

See discussions, stats, and author profiles for this publication at: <https://www.researchgate.net/publication/343759281>

Trace-based Debloat for Java Bytecode

Preprint · August 2020

CITATIONS

0

READS

22

4 authors, including:



César Soto-Valero

KTH Royal Institute of Technology

28 PUBLICATIONS 40 CITATIONS

[SEE PROFILE](#)



Thomas Durieux

KTH Royal Institute of Technology

34 PUBLICATIONS 550 CITATIONS

[SEE PROFILE](#)



Benoit Baudry

National Institute for Research in Computer Science and Control

295 PUBLICATIONS 4,729 CITATIONS

[SEE PROFILE](#)

Some of the authors of this publication are also working on these related projects:







DIVERSIFY [View project](#)



Model-based Testing [View project](#)

Trace-based Debloat for Java Bytecode

César Soto-Valero , Thomas Durieux , Nicolas Harrand , and Benoit Baudry 
KTH Royal Institute of Technology, Stockholm, Sweden
 Email: {cesarsv, tdurieux, harrand, baudry}@kth.se

Abstract—Software bloat is code that is packaged in an application but is actually not used and not necessary to run the application. The presence of bloat is an issue for software security, for performance, and for maintenance. In recent years, several works have proposed techniques to detect and remove software bloat. In this paper, we introduce a novel technique to debloat Java bytecode through dynamic analysis, which we call trace-based debloat. We have developed JDBL, a tool that automates the collection of accurate execution traces and the debloating process. Given a Java project and a workload, JDBL generates a debloated version of the project that is syntactically correct and preserves the original behavior, modulo the workload. We evaluate the feasibility and the effectiveness of trace-based debloat with 395 open-source Java libraries for a total 10M+ lines of code. We demonstrate that our approach significantly reduces the size of these libraries while preserving the functionalities needed by their clients.

Index Terms—Software Bloat, Dynamic Analysis, Program Specialization, Build Automation, Software Maintenance

1 INTRODUCTION

SOFTWARE systems have a natural tendency to grow over time, both in terms of size and complexity [1], [2], [3]. A part of this growth comes with the addition of new features or different types of patches. Another part is due to the potentially useless code that accumulates over time. This phenomenon, known as software bloat, is becoming more prevalent with the emergence of large software frameworks [4], [5], [6], and the widespread practice of code reuse [7], [8]. Software debloat consists of automatically removing unnecessary code [9]. This poses several challenges for code analysis and transformation: determine the bloated parts of the software system [10], [11], [12], remove this part while keeping the integrity of the system, produce a debloated version of the system that can still run and provide useful features. In this context, the problem of effectively and safely debloating real-world applications remains a long-standing software engineering endeavor.

Previous works on software debloat have proposed different techniques, each of them tailored to a specific language. Significant efforts have been devoted to software debloat for C/C++ executable binaries. In this context, debloat starts from a program in which dependencies are compiled and linked statically [13], [14], [12]. Debloat approaches for Java are scarce in the literature, and rely on static analysis to detect unreachable code [15], [16]. These techniques are challenged by the dynamic features of the language, such as type-induced dependencies [17], dynamic class loading [18], and reflection [19]. In addition, static analysis techniques are conservative and do not remove *unused code* [20], [16], i.e., the parts of an application that can be reached statically but are not executed at run-time, within a specific period, in a production environment.

In this paper, we propose a novel software debloat technique to remove unused Java bytecode: *trace-based debloat*. The core novelty consists of steering the debloat process with information obtained from the collection of execution traces. This technique aims to capture bytecode usage information by monitoring the dynamic behavior of the system. Its automatable nature allows us to scale the debloat technique to large and diverse software projects, without any additional configuration. We implement this approach in a tool called JDBL, the Java

DeBLoat tool designed to debloat Java projects configured to build with Maven.

JDBL is the first software tool to debloat Java bytecode that combines trace collection, bytecode removal, and build validation. JDBL is designed around three debloat phases. First, JDBL addresses the challenge of spotting unnecessary code while keeping the program cohesive. It leverages diverse code coverage tools to collect a set of accurate execution traces for debloating. This process involves executing the Maven project with an existing workload and monitoring its behavior at run-time through dynamic analysis. Second, JDBL modifies the bytecode to remove unnecessary code, i.e. code that has not been traced in the first phase. Bytecode removal is performed on the project as well as on the whole tree of third-party dependencies. Third, JDBL validates the debloat by executing the workload and verifying that the debloated project preserves the behavior of the original project.

We run JDBL on a curated benchmark of 395 open-source Java libraries, to evaluate its correctness and effectiveness. Our results show that JDBL is capable of automatically debloating 311 (78.7%) real-world Java libraries, and preserving the correctness of 220 (70.7%) of these libraries. We provide quantitative evidence of the massive presence of unnecessary code in software artifacts: 62.2% of classes in the libraries are bloated. The removal of this bloated code significantly reduces bytecode size: JDBL saves 68.3% of the libraries' disk space, which represents a mean reduction of 25.8% per library.

In order to further validate the relevance of trace-based debloat, we perform a second set of experiments with projects that use the libraries that we debloated with JDBL. This is the first time that the debloat results are not only evaluated with respect to the debloated subjects, but also on their clients. The goal is to demonstrate that JDBL produces debloated artifacts that still provide relevant functionalities. Our experiment shows firstly that the compilation of 957/1,001 (95.6%) clients are not affected by the debloat of the library. Secondly, that the behavior of the test suite of 229/283 (80.9%) clients is preserved.

In summary, this paper makes the following contributions:

- The conceptual foundation of trace-based debloat for Java: a practical approach to debloat software through the collection of execution traces.

- A novel open-source tool, JDBL, which is capable of automatically producing debloated versions of Java artifacts.
- The largest empirical study of software debloat powered by an original protocol and an experimental framework that automates the evaluation of JDBL on 395 libraries hosted on GitHub.
- A novel assessment of the impact of debloat for library clients, performed with 1,370 clients of the 220 libraries that JDBL successfully debloats.

The rest of this paper is structured as follows: Section 2 presents a motivating example together with the definition of trace-based debloat. Sections 3, 4, and 5 present the design and evaluation of JDBL. Sections 6 and 7 present the threats to validity and the related work.

2 TRACE-BASED DEBLOAT

In this section, we illustrate the impact of software bloat in the context of Java applications and introduce our novel concept of trace-based debloat.

2.1 Motivating example

Figure 1 illustrates the architecture 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-configuration2¹ library.

The commons-configuration2 library provides a generic interface which enables any Java application to read configuration data files from a variety of source formats, e.g., properties, json, xml, yaml, etc. In our example, JPROJECT accepts two types of configuration formats, properties and json files, and uses commons-configuration2 to parse them. To read a properties file, it uses the Configurations helper class from the library. To read a json file, it instantiates a FileBasedConfigurationBuilder for reading the configuration file with the class JSONConfiguration, a specialized hierarchical configuration class in the library that parses json files.

Although the commons-configuration2 library provides an interface to handle a diverse set of file formats, JPROJECT only needs to use two of them. Therefore, all the rest of the functionalities provided by the library are unnecessary for the project. Still, all the classes of the library must be added in the classpath of JPROJECT, as well as all the run-time dependencies of the library as well. Figure 1 shows this phenomenon: only the API members in green are necessary for the project, whