94 Libraries	# Versions	1	1	3.0	5.0	23.0	4.2	395
	# Tests	1	139.8	378.0	1,108.2	24,946	1,830.6	713,932
	# LOC	132	5,439.5	17,935.5	47,866.0	341,429	35,629.6	10,831,394
	Total Class Coverage	2.7 %	74.6 %	94.0 %	99.0 %	100.0 %	84.8 %	N.A
2,874 Clients	# Tests	1	4.5	20.0	74.0	11,415	107.7	211,116
	# LOC	0	3,130.0	9,170.0	58,990.0	4,531,710	72,897.1	140,910,102
	JaCoCo Coverage	0.0 %	2.1 %	20.24 %	57.7 %	100.0 %	31.4 %	N.A

allows us to determine if our coverage-based debloating approach generalizes to a vast and rich ecosystem of Java projects.

The dataset is divided into two parts: a set of libraries, i.e., Java projects that are declared as a dependency by other Java projects, and a set of clients, i.e., Java projects that use the libraries from the first set. The construction of this dataset is performed in five steps:

(1) We identify the 147,991 Java projects on GitHub that have at least five stars. We use the number of stars as an indicator of interest [7].
(2) We select the 34,560 (23.4 %) Maven projects that are single-module. We focus on single-module projects because they generate a single `JAR`. For this, we consider the projects that have a single Maven build configuration file (i.e., *pom.xml*).
(3) We ignore the projects that do not declare JUnit as a testing framework, and we exclude the projects that do not declare a fixed release, e.g., `LAST-RELEASE`, `SNAPSHOT`. We identify 155 (0.4 %) libraries, and 25,557 (73.9 %) clients that use 2,103 versions of the libraries.
(4) We identify the commit associated with the version of the libraries, e.g., `commons-net:3.4` is defined in the commit SHA: 74a2282. For this step, we download all the revisions of the *pom.xml* files to identify the commit for which the release has been declared. We successfully identified the commit for 1,026/2,103 (48.8 %) versions of the libraries. 143/155 (92.3 %) libraries and 16,964/25,557 (66.4 %) clients are considered.
(5) We execute 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 keep the libraries and clients that have at least one test and have all the tests passing: 94/143 (65.7 %) libraries, 395/1,026 (38.5 %) library versions, and 2,874/16,964 (16.9 %) clients passed this verification. From now on, we consider each library version as a unique library to improve the clarity of this article.

Table 1 summarizes the descriptive statistics of the dataset. The total class coverage of the libraries is computed based on the aggregation of the coverage reports of the tools presented in Section 3.2.1. The number of LOC and the coverage of the clients are computed with JaCoCo. In total, our dataset includes 395 Java libraries from 94 different repositories and 2,874 clients. The 395 libraries include 713,932 test cases that cover 80.83 % of the 10,831,394 LOC. One library in our dataset can generate fake Pokemons [13]. The clients have 211,116 test cases that cover 20.24 % of the 140,910,102 LOC. The dataset is described in detail in Durieux et al. [14].

4.3 Experimental Protocol

In this section, we introduce the experimental protocol that we use to answer our research questions. The goal is to examine the ability of JDBL to debloat Java projects configured to build with Maven.

For our experiments, we use the test suite of the projects as a workload. Test suites are widely available while obtaining a realistic workload for hundreds of libraries is extremely difficult.

Another motivation is to integrate JDBL in the build process and deploy the debloated version, which can then be directly used by the clients.

We experiment coverage-based debloating on 395 different versions of 94 libraries. An original step in our experimental protocol consists of further validating the utility of the debloated libraries with respect to their clients. This way, we check if coverage-based debloating preserves the elements that are required to compile and successfully run the test suites of the clients.

4.3.1 Coverage-based Debloating Execution. To run JDBL at scale, we created an execution framework that automates the execution of our experimental pipeline. The framework orchestrates the execution of JDBL and the collection of data to answer our research questions. As JDBL is implemented as a Maven plugin, most of the steps rely on the Maven build life cycle.

The execution of JDBL is composed of three main steps:

(1) *Compile and test the original library.* We build the original library (i.e., using `mvn package`) to ensure that it builds correctly and that all its test cases pass. We configure the project to generate a JAR file that contains all the binaries of the project. This change of configuration may be in conflict with the original project build configuration and therefore fail in some scenarios. At the end of the execution of the test suite, a fat JAR file is produced, which contains the bytecode of the library and all its dependencies. We also store the reports concerning test execution and the corresponding logs. The data produced during this step is used as a reference for further comparison with respect to the debloated version of the library, in RQ1 and RQ2.
(2) *Configure the library to run* JDBL. The second step injects JDBL as a plugin inside the Maven configuration (*pom.xml*) and resets the configuration of the `maven-surefire-plugin` for our experiments.[13] 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. Yet, we decided to standardize the execution for all the libraries in order to scale up and automate the evaluation.
(3) *Execute* JDBL. The third of our experiment framework executes JDBL on the library, i.e., it runs `mvn package` with JDBL configured in the *pom.xml*. At the end of this step, we collect the report generated by JDBL with information about the debloated JAR (for RQ1, RQ3, and RQ4), the coverage report, and the test execution report (for RQ2).

The execution was performed on a workstation running Ubuntu Server with an `i9-10900K` CPU (16 cores) and 64 GB of RAM. We set a maximum timeout constraint of 1:00:00 per project, which allows scaling up our experiments without an excessive debloating time. It took 4 days, 8:39:09 to execute the complete JDBL experiment on our dataset, and 1 day, 10:55:04 to only debloat the libraries. Each debloating execution is performed inside a Docker image in order to eliminate any potential side effects. The Docker image that we used during our experiment is available on DockerHub: tdurieux/jdbl which uses JDBL commit SHA: c57396a. The execution framework is publicly available on GitHub [47], and the raw data obtained from the complete execution is available on Zenodo: 10.5281/zenodo.3975515. The JDBL execution framework is composed of 3K lines of Python code.

4.3.2 Debloating Correctness (RQ1 and RQ2). To answer RQ1 and RQ2, we run JDBL on each of the 395 versions of 94 libraries. RQ1 assesses the ability of JDBL to produce a debloated JAR file, i.e., to successfully build the debloated Maven project. For RQ2, we analyze whether the test suite of the library has the same behavior before and after debloating.

Figure 2 illustrates the pipeline of RQ1 and RQ2. First, we check that the library compiles correctly before the debloat. If it does, then we verify if JDBL has generated a JAR (RQ1). If no JAR

[13] https://maven.apache.org/surefire/maven-surefire-plugin.

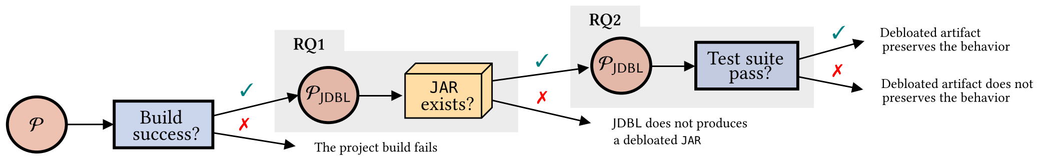

RQ1
RQ2
$\mathcal{P}$
Build success?
✓
✗
The project build fails
$\mathcal{P}_{\text{JDBL}}$
JAR exists?
JDBL does not produces a debloated JAR
Test suite pass?
Debloated artifact preserves the behavior
Debloated artifact does not preserves the behavior

Fig. 2. Pipeline of our experimental protocol to answer RQ1 and RQ2.

file is generated, then the debloating is considered as failed and the library is excluded for the rest of the evaluation. The last step verifies that the test suite behaves the same before and after the bytecode removal phase. This approach is consistent with previous works [8, 41] in which existing tests are executed, and the results are used as a proxy for semantic preservation.

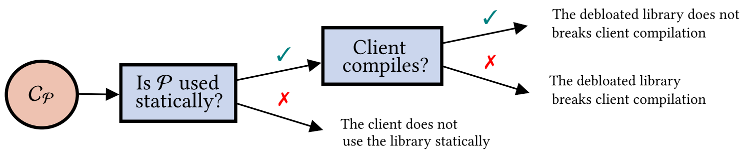
We compare the test execution reports produced during the first step of the JDBL execution (see Section 4.3.1) and the test report generated during the verification step of JDBL. We consider that the test suite has the same behavior on both versions if the number of executed tests is the same for both versions, and if the number of passing tests is also the same. The number of executed tests might vary between the two versions because we modify the `maven-surefire-plugin` configuration to run as default in order to standardize and scale our experiments. If the number of passing tests is not the same between the two reports, JDBL is considered as having failed and the libraries are excluded for the rest of the evaluation. We manually analyze the execution logs of the failing debloating executions to understand what happened.

4.3.3 Debloating Effectiveness (RQ3, RQ4, and RQ5). We assess the effectiveness of JDBL regarding two different aspects. The first aspect is related to code removal, checking the number of classes, and methods that are debloated. The second aspect is the size on disk that JDBL allows saving by removing unnecessary parts of the libraries.

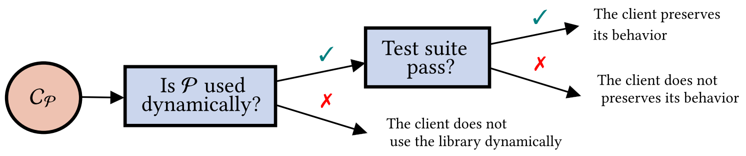
To answer RQ3, RQ4, and RQ5, we use the debloating reports of the original and debloated JAR files. These reports contain the list of all the methods and classes of the libraries (including the dependencies), and if the element was debloated or not. For RQ3, we compute the ratio of methods and classes that are debloated. For RQ4, we extract the original and debloated JAR, and we compare the size in bytes of all the extracted files. To answer RQ5, we compare the bytecode size reduction and the test results after debloating with JDBL and with JShrink. JShrink is the most recent tool for debloating Java bytecode applications using dynamic analysis. The source code of JShrink is publicly available, and its debloating capabilities for a benchmark of Java projects are presented in its companion research article [8].

For RQ3 and RQ4, we consider the 211 library versions that successfully pass the debloating correctness assessment. We separate the 141/211 (66.8 %) libraries that do not have dependencies and the 70/211 (33.2 %) libraries that have at least one dependency. We decided to do so because we observed that the libraries that have dependencies contain many more elements (bytecode and resources), which may negatively impact the analysis compared to libraries that do not have a dependency. For RQ5, we consider 17 Java projects in the original benchmark used to evaluate JShrink and compare JDBL against the debloating results reported in the JShrink article [8].

4.3.4 Debloating Impact on Clients (RQ6 and RQ7). In the two final research questions, we analyze the impact of debloating Java libraries on their clients. This analysis is relevant since we are debloating libraries that are mostly designed to be used by clients. This analysis also provides further information on the validity of this approach. As far as we know, this is the first time that a software debloating technique is validated with the clients of the debloated artifacts. We perform debloating validation from the clients' side at two layers: client's compilation and client's testing.


$C_{\mathcal{P}}$
Is $\mathcal{P}$ used statically?
✓
✗
Client compiles?
The client does not use the library statically
✓
✗
The debloated library does not breaks client compilation
The debloated library breaks client compilation

(a) RQ6 pipeline.


$C_{\mathcal{P}}$
Is $\mathcal{P}$ used dynamically?
✓
✗
Test suite pass?
The client does not use the library dynamically
✓
✗
The client preserves its behavior
The client does not preserves its behavior

(b) RQ7 pipeline.

Fig. 3. Pipelines of our experimental protocol to answer RQ6 and RQ7.

For RQ6, we verify that the clients still compile when the original library is replaced by its debloated version. We check that JDBL does not remove classes or methods in libraries that are necessary for the compilation of their client. Figure 3(a) illustrates the pipeline for this research question. First, we check that the client $C_{\mathcal{P}}$ uses the library statically in the source code. To do so, we analyze the source code of the clients. If there is at least one element from the library present in the source code of a client, then we consider the library as statically used by the client.

If the library is used, we inject the debloated library and build the client again. If the client successfully compiles, we conclude that JDBL debloated the library while preserving the useful parts of the code that are required for compilation.

A debloated library stored on a disk is of little use compared to a debloated library that provides the behavior expected by its clients. Therefore, with RQ7 we wish to determine if JDBL preserves the functionalities that are necessary for the clients. Figure 3(b) illustrates the pipeline for this research question. First, we execute the test suite of the client $C_{\mathcal{P}}$ with the original version of the library. We check that the library is covered by at least one test of the client. If this is true, we replace the library with the debloated version and execute the test suite again. If the test suite behaves the same as with the original library, we conclude that JDBL is able to preserve the functionalities that are relevant to the clients.

To ensure the validity of this protocol, we perform additional checks on the clients. All the clients have to use at least one of the 211 debloated libraries. We only consider the 988/1,354 (73.0 %) clients that either have a direct reference to the debloated library in their source code or which test suite covers at least one class of the library (static or dynamic usage). The 988 clients that statically use the library serve as the study subjects to answer RQ6. The 281/988 (28.4 %) clients that have at least a test that reaches the debloated library serve as the study subjects to answer RQ7.

5 RESULTS

We present our experimental results on the correctness, effectiveness, and impact of coverage-based debloating for automatically removing unnecessary bytecode from Java projects.

5.1 Debloating Correctness (RQ1 and RQ2)

In this section, we report on the successes and failures of JDBL to produce a correct debloated version of Java libraries.

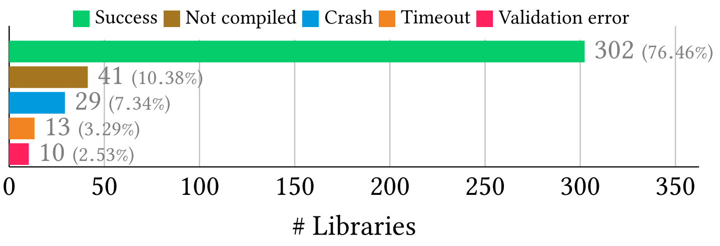


Fig. 4. Number of libraries for which JDBL succeeds or fails to produce a debloated JAR file.

5.1.1 RQ1. To what extent can a generic, fully automated coverage-based debloating technique produce a debloated version of Java libraries? In the first research question, we evaluate the ability of JDBL at performing automatic coverage-based debloating for the 395 libraries in our initial dataset. Here, we consider the debloating procedure to be successful if JDBL produces a valid debloated JAR file for a library. To reach this successful state, the project to be debloated must go through all the build phases of the Maven build life-cycle, i.e., compilation, testing, and packaging, according to the protocol described in Section 4.3.2.

Figure 4 shows a bar plot of the number of successfully debloated libraries. It also displays the number of cases where JDBL does not produce a debloated JAR file, due to failures in the build.

For the 395 libraries of our dataset, JDBL succeeds in producing a debloated JAR file for a total of 302 libraries and fails to debloat 93 libraries. Therefore, the overall debloating success rate of JDBL is 76.5 %. When considering only the libraries that were originally compiled, JDBL succeeds in debloating 85.3 % of the libraries. We manually identify and classify the causes of failures in four categories:

— *Not compiled.* As a sanity-check, we compile the project before injecting JDBL in its Maven build. The only modification consists in changing the *pom.xml* to request the generation of a JAR that contains the bytecode of the project, along with all its runtime dependencies. If this step fails, the project does not compile, and it is ignored for the rest of the evaluation.
— *Crash.* We run a second Maven build, with JDBL. This modifies the bytecode to remove unnecessary code. In certain situations, this procedure causes the build to stop at some phase and terminate abruptly, i.e., due to accessing invalid memory addresses, using an illegal opcode, or triggering an unhandled exception.
— *Time-out.* JDBL utilizes various coverage tools that instrument the bytecode of the project and its dependencies. This process induces an additional overhead in the Maven build process. Moreover, the incorrect instrumentation with at least one of the coverage tools may cause the test to enter into an infinite loop, e.g., due to blocking operations.
— *Validation error.* Maven includes dedicated plugins to check the integrity of the produced JAR file. JDBL alters the behavior of the project build by packaging the debloated JAR using the `maven-assembly-plugin`. Some other plugins may not be compatible with JDBL (e.g., when using customized assemblies), triggering validation errors during the build life-cycle. Moreover, we observe that for some libraries, the tests in the debloated JAR are not correctly executed due to particular library configurations in the `maven-surefire-plugin`.

We manually investigate the causes of the validation errors for the 10 libraries that fall into this category. We found that Maven fails to validate the execution of the tests, either due to errors when running the instrumented code to collect coverage or incompatibilities among plugins that exercise the instrumented version of the library. For example, in the case of *org.apache.commons: collection:4.0*, the `MANIFEST.MF` file is missing in the debloated JAR due to an incompatibility with

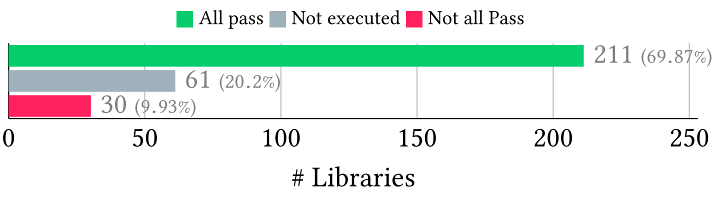


Fig. 5. Number of debloated libraries for which the test suite passes; number of debloated libraries for which the number of executed tests does not match the original test execution (ignored for the research question); number of debloated libraries that have at least one failing test case.

library plugins. Therefore, Maven fails to package the debloated bytecode. As another example, the Maven build of *org.yaml:snakeyaml:1.17* fails because of Yajta's instrumentation. This tool relies on Javassist for inserting probes in the bytecode. In this case, JDBL changes a class that was frozen by Javassist when it was loaded. Consequently, Javassist crashes because further changes in a frozen class are prohibited.[14]

Answer to RQ1: JDBL successfully produces a debloated JAR file for 302 libraries in our dataset, which represents 85.3 % of the libraries that compile correctly. This is the largest number of debloated subjects in the literature.

5.1.2 RQ2. To what extent do the debloated library versions preserve their original behavior w.r.t. the debloating workload? Our second research question evaluates the behavior of the debloated library with respect to its original version. This evaluation is based on the test suite of the project. We investigate if the code debloated by JDBL affects the results of the tests of the 302 libraries for which JDBL produces a valid JAR file. This behavioral correctness assessment corresponds to the last phase in the execution of JDBL.

Figure 5 summarizes the comparison between the test suite executed on the original and the debloated libraries. From the 302 successfully debloated, 211 (69.9 %) preserve the original behavior (i.e., all the tests pass). In the case of 30 (9.9 %) libraries, we observe at least one test failure. This high test success rate is a fundamental result to ensure that the debloated version of the artifact preserves the behavior of the library. A table with the full list of the 211 successfully debloated libraries that pass all the tests is available in the replication package of this article.[15]

We excluded 61 (20.2 %) libraries because the numbers of executed tests before and after the debloating did not match. This is due to changes in the tests' configuration after injecting JDBL into the build of the libraries. We excluded those libraries since different numbers of test runs imply a different test-based specification for the original and the debloated version of the library. Consequently, the results of the tests do not provide a sound basis for behavioral comparison. The manual configuration of the libraries is a solution to handle this problem (expected usage of JDBL), yet it is impractical in our experiments because of the large number of libraries that we debloat.

In total, we execute 342,835 unique tests, from which 341,430 pass and 1,405 do not pass (973 fail, and 432 result in an error). This represents an overall behavior preservation ratio of 99.59 %, considering the total number of tests. This result shows that our code-coverage debloating approach is able to capture most of the project behavior, as observed by the tests while removing the unnecessary bytecode.

[14]https://www.javassist.org/tutorial/tutorial.html.
[15]https://github.com/castor-software/jdbl-experiments/blob/master/list_of_libs_succesfully_debloated_with_jdbl.md.

We investigate the causes of test failures in the 30 libraries that have at least one failure. To do so, we manually analyze the logs of the tests, as reported by Maven. We find the following five causes:

— `NoClassDefFound (NCDF)`: JDBL mistakenly removes a necessary class.
— `TestAssertionFailure (TAF)`: the asserting conditions in the test fail for multiple reasons, e.g., flaky tests, or test configuration errors.
— `UnsupportedOperationException (UOE)`: JDBL mistakenly modifies the body of a necessary method, removing bytecode used by the test suite.
— `NullPointerException (NPE)`: a necessary object is referenced before being instantiated.
— `Other`: The tests are failing for another reason than the ones previously mentioned.

Table 2 categorizes the test failures for the 30 libraries with at least one test that does not pass. They are sorted in descending order according to the percentage of tests that fail on the debloated version. The first column shows the name and version of the library. Columns 2–7 represent the five causes of test failure according to our manual analysis of the tests' logs: `TAF`, `UOE`, `NPE`, `NCDF`, and `Other`. The column labeled as `Other` shows the number of test failures that we were not able to classify. The last column shows the percentage of tests that do not pass with respect to the total number of tests in each library. For example, *equalsverifier:3.4.1* has the largest number of test failures. After debloating, we observe 605 test failures out of 921 tests (283 `TAF`, 221 `NCDF`, and 1 `Other`). These test failures represent 65.7 % of the total number of tests in *equalsverifier:3.4.1.* This is an exceptional case, as for most of the debloated libraries, the tests that do not pass represent less than 5 % of the total.

The most common cause of test failure is `NCDF` (735), followed by `TAF` (592). We found that these two types of failures are related to each other: when the test uses a non-covered class, the log shows a `NCDF`, and the test assertion fails consequently. We notice that `NCDF` and `UOE` are directly related to the removal procedure during the debloating procedure, meaning that JDBL is removing necessary classes and methods, respectively. This occurs because there are some Java constructs that JDBL does not manage to cover dynamically, causing an incomplete debloating result, despite the union of information gathered from different coverage tools. Primitive constants, custom exceptions, and single-instruction methods are typical examples. These are ubiquitous components of the Java language, which are meant to support robust object-oriented software design, with little or no procedural logic. They are important for humans and they are useless for the machine to run the program. Consequently, they are not part of the executable code in the bytecode, and cannot be covered dynamically.

JDBL can generate a debloated program that breaks a few test cases. These cases reveal some limitations of JDBL concerning behavior preservation, i.e., it fails to cover some classes and methods, removing necessary bytecode. One of the explanations is that the coverage tools modify the bytecode of the libraries. Those modifications can cause some test failures. A failing test case stops the execution of the test and can introduce a truncated coverage report of the execution. Since some code is not executed after the failing assertion, some required classes or methods will not be covered and therefore debloated by JDBL. For example, in the *reflections* library, a library that provides a simplified reflection API, some tests verify the number of fields of a class extracted by the library. However, JaCoCo injects a field in each class, which will invalidate the asserts of *reflections* tests.

More generally, this reveals the remaining challenges of coverage-based debloating for real-world Java applications when using the test suite as a workload. For this study, handling these challenging cases to achieve 100 % correctness requires significant engineering effort, providing only marginal insights. Therefore, we recommend always using our validation approach to be safe of semantic alterations when performing aggressive debloating transformations.

Table 2. Classification of the Tests that Fail for the 30 Libraries that do not Pass All the Tests

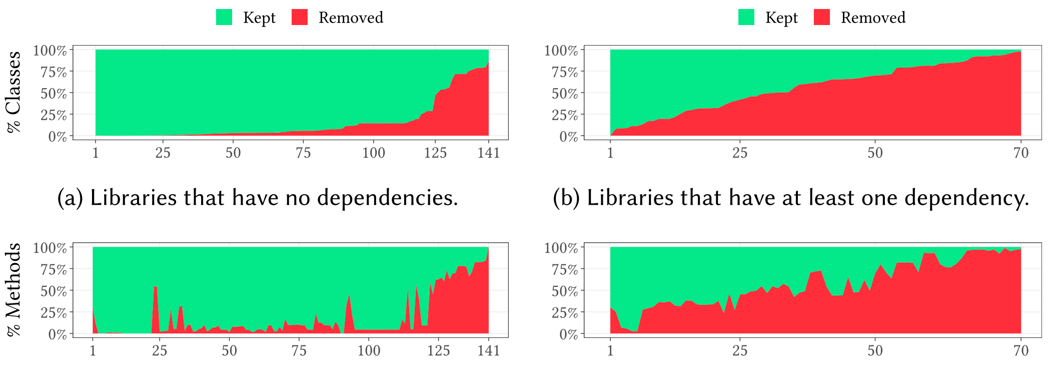
Library	`TAF`	`UOE`	`NPE`	`NCDF`	`Other`	Test failures
jai-imageio-core:1.3.1				3		3/3 (100.0 %)
jai-imageio-core:1.3.0				3		3/3 (100.0 %)
reflectasm:1.11.7	3			11		14/16 (87.5 %)
equalsverifier:3.3	273			315	1	589/894 (65.9 %)
equalsverifier:3.4.1	283			321	1	605/921 (65.7 %)
spark:2.0.0	3				22	25/57 (43.9 %)
logstash-logback-encoder:6.2				74	36	110/307 (35.8 %)
reflections:0.9.9	3					3/63 (4.8 %)
reflections:0.9.10	3					3/64 (4.7 %)
reflections:0.9.12	3					3/66 (4.5 %)
reflections:0.9.11	3					3/69 (4.3 %)
commons-jexl:2.0.1	6		2			8/223 (3.6 %)
commons-jexl:2.1.1	5		3			8/275 (2.9 %)
jackson-dataformat-csv:2.7.3		3				3/129 (2.3 %)
sslr-squid-bridge:2.7.0.377		1				1/43 (2.3 %)
jline2:2.14.3	2					2/141 (1.4 %)
jackson-annotations:2.7.5		1				1/77 (1.3 %)
commons-bcel:6.0	1					1/103 (1.0 %)
commons-bcel:6.2	1					1/107 (0.9 %)
commons-compress:1.12	2	1		2		5/577 (0.9 %)
commons-net:3.4				2		2/271 (0.7 %)
commons-net:3.5				2		2/274 (0.7 %)
jline2:2.13	1					1/141 (0.7 %)
commons-net:3.6				2		2/283 (0.7 %)
kryo-serializers:0.43		1				1/660 (0.2 %)
jongo:1.3.0					1	1/551 (0.2 %)
commons-codec:1.9		1				1/616 (0.2 %)
commons-codec:1.10		1				1/662 (0.2 %)
commons-codec:1.11		1				1/875 (0.1 %)
commons-codec:1.12		1				1/903 (0.1 %)
Total	592	11	5	735	61	1,405 (15.0 %)

We identified five causes of failures through the manual inspection of the Maven build testing logs: TestAssertionFailure (`TAF`), UnsupportedOperationException (`UOE`), NullPointerException (`NPE`), NoClassDefFound (`NCDF`), and other unknown causes (`Other`).

Answer to RQ2: JDBL automatically generates a debloated `JAR` that preserves the original behavior of 211 (69.9 %) libraries. A total of 341,430 (99.59 %) tests pass on 241 libraries. This behavioral assessment of coverage-based debloating demonstrates that JDBL preserves a large majority of the libraries' behavior, which is essential to meet the expectations of the libraries' users.

5.2 Debloating Effectiveness (RQ3, RQ4, and RQ5)

In this section, we report on the effects of debloating Java libraries with JDBL in terms of bytecode size reduction.



(a) Libraries that have no dependencies.

(b) Libraries that have at least one dependency.

(c) Libraries that have no dependencies.

(d) Libraries that have at least one dependency.

Fig. 6. Percentage of classes kept and removed in (a) libraries that have no dependencies, and (b) libraries that have at least one dependency. Percentage of methods kept and removed in (c) libraries that have no dependencies, and (d) libraries that have at least one dependency.

5.2.1 RQ3. How much bytecode is removed in the compiled libraries and their dependencies? To answer our third research question, we compare the status (kept or removed) of dependencies, classes, and methods in the 211 libraries correctly debloated with JDBL. The goal is to evaluate the effectiveness of JDBL to remove these bytecode elements through coverage-based debloating.

Figure 6 shows area charts representing the distribution of kept and removed classes and methods in the 211 correctly debloated libraries. To analyze the impact of dependencies, we separate the libraries into two sets: the libraries that have no dependency (Figure 6(a) and (c)), and the libraries that have at least one dependency (Figure 6(b) and (d)). In each figure, the x-axis represents the libraries in the set, sorted in increasing order according to the number of removed classes, whereas the y-axis represents the percentage of classes (Figure 6(a) and (b)) or methods (Figure 6(c) and (d)) kept and removed. The order of the libraries, on the x-axis, is the same for each figure.

Figure 6(a) shows the comparison between the percentages of kept and removed classes in the 141 libraries that have no dependency. A total of 116 libraries have at least one removed class. The library with the largest percentage of removed classes is *jfree-jcommon:1.0.23* with the 86.7 % of its classes considered as bloated. On the other hand, Figure 6(b) shows the percentage of removed classes for the 70 libraries that have at least one dependency. We observe that the ratio of removed classes in these libraries is significantly higher with respect to the libraries with no dependencies. All the libraries that have dependencies have at least one removed class, and 45 libraries have more than 50 % of their classes bloated. This result hints at the importance of reducing the number of dependencies to mitigate software bloat.

Figure 6(c) shows the percentage of kept and removed methods in the 141 libraries that have no dependencies. We observe that libraries with a few removed classes still contain a significant percentage of removed methods. For example, the library *net.iharder:base64:2.3.9* has 42.2 % of its methods removed in the 99.4 % of its kept classes. This suggests that a fine-grained debloat, to the level of methods, is beneficial for some libraries. The used classes may still contain a significant number of bloated methods. On the other hand, Figure 6(d) shows the percentage of kept methods in libraries with at least one dependency. All the libraries have a significant percentage of removed methods. As more bloated classes are in the dependencies, the artifact globally includes more bloated methods.

Now we focus on determining the difference between the bloat that is caused exclusively by the classes in the library, and the bloat that is a consequence of software reuse through the declaration

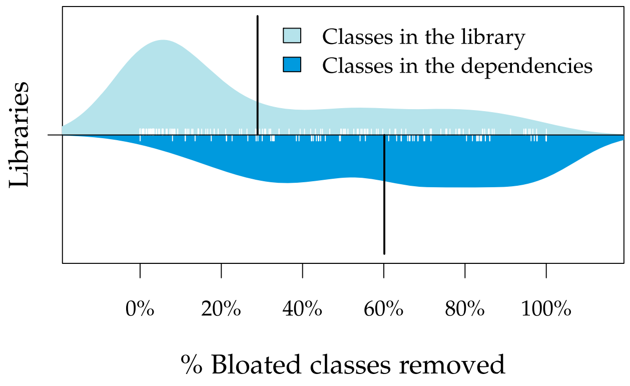


Fig. 7. Distribution of the percentage of bloated classes that belong to libraries, and bloated classes that belong to dependencies. The strip chart (white marks) in between represents the libraries that belong to each of the two groups. The two vertical bars represent the average value for each group.

Table 3. Summary of the Number of Dependencies, Classes, and Methods Removed in the 211 Libraries Correctly Debloated with JDBL

	Removed (%)
Dependencies	38/187 (20.3 %)
Classes	61,929/103,032 (60.1 %)
Methods	411,997/693,703 (59.4 %)

of dependencies. Figure 7 shows a beanplot [29] comparing the distribution of the percentage of bloated classes in libraries, with respect to the bloated classes in dependencies. The density shape at the top of the plot shows the distribution of the percentage of bloated classes that belong to the 211 libraries. The density shape at the bottom shows this percentage for the classes in the dependencies of the 70 libraries that have at least one dependency. The average bloat in libraries is 27.3 %, whereas in the dependencies it is 59.8 %. Overall, the average of bloated classes removed for all the libraries, including their dependencies, is 32.5 %. We perform a two-samples Wilcoxon test, which confirms that there are significant differences between the percentage of bloated classes in the two groups ($p\text{-value} < 0.01$). Therefore, we reject the null hypothesis and confirm that the ratio of bloated classes is more significant among the dependencies than among the classes of the artifacts.

Table 3 summarizes the debloating results for the dependencies, classes, and methods. Interestingly, JDBL completely removes the bytecode for 20.3 % of the dependencies. In other words, 38 dependencies in the dependency tree of the projects are not necessary to successfully execute the workload in our dataset. At the class level, we find 60.1 % of bloat, from which we determine that 36.7 % belong to dependencies. JDBL debloats 59.4 % of the methods, from which 41.8 % belong to dependencies.

Answer to RQ3: JDBL removes bytecode in all libraries. It reduces the number of dependencies, classes, and methods by 20.3 %, 60.1 %, and 59.4 %, respectively. This result confirms the relevance of the coverage-based debloating approach for reducing the unnecessary bytecode of Java projects, while preserving their correctness.

Table 4. Size in Bytes of the Elements in the JAR Files

Metrics	Size in bytes (%)
Resources	257,982,707 (22.4 %)
Bytecode	893,531,088 (77.6 %)
Non-bloated classes	283,424,921 (31.7 %)
Bloated classes	596,337,540 (66.7 %)
Bloated methods	13,768,627 (1.5 %)
Total size	1,151,513,795

5.2.2 RQ4. What is the impact of using the coverage-based debloating approach on the size of the packaged artifacts? We consider all the elements in the JAR files before the debloating, and study the size of the debloated version of the artifact, with respect to the original bundle. Decreasing the size of JAR files by removing bloated bytecode has a positive impact on saving space on disk, and helps reduce overhead when the JAR files are shipped over the network.

JAR files contain bytecode, as well as additional resources that depend on the functionalities of the artifacts (e.g., HTML, DLL, SO, and CSS files). JAR files also contain resources required by Maven to handle configurations and dependencies (e.g., MANIFEST.MF and *pom.xml*). However, JDBL is designed to debloat only executable code (class files). Therefore, we assess the impact of bytecode removal with respect to the executable code in the original bundle.

Table 4 summarizes the main metrics related to the content and size of the JAR files in our dataset. We observe that the additional resources represent 22.4 % of the total JAR size, whereas 77.6 % of the size is dedicated to the bytecode. This observation supports the relevance of debloating the bytecode in order to shrink the size of the Maven artifacts.

Overall, the bloated elements in the compiled artifacts in our dataset represent 610.3/893.7 MB (68.3 %) of pure bytecode: 596.5 MB of bloated classes and 13.8 MB of bloated methods. The used bytecode represents 31.7 % of the size. Interfaces, enumeration types, annotations, and exceptions represent 15.8% of the size on the disk of all the class files. In comparison with the classes, the debloat of methods represents a relatively limited size reduction. This is because we are reporting the removal of methods in the classes that are not entirely removed by JDBL. Furthermore, the methods cannot be completely removed, only the body of the method is replaced by an exception as detailed in Section 3.3.

Figure 8 shows a beanplot comparing the distribution of the percentage of bytecode reduction in the libraries that have no dependency, with respect to the libraries that have at least one dependency. From our observations, the average bytecode size reduction in the libraries that have dependencies (46.7 %) is higher than in the libraries with no dependencies (14.5 %). Overall, the average percentage of bytecode reduction for all the libraries is 25.8 %. We performed a two-sample Wilcoxon test, which shows that there are significant differences between those two groups (p-value < 0.01). Therefore, we reject the null hypothesis that the coverage-based debloating approach has the same impact in terms of reduction of the JAR size for libraries that declare dependencies, and libraries that do not.

We perform a Spearman's rank correlation test between the original number of classes in the libraries and the size of the removed bytecode. We found that there is a significant positive correlation between both variables ($\rho = 0.97$, p-value < 0.01). This result confirms the intuition that projects with many classes tend to occupy more space on disk due to bloat. However, the decision of what is necessary or not heavily depends on the library, as well as on the workload.

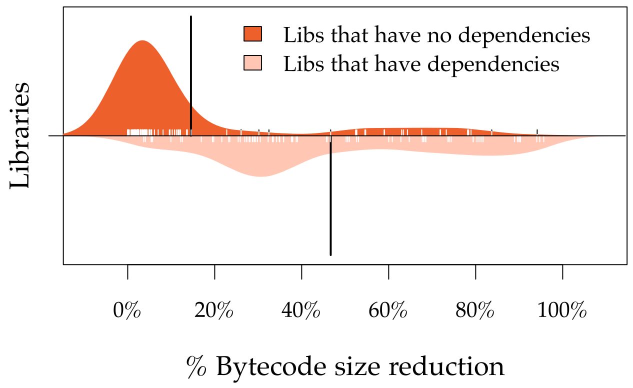


Fig. 8. Distribution of the percentage of reduction of the JAR size in libraries that have no dependencies and libraries that have at least one dependency, with respect to the original bundle.

Answer to RQ4: JDBL removes 68.3 % of pure bytecode in JAR files, which represents an average size reduction of 25.8 % per library JAR file. The JAR size reduction is significantly higher in libraries with at least one dependency compared to libraries with no dependency.

5.2.3 RQ5. How does coverage-based debloating compare with the state-of-the-art of Java debloating regarding the size of the packaged artifacts and behavior preservation? In this research question, we compare the debloating results of JDBL with respect to JShrink. The comparison is based on two metrics: the reduction of bytecode size after debloat, and the preservation of test results after debloat. The JShrink benchmark includes 25 Java projects. To answer RQ5, we discard eight projects: Six that are multi-module (JDBL is designed to debloat single-module Maven projects), and two projects whose builds fail due to test errors caused by unavailable network resources. Therefore, our comparison of JDBL and JShrink is based on 17 projects in total. For each project, we configure it to execute JDBL and generate a debloated version of the fat JAR. To validate the semantic correctness of the debloated artifact, we execute the project's test suite on the debloated version.

Table 5 describes the benchmark along with the debloating results obtained with JShrink and JDBL. The first column shows the name of the project as it appears on GitHub. The second column shows the commit SHA of the project, which is the same used in the companion article of JShrink [8]. The third column is the size of the original packaged JAR of the project, which includes all its dependencies with a `compile` scope. Projects are sorted in decreasing order according to their original size.

We report the bytecode size reduction of the debloated version of the projects achieved with JShrink and JDBL. The average size reduction achieved with JDBL is 35.1 %, which is more than double the size reduction obtained with JShrink. An explanation is that JShrink makes a more conservative debloating decision by setting all public methods, main methods, and test methods of libraries as entry points to approximate possible usages, whereas JDBL debloats according to a workload (i.e., the coverage information collected by running the test suite of the library). We observe that the percentage of reduction varies greatly among the projects depending on their size. We performed a Spearman's rank correlation test between the size of the original compiled project and the percentage of size reduction obtained with JShrink and JDBL. We found that there is a significant positive correlation between both variables for JShrink ($\rho = 0.52$, p-value < 0.05) and

Table 5. Debloating Results of JShrink and JDBL in the Benchmark of Bruce et al. [8]

Project Name	Commit SHA	Size (kB)	#Tests	JShrink Size Reduction	JShrink #Test Failures	JDBL Size Reduction	JDBL #Test Failures
lanterna	5982dbf	18,050.3	34	2.0 %	✓ (0)	66.4 %	✓ (0)
AutoLoadCache	06f6754	14,845.8	11	20.2 %	✗ (9)	66.8 %	✓ (0)
gwt-cal	e7e5250	13,059.8	921	17.5 %	✓ (0)	42.2 %	✓ (0)
maven-config-processor-plugin	c92e588	5,762.4	77	29.8 %	✓ (0)	73.0 %	✓ (0)
Bukkit	f210234	4,781.3	906	18.5 %	✓ (0)	61.5 %	✓ (0)
Mybatis-PageHelper	525394c	4,272.3	106	23.9 %	✗ (55)	66.4 %	✓ (0)
RxRelay	82db28c	2,295.9	58	17.5 %	✓ (0)	10.0 %	✓ (0)
RxReplayingShare	fbedd63	2,117.6	20	22.1 %	✓ (0)	7.3 %	✓ (0)
qart4j	70b9abb	1,869.0	1	46.8 %	✓ (0)	59.1 %	✓ (0)
retrofit1-okhttp3-client	9993fdc	718.9	9	11.5 %	✓ (0)	21.9 %	✓ (0)
junit4	67d424b	407.1	1,081	6.8 %	✗ (13)	4.7 %	✓ (0)
gson	27c9335	235.8	1,050	5.5 %	✓ (0)	1.7 %	✓ (0)
zt-zip	6933db7	134.3	121	11.3 %	✓ (0)	10.3 %	✓ (0)
TProfiler	8344d1a	102.8	3	10.2 %	✓ (0)	83.1 %	✓ (0)
Algorithms	9ae21a5	99.9	493	5.5 %	✓ (0)	12.2 %	✓ (0)
http-request	2d62a3e	36.1	163	6.6 %	✓ (0)	7.5 %	✓ (0)
DiskLruCache	3e01635	22.7	61	1.7 %	✓ (0)	2.9 %	✓ (0)
Total	N.A	6,881,194.8	5,115	N.A	77	N.A	0
Median	N.A	1,869.0	77	11.5 %	0	21.9 %	0
Avg.	N.A	4,047.8	300.9	15.14 %	4.53	35.1 %	0

JDBL ($\rho = 0.52$, p-value < 0.05). This result confirms the results obtained in RQ4 where we show that larger libraries are prone to become more bloated.

To assess the behavior preservation of the debloated projects w.r.t. their original version, we run existing test cases before and after debloating. Table 5 shows the semantic preservation capabilities of JShrink and JDBL on the benchmark of 17 projects. We consider a debloated project to have broken semantics if at least one of its tests fails after debloating. A project with no broken semantic is denoted by ✓, while ✗ denotes the presence of at least one test failure after debloating. JShrink causes test failures in three projects (77 failures in total). On the contrary, JDBL preserves the behavior of all the projects on this benchmark, according to the results of the tests.

Answer to RQ5: JDBL successfully debloats the 17 single-module Java projects in the benchmark of Bruce et al. [8], with a size reduction of 35.1 % on average, and preserves the behavior according to the tests. JShrink reduces size by 15.1 % on average. This is evidence that coverage-based debloating is a promising technique that advances the state-of-the-art of Java bytecode debloating.

5.3 Impact of Debloating on Library Clients (RQ6 and RQ7)

In this section, we study the repercussions of performing coverage-based debloating on library clients. To the best of our knowledge, this is the first experimental report that measures the impact of debloating libraries on the syntactic and semantic correctness of their clients.

5.3.1 RQ6. To what extent do the clients of debloated libraries compile successfully? In this research question, we investigate how debloating a library with a coverage-based approach impacts the compilation of the library's clients. We hypothesize that the essential functionalities of the library are less likely to be debloated, limiting the syntactic negative impact on their clients.

As described in Section 4.3.4, we consider the 1,354 clients that use the 211 debloated libraries that pass all the tests. We check that the clients use at least one class in the library through static analysis. We identify 988/1,354 (73.0 %) clients that satisfy this condition. Figure 9 shows the

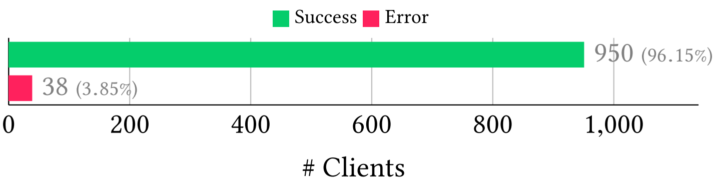


Fig. 9. Results of the compilation of the 988 clients that use at least one debloated library in the source code.

Table 6. Frequency of the Errors for the 38 Unique Clients that have Compilation Errors

Description	# Libraries	# Clients	Occurrence
Cannot find class	12/19 (63.2 %)	20/38 (52.6 %)	314/640 (49.1 %)
Unmappable character for encoding UTF8	1/19 (5.3 %)	1/38 (2.6 %)	100/640 (15.6 %)
Package does not exist	6/19 (31.6 %)	13/38 (34.2 %)	78/640 (12.2 %)
Cannot find variable	7/19 (36.8 %)	9/38 (23.7 %)	77/640 (12.0 %)
Cannot find method	1/19 (5.3 %)	1/38 (2.6 %)	28/640 (4.4 %)
Static import only from classes and interfaces	1/19 (5.3 %)	1/38 (2.6 %)	14/640 (2.2 %)
Method does not override a method from a supertype	2/19 (10.5 %)	2/38 (5.3 %)	12/640 (1.9 %)
Processor error	2/19 (10.5 %)	11/38 (28.9 %)	11/640 (1.7 %)
Cannot find other symbol	2/19 (10.5 %)	2/38 (5.3 %)	4/640 (0.6 %)
UnsupportedOperationException	1/19 (5.3 %)	1/38 (2.6 %)	1/640 (0.2 %)
Plugin verification	1/19 (5.3 %)	1/38 (2.6 %)	1/640 (0.2 %)
11 Unique errors	19 Unique libraries	38 Unique clients	640 Compilation errors

Note that a client can have multiple errors from different categories.

results obtained after attempting to compile the clients with the debloated library. JDBL generates debloated libraries for which 950 (96.2 %) of their clients successfully compile.

From the 988 clients that use at least one class of the library, we only observe compilation failures in 38 (3.8 %) clients. Table 6 shows our manual classification of the errors, based on the analysis of the Maven build logs. The first column describes the error message, columns 2–3 represent the number of libraries that trigger this kind of error, the number of clients that are affected, and the percentage relative to the number of libraries or clients impacted by a compilation error. Note that a client can be impacted by several different errors. The fourth column represents the occurrence of the error in the clients, as quantified from the Maven logs.

The causes of compilation errors are diverse. However, they are primarily due to errors related to missing packages, classes, methods, and variables (79.8 % of all the compilation errors). The debloating procedure directly causes those errors, as the bytecode elements are removed, and the clients do not compile. We detect 640 errors in total. Most of them occur for similar causes. Indeed, 20/38 (52.6 %) clients are not compiling because of one single error cause (which can be unique for each client). Moreover, when a client fails for a library, the other clients of the same library are generally failing for the same reason. It means that a single action can solve most of the client's problems, i.e., by adding the missing element to the debloated library.

Several clients face compilation issues because of their plugins. In order to have the client use the debloated library, we inject the debloated library inside the bytecode folder of the clients. Unfortunately, some plugins of the clients will also analyze the bytecode of the debloated library that may not follow the same requirements. `Plugin verification error`, `Unmappable character for encoding UTF8`, and `Processor error` are related to this type of bytecode validation.

In the list of errors, we also observe a runtime exception: `UnsupportedOperationException`. This error is unexpected since the compilation should not execute code and therefore should not trigger a runtime exception. It happens during the build of jenkinsci/warnings-plugin, which uses the *commons-io:2.6* library. In this case, the compilation itself of this client does not fail, but the

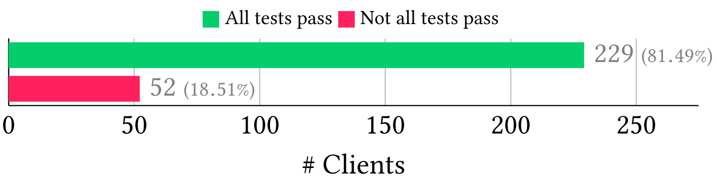


Fig. 10. Results of the tests on the 281 clients that cover at least one debloated library.

Table 7. Frequency of the Exceptions Thrown During the Execution of the Tests for the 52 Unique Clients that have Failing Test Cases

Exception	# Libraries	# Clients	Occurrence
UnsupportedOperationException	27/37 (73.0 %)	40/52 (76.9 %)	609/716 (85.1 %)
IllegalStateException	1/37 (2.7 %)	2/52 (3.8 %)	55/716 (7.7 %)
NoClassDefFoundError	6/37 (16.2 %)	6/52 (11.5 %)	24/716 (3.4 %)
Assert	5/37 (13.5 %)	5/52 (9.6 %)	12/716 (1.7 %)
ExceptionInInitializerError	4/37 (10.8 %)	4/52 (7.7 %)	4/716 (0.6 %)
TargetHostsLoadException	1/37 (2.7 %)	1/52 (1.9 %)	4/716 (0.6 %)
NullPointerException	1/37 (2.7 %)	1/52 (1.9 %)	3/716 (0.4 %)
TimeoutException	1/37 (2.7 %)	1/52 (1.9 %)	2/716 (0.3 %)
PushSenderException	1/37 (2.7 %)	1/52 (1.9 %)	2/716 (0.3 %)
AssertionError	1/37 (2.7 %)	1/52 (1.9 %)	1/716 (0.1 %)
10 Unique exceptions	37 Unique libraries	52 Unique clients	716 Failing tests

Note that a client can have multiple exceptions from different categories.

Maven build does. One of the Maven plugins of this project relies on one method that is debloated in `apache/commons-io`, therefore, the compilation does not fail because of the source code of the client but because of one particular Maven plugin used by the client.

Answer to RQ6: JDBL preserves the syntactic correctness of 950 (96.2 %) clients that use a library debloated by JDBL. This is the first empirical demonstration that debloating can preserve essential functionalities to successfully compile the clients of debloated libraries.

5.3.2 *RQ7. To what extent do the clients behave correctly when using a debloated library?* In this research question, we analyze another facet of debloating that may affect the clients: the disturbance of their behavior. To do so, we use the test suite of the clients as the oracle for assessing correct behavior. If a test in the client fails with the debloated library, then the coverage-based debloating breaks the behavior of the client.

We consider 1,354 clients to answer this question. They use the 211 debloated libraries that pass all the tests. We check that the client tests cover at least one class in the library, through dynamic code coverage. We identify 281/1,354 (20.8 %) clients that satisfy this condition.

Figure 10 presents the test results after building the clients with the debloated library. In total, 229 (81.5 %) clients pass all their tests, i.e., they behave the same with the original and with the debloated library. There are 52 (18.5 %) clients that have more failing test cases with the debloated library than with the original. However, the number of tests that fail is only 716/44,357 (1.6 %) of the total number of tests in the clients. This indicates that the negative impact of debloated libraries, as measured by the number of affected client tests, is marginal.

We investigate the causes of the test failures. Table 7 quantifies the exceptions thrown by the clients. The first column shows the 10 types of exceptions that we find in the Maven logs. Columns

2–3 represent the number of libraries involved in the failure and the number of clients affected. Column 4 represents the occurrence of the exception, as quantified from the logs.

`UsupportedOperationException` is the most frequent exception, with 609 occurrences in the failing tests, affecting 76.9 % of clients with errors. This exception is triggered when one of the debloated methods is executed. The second most common exception is `IllegalStateException`, with a total of 55 occurrences. This exception happens when the client tries to load a bloated configuration class and fails. The third most frequent exception, with 24 occurrences, is `NoClassDefFoundError`. Similar to `UsupportedOperationException`, it happens when the clients try to load a debloated class in the library.

Interestingly, there are only 12 assertion-related exceptions (11 `Assert` and 1 `AssertionError`). Having runtime exceptions that are triggered during the executions of the clients is less harmful than having behavior changes. The runtime exceptions can be monitored and handled while running the client, while a behavior change can stay hidden and corrupt the state of the clients.

Answer to RQ7: JDBL preserves the behavior of 229 (81.5 %) clients of debloated libraries. The 52 other clients still pass 43,684 (98.5 %) of their test cases. In these cases, 99.1 % of the test failures are due to a missing class or method, which can be easily located and fixed. This experiment shows that the risks of removing code in software libraries are limited for their clients.

6 DISCUSSION

In this section, we discuss key aspects of the design for coverage-based debloating. Then, we address the threats to the validity of the evaluation of JDBL.

6.1 Complementarity of Code-Coverage Tools

As presented in Section 3.3, we leverage the diversity of implementations of code coverage tools and the dynamic logging capabilities of the JVM class loader to collect precise coverage information. Now, we discuss the advantages of using this approach for debloating.

We collect and aggregate the coverage reports of the four tools used by JDBL to capture class usage information: JaCoCo, JCov, Yajta, and the JVM class loader. We consider a class as covered if it is reported as used by at least one of these tools. Figure 11 shows a Venn diagram of the classes reported as covered by each tool. There are a total of 76,549 classes covered by at least one tool in our dataset of 211 successfully debloated libraries. One key observation is that JaCoCo covers only 59,934 (78.3 %) of the classes that are used when running our workloads. This means that if we rely on JaCoCo only, the state-of-the-art coverage tool for Java, we would capture a significant share of false positive cases of bloated classes. This is critical, since removing these classes would produce a debloated project that cannot be properly used to run the workload. Another interesting observation is about the diversity of behaviors in modern coverage tools. There are only 34,582 (45.2 %) classes that are covered by the four tools. The JVM class loader is the one that captures the largest number of unique classes: 14,972 (19.6 %). This is because it logs the usage of dynamically loaded classes at runtime. In contrast, the other coverage tools can provide more fine-grained coverage information (e.g., methods and instruction) but miss usages of dynamically loaded resources (e.g., classes loaded via the Java reflection mechanism). The addition of JCov and Yajta improves the coverage of used classes, accounting for 2 and 662 unique covered classes, respectively.

This is evidence that combining the features of several coverage tools is relevant to accurately capture the code that is used at runtime. Capturing the complete coverage of classes that are necessary for a workload is critical for debloating. Failing to do so leads to the generation of a debloated project that does not compile, or even worse, that leads to runtime errors when client projects use

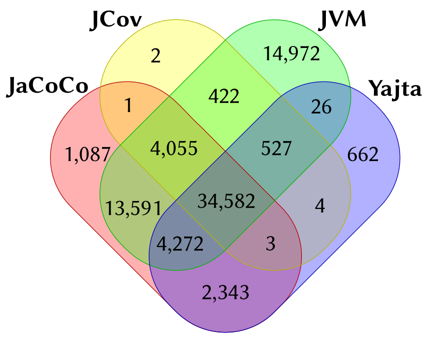


Fig. 11. Number of classes covered by the diverse tools used to collect coverage information in our dataset of 211 successfully debloated libraries.

debloated libraries. A more in-depth analysis of the similarities and differences of code coverage tools is a promising direction for future research on Java bytecode debloating.

6.2 Execution Time

The vision for debloating is to integrate it as part of building pipelines. In this context, execution time is an important consideration. Here, we discuss whether the execution time of JDBL makes coverage-based debloating feasible for everyday software development. In our experiments, we executed JDBL on 395 libraries, for a total of 1 day and 10:55:00 h. This represents an average debloating time of 5.3 minutes per library. Table 8 shows the execution times of JDBL and JShrink, with the benchmark of Bruce et al. [8]. The first and second columns show the projects' names and commit SHAs, respectively. The third and fourth columns show the execution time of JDBL and JShrink for each project, measured in seconds and sorted in decreasing order according to the results of JDBL. The comparison is made on the same hardware as the main JDBL experiment, described in Section 4.3. We observe that it took a total of 2,685 seconds (less than one hour) to debloat the benchmark with JDBL, whereas JShrink took a total of 29,397 seconds (more than eight hours). The average debloating time for this benchmark using JDBL is 157.9s (less than 3 minutes per project), which is 11 times faster than JShrink.

With times in the range of minutes, coverage-based debloating with JDBL can be used in a daily build. We also show that JDBL significantly improves the state-of-the-art of Java debloating, regarding execution time. This is an important contribution toward the integration of debloating into regular development processes.

6.3 Threats to Validity

6.3.1 Internal Validity. The threats to internal validity relate to the effectiveness of JDBL on generic real-world Java projects, as well as the design decisions that can influence our results. Coverage-based debloating has some inherent limitations, e.g., inadequate test cases and random behaviors. To mitigate these threats, we use as study subjects libraries with high coverage (see Table 1), and execute the test suite three times to avoid test randomness. If the test suite does not capture all desired behaviors, some necessary code might not be executed and be removed. The debloated libraries can also have non-deterministic test cases. For example, tests that use the current date and time to perform an action or not. Due to these behaviors, executing the application multiple times with the same input may lead to different coverage results.

As explained in Section 3.3, JDBL relies on a complex stack of existing bytecode coverage tools. It is possible that some of these tools may fail to instrument the classes for a particular project. However, since we rely on a diverse set of coverage tools, the failures of one specific tool are

Table 8. Execution Time of JDBL and JShrink in the Benchmark of Bruce et al. [8]

Project	Commit SHA	JDBL Execution Time (s)	JShrink Execution Time (s)
maven-config-processor-plugin	c92e588	612	1,159
lanterna	5982dbf	365	1,628
AutoLoadCache	06f6754	347	5,941
gwt-cal	e7e5250	293	2,075
Bukkit	f210234	143	10,452
Mybatis-PageHelper	525394c	128	1,749
RxRelay	82db28c	109	502
RxReplayingShare	fbedd63	95	753
junit4	67d424b	80	352
retrofit1-okhttp3-client	9993fdc	77	506
qart4j	70b9abb	72	2,137
DiskLruCache	3e01635	66	223
http-request	2d62a3e	63	616
gson	27c9335	63	842
zt-zip	6933db7	60	111
TProfiler	8344d1a	56	98
Algorithms	9ae21a5	56	253
Total	N.A	2,685	29,397
Median	N.A	80	753
Avg.	N.A	157.9	1,729.2

likely to be corrected by the others. 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. 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 build the fat `JAR` of all the study subjects.

6.3.2 *External Validity.* The threats to external validity are related to the generalizability of our findings. Our observations in Section 5 about bloat are made on single-module Maven projects and the Java ecosystem. Our findings are valid for software projects with these particular characteristics. Meanwhile, coverage-based debloating in different languages could yield different conclusions than ours. Moreover, our debloating results are influenced by the coverage of the libraries and clients used as study subjects. However, we took care to select open-source Java libraries available on GitHub, which cover projects from different domains (e.g., logging, database handling, encryption, IO utilities, metaprogramming, and networking).[16] To the best of our knowledge, this is the largest set of study subjects used in software debloating experiments.

6.3.3 *Construct Validity.* The threats to construct validity are related to the relation between the coverage-based debloating approach and the experimental protocol. Our analysis is based on a diverse set of real-world open-source Java projects, with minimal modifications to run JDBL (only the *pom.xml* file is modified). We assume that all the plugins involved in the Maven build life-cycle are correct, as well as all the generated reports. Note that, if a dependency is not resolved correctly by Maven, then its bytecode will not be instrumented. Thus, the quality of the debloat result depends on the effectiveness of the Maven dependency resolution mechanism.

The applicability of coverage-based debloating depends on the quality of the workload. In our experiments, we rely on the projects' test suite. Consequently, our observations partly depend on the coverage of the projects. As explained in Section 4.2, the coverage of the libraries in our dataset is high. On the other hand, the coverage of the clients is lower (20.24% on average), which may cause some used functionalities in the debloated libraries that are not executed by the clients' tests. However, we believe this does not affect our results because we assess the semantic correctness of

[16]See https://github.com/castor-software/jdbl-experiments/blob/master/dataset/data/jdbl_dataset.json.

the client applications when using the debloated version of a library based on the client's usage intent expressed by its test suite.

7 RELATED WORK

In this section, we present the works related to software debloating techniques and dynamic analysis.

7.1 Software Debloating

Research interest in software debloating has grown in recent years, motivated by the reuse of large open-source libraries designed to provide several functionalities for different clients [15, 27]. Seminal work on debloating for Java programs was performed by Tip et al. [53, 54]. They proposed a comprehensive set of transformations to reduce the size of Java bytecode including class hierarchy collapsing, name compression, constant pool compression, and method inlining. Recent works investigate the benefits of debloating Java frameworks and Android applications using static analysis. Jiang et al. [26] presented JRed, a tool to reduce the attack surface by trimming redundant code from Java binaries. RedDroid [25] and PolyDroid [20] propose debloating techniques for mobile devices. They found that debloating significantly reduces the bandwidth consumption used when distributing the application, improving the performance of the system by optimizing resources. Other works rely on debloating to improve the performance of the Maven build automation system [9], removing bloated dependencies [50], and mitigating runtime bloat [59]. More recently, Haas et al. [19] investigate the use of static analysis to detect unnecessary code in Java applications based on code stability and code centrality measures. Most of these works show that static analysis, although conservative by nature, is a useful technique for debloating in practice.

To improve the debloating results of static analysis, recent debloating techniques drive the removal process using information collected at runtime. In this context, various dynamic analysis strategies can be adopted, e.g., monitoring, debugging, or performance profiling. This approach allows debloating tools to collect execution paths, tailoring programs to specific functionalities by removing unused code [21, 45, 55]. Unfortunately, most of the existing tools currently available for this purpose do not target large Java applications, focusing primarily on small C/C++ executable binaries. Sharif et al. [46] propose Trimmer, a debloating approach that relies on user-provided configurations and compiler optimization to reduce code size. Qian et al. [42] present RAZOR, a tool for debloating program binaries based on test cases and control-flow heuristics. However, the authors do not provide a thorough analysis of the challenges and benefits of using code coverage to debloat software. More recently, Bruce et al. [8] propose JShrink, a tool to dynamically debloat modern Java applications. However, JShrink is not directly automatable within a build pipeline and the effect of debloating on the library clients is not studied. These previous works assess the impact of debloating on the size of the programs, yet, they rarely evaluate to what extent the debloating transformations preserve program behavior.

This work contributes to the state-of-the-art of software debloating. We propose an approach for debloating Java libraries based on the usage of code coverage to identify unused software parts. Our tool, JDBL, integrates the debloating procedure into the Maven build life-cycle, which facilitates its evaluation and its integration in most real-world Java projects. We evaluate our approach on the largest set of programs ever analyzed in the debloating literature, and we provide the first quantitative investigation of the impact of debloating on the library clients.

7.2 Dynamic Analysis

Dynamic analysis is the process of collecting and analyzing the data produced from executing a program. This long-time advocated software engineering technique is used for several tasks,

Table 9. Comparison of Existing Java Debloating Techniques, w.r.t. this Work

Tool	Target	Analysis	Scale	Granularity				Correctness evaluation criteria		
				F	M	C	D	lib compiles	lib's tests pass	lib's clients tests pass
DepClean [50]	Source	Static	30 libs	✗	✗	✗	✓	✓	✓	✗
Jax [54]	Bytecode	Static	13 libs	✓	✓	✓	✗	✓	✗	✗
JRed [26]	Bytecode	Static	9 libs	✗	✓	✓	✗	✓	✗	✗
J-Reduce [28]	Bytecode	Dynamic	3 libs	✗	✗	✓	✗	✓	✓	✗
JShrink [8]	Bytecode	Hybrid	26 libs	✓	✓	✓	✗	✓	✓	✗
JDBL	Bytecode	Dynamic	395 libs and 1,370 clients	✗	✓	✓	✓	✓	✓	✓

Target is the type of artifact considered for debloating; Analysis refers to the type of code analysis performed for debloating; Scale provides the number of study subjects used to evaluate the technique; Granularity is the code level at which debloating is performed: field (F), method (M), class (C) or dependency (D); the last columns enumerate the three ways found in the state-of-the-art to assess the validity and utility of the debloated artifact.

such as program slicing [2], program comprehension [12], or dynamic taint tracking [4]. Through dynamic analysis, developers can obtain an accurate picture of the software system by exposing its actual behavior. For example, trace-based compilation uses dynamically-identified frequently-executed code sequences (traces) as units for optimizing compilation [16, 24]. Mururu et al. [37] implemented a scheme to perform demand-driven loading of libraries based on the localization of call sites within its clients. This approach allows reducing the exposed code surface of vulnerable linked libraries, by predicting the near-exact set of library functions needed at a given call site during the execution. Palepu et al. [40] use dynamic analysis to effectively summarize the execution and behavior of modern applications that rely on large object-oriented libraries and components. In this work, we employ dynamic analysis for bytecode reduction, as opposed to runtime memory bloat, which was the target of previous works [5, 35, 36, 38, 39, 58, 60].

In Java, dynamic analysis is often used to overcome the limitations of static analysis. Landman [31] performed a study on the usage of dynamic features and found that reflection was used in 78 % of the analyzed projects. Recent work from Xin et al. [57] utilizes execution traces to identify and understand features in Android applications by analyzing their dynamic behavior. In order to leverage dynamic analysis for debloating, we need to collect a very accurate coverage report, which guides the debloating procedure.

Our work contributes to the state-of-the-art of dynamic analysis for Java programs. Our technique combines information obtained from four distinct code coverage tools through bytecode instrumentation [6]. The composition of these four types of observations allows us to build a very accurate and complete coverage report, which is necessary to identify exactly what parts of the code are used at runtime and which ones can be removed. To collect coverage, we rely on the test suite of the libraries. This approach is similar to other dynamic analyses, e.g., for finding backward incompatibilities [10].

Table 9 summarizes the state-of-the-art of published techniques for debloating Java applications in comparison with JDBL. As observed, most existing techniques target bytecode instead of source code. DepClean [50] is the exception, which focuses on debloating *pom.xml* files based on static bytecode analysis. JShrink [8] uses a combination of static and dynamic analysis to address the potential unsoundness of static analysis in the presence of new language features. To our knowledge, JDBL is the first fully automatic debloating technique that also debloats code in third-party dependencies, and that assesses the correctness of debloating with respect to both the successful build of the debloated library and the successful execution of the library's clients. Furthermore, our experiments are at least one order of magnitude larger than previous works.

8 CONCLUSION

In this work, we introduce coverage-based debloating for Java applications. We have addressed one key challenge of dynamic debloating: collect accurate and complete coverage information that includes the minimum set of classes and methods that are necessary to execute the program with a given workload. We implemented coverage-based debloating in an open-source tool called JDBL. We have performed the largest empirical validation of Java debloating in the literature with 354 libraries and 1,354 clients that use these libraries. We evaluated JDBL using an original experimental protocol that assessed the impact of debloating on the libraries' behavior, their size, as well as on their clients. Our results indicate that JDBL can reduce 68.3 % of the bytecode size and that 211 (69.9 %) debloated libraries compile and preserve their test behavior. We also show that JDBL outperforms JShrink regarding size reduction and behavior preservation when used on the same benchmark as in the JShrink article. For the first time in the literature, we assess the utility of debloated libraries for their clients: 81.5 % of the clients can successfully compile and run their test suite with a debloated library.

Our results provide evidence of the massive presence of unnecessary code in software applications and the usefulness of debloating techniques to handle this phenomenon. Furthermore, we demonstrate that dynamic analysis can be used to automatically debloat libraries while preserving the functionalities that are necessary for their clients.

The next step of coverage-based debloating is to specialize applications with respect to usage profiles collected in production environments, extending the debloating to other parts of the program stack, e.g., to the Java Runtime Environment (JRE), program resources, or containerized applications. As for the empirical investigation of the impact of debloating, we aim at evaluating the effectiveness of coverage-based debloating in reducing the attack surface of modern applications. These are major milestones towards full-stack debloating for software hardening.

ACKNOWLEDGMENTS

We would like to thank Dr. Bobby Bruce and all the authors of JShrink for sharing the source code and providing valuable feedback during our reproduction of their experiments.

REFERENCES

[1] Ioannis Agadakos, Nicholas Demarinis, Di Jin, Kent Williams-King, Jearson Alfajardo, Benjamin Shteinfeld, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. 2020. Large-scale debloating of binary shared libraries. *Digital Threats: Research and Practice* 1, 4 (2020), 28 pages.

[2] Hiralal Agrawal and Joseph R. Horgan. 1990. Dynamic program slicing. *SIGPLAN Notices* 25, 6 (1990), 246–256.

[3] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the USENIX Security Symposium.* 1697–1714.

[4] Jonathan Bell and Gail Kaiser. 2014. Phosphor: Illuminating dynamic data flow in commodity JVMs. *ACM SIGPLAN Notices* 49, 10 (2014), 83–101.

[5] Suparna Bhattacharya, Kanchi Gopinath, and Mangala Gowri Nanda. 2013. Combining concern input with program analysis for bloat detection. In *Proceedings of the OOPSLA.* 745–764.

[6] Walter Binder, Jarle Hulaas, and Philippe Moret. 2007. Advanced java bytecode instrumentation. In *Proceedings of the Symposium on Principles and Practice of Programming in Java.* 135–144.

[7] Hudson Borges and Marco Tulio Valente. 2018. What's in a github star? Understanding repository starring practices in a social coding platform. *Journal of Systems and Software* 146 (2018), 112–129.

[8] 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 ESEC/FSE.* 135–146.

[9] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build system with lazy retrieval for java projects. In *Proceedings of the ESEC/FSE.* 643–654.

[10] Lingchao Chen, Foyzul Hassan, Xiaoyin Wang, and Lingming Zhang. 2020. Taming behavioral backward incompatibilities via cross-project testing and analysis. In *Proceedings of the ICSE.* 112–124.

[11] Yurong Chen, Tian Lan, and Guru Venkataramani. 2017. DamGate: Dynamic adaptive multi-feature gating in program binaries. In *Proceedings of the FEAST Workshop*. 23–29.

[12] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. 2009. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering* 35, 5 (2009), 684–702.

[13] DiUS. 2022. Brings the Popular Ruby Faker Gem to Java. Retrieved November 21st, 2021 from https://github.com/DiUS/java-faker.

[14] Thomas Durieux, César Soto-Valero, and Benoit Baudry. 2021. DUETS: A dataset of reproducible pairs of java library-clients. In *Proceedings of the MSR*. 545–549.

[15] Sebastian Eder, Henning Femmer, Benedikt Hauptmann, and Maximilian Junker. 2014. Which features do my users (Not) use?. In *Proceedings of the ICSME*. 446–450.

[16] Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David Mandelin, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, Jesse Ruderman, Edwin W. Smith, Rick Reitmaier, Michael Bebenita, Mason Chang, and Michael Franz. 2009. Trace-based just-in-time type specialization for dynamic languages. *ACM SIGPLAN Notices* 44, 6 (2009), 465–478.

[17] Antonios Gkortzis, Daniel Feitosa, and Diomidis Spinellis. 2021. Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities. *Journal of Systems and Software* 172 (2021), 110653.

[18] Zhaoqiang Guo, Shiran Liu, Jinping Liu, Yanhui Li, Lin Chen, Hongmin Lu, and Yuming Zhou. 2021. How far have we progressed in identifying self-admitted technical debts? A comprehensive empirical study. *ACM Transactions on Software Engineering and Methodology* 30, 4 (2021), 56 pages.

[19] Roman Haas, Rainer Niedermayr, Tobias Roehm, and Sven Apel. 2020. Is static analysis able to identify unnecessary source code? *ACM Transactions on Software Engineering and Methodology* 29, 1 (2020), 1–11.

[20] Brian Heath, Neelay Velingker, Osbert Bastani, and Mayur Naik. 2019. PolyDroid: Learning-driven specialization of mobile applications. arXiv:1902.09589. Retrieved from https://arxiv.org/abs/1902.09589.

[21] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the CCS*. 380–394.

[22] G. J. Holzmann. 2015. Code inflation. *IEEE Software* 32, 2 (2015), 10–13.

[23] Ferenc Horváth, Tamás Gergely, Árpád Beszédes, Dávid Tengeri, Gergő Balogh, and Tibor Gyimóthy. 2019. Code coverage differences of java bytecode and source code instrumentation tools. *Software Quality Journal* 27, 1 (2019), 79–123.

[24] Hiroshi Inoue, Hiroshige Hayashizaki, Peng Wu, and Toshio Nakatani. 2011. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the CGO*.

[25] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu. 2018. RedDroid: Android application redundancy customization based on static analysis. In *Proceedings of the ISSRE*. 189–199.

[26] Y. Jiang, D. Wu, and P. Liu. 2016. JRed: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the COMPSAC*. 12–21.

[27] Y. Jiang, C. Zhang, D. Wu, and P. Liu. 2016. Feature-based software customization: Preliminary analysis, formalization, and methods. In *Proceedings of the HASE*. 122–131.

[28] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the ESEC/FSE*. 556–566.

[29] Peter Kampstra. 2008. Beanplot: A boxplot alternative for visual comparison of distributions. *Journal of Statistical Software* 28, 1 (2008), 1–9.

[30] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. 2019. Configuration-driven software debloating. In *Proceedings of the EuroSec Workshop*.

[31] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for static analysis of java reflection: Literature review and empirical study. In *Proceedings of the ICSE*. 507–518.

[32] Nan Li, Xin Meng, Jeff Offutt, and Lin Deng. 2013. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools. In *Proceedings of the ISSRE*. 380–389.

[33] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java Virtual Machine Specification*. Pearson Education.

[34] Christian Macho, Stefanie Beyer, Shane McIntosh, and Martin Pinzger. 2021. The nature of build changes. *Empirical Software Engineering* 26, 3 (2021), 32.

[35] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. 2009. Four trends leading to Java runtime bloat. *IEEE Software* 27, 1 (2009), 56–63.

[36] N. Mitchell, E. Schonberg, and G. Sevitsky. 2010. Four trends leading to java runtime bloat. *IEEE Software* 27, 1 (2010), 56–63.

[37] Girish Mururu, Chris Porter, Prithayan Barua, and Santosh Pande. 2019. Binary debloating for security via demand driven loading. arXiv:1902.06570. Retrieved from https://arxiv.org/abs/1902.06570.

[38] Khanh Nguyen, Kai Wang, Yingyi Bu, Lu Fang, and Guoqing Xu. 2018. Understanding and combating memory bloat in managed data-intensive systems. *ACM Transactions on Software Engineering and Methodology* 26, 4 (2018), 41 pages.

[39] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the ESEC/FSE*. 268–278.

[40] Vijay Krishna Palepu, Guoqing Xu, and James A. Jones. 2017. Dynamic dependence summaries. *ACM Transactions on Software Engineering and Methodology* 25, 4, (2017), 41 pages.

[41] S. Ponta, W. Fischer, H. Plate, and A. Sabetta. 2021. The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application. In *Proceedings of the ICSME*. 555–558.

[42] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the USENIX Security Symposium*. 1733–1750.

[43] Anh Quach, Rukayat Erinfolami, David Demicco, and Aravind Prakash. 2017. A multi-OS cross-layer study of bloating in user programs, kernel, and managed execution environments. In *Proceedings of the FEAST Workshop*. 65–70.

[44] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically debloating containers. In *Proceedings of the ESEC/FSE*. 476–486.

[45] Ulrik P. Schultz, Julia L. Lawall, and Charles Consel. 2003. Automatic program specialization for java. *ACM Transactions on Programming Languages and Systems* 25, 4 + (2003), 452–499.

[46] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application specialization for code debloating. In *Proceedings of the ASE*. 329–339.

[47] César Soto-Valero. 2022. Open-science repository for the experiments with JDBL. Retrieved November 21st, 2021 from https://github.com/castor-software/jdbl-experiments.

[48] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The emergence of software diversity in maven central. In *Proceedings of the MSR*. 333–343.

[49] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated java dependencies. In *Proceedings of the ESEC/FSE*. 1021–1031.

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

[51] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the soundness of call graph construction in the presence of dynamic language features—a benchmark and tool evaluation. In *Proceedings of the Programming Languages and Systems*. Sukyoung Ryu (Ed.), 69–88.

[52] Dávid Tengeri, Ferenc Horváth, Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. 2016. Negative effects of bytecode instrumentation on java source code coverage. In *Proceedings of the SANER*. 225–235.

[53] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. 1999. Practical experience with an application extractor for java. In *Proceedings of the OOPSLA*. 292–305.

[54] 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* 24, 6 (2002), 625–666.

[55] H. C. Vázquez, A. Bergel, S. Vidal, J. A. Díaz Pace, and C. Marcos. 2019. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and Software Technology* 107 (2019), 18–29.

[56] Niklaus Wirth. 1995. A plea for lean software. *IEEE Computer* 28, 2 (1995), 64–68.

[57] Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso. 2019. Identifying features of android apps from execution traces. In *Proceedings of the MOBILESoft*. 35–39.

[58] Guoqing Xu. 2013. CoCo: Sound and adaptive replacement of java collections. In *Proceedings of the ECOOP*. 1–26.

[59] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. 2014. Scalable runtime bloat detection using abstract dynamic slicing. *ACM Transactions on Software Engineering and Methodology* 23, 3 (2014), 50 pages.

[60] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the FoSER*. 421–426.

[61] Qian Yang, J. Jenny Li, and David M. Weiss. 2009. A survey of coverage-based testing tools. *The Computer Journal* 52, 5 (2009), 589–597.

[62] Hao Zhong and Hong Mei. 2019. An empirical study on API usages. *IEEE Transactions on Software Engineering* 45, 4 (2019), 319–334.

[63] Andreas Ziegler, Julian Geus, Bernhard Heinloth, Timo Hönig, and Daniel Lohmann. 2019. Honey, I shrunk the ELFs: Lightweight binary tailoring of shared libraries. *ACM Transactions on Embedded Computing Systems* 18, 5 (2019), 1–18.

Received 8 December 2021; revised 19 May 2022; accepted 1 June 2022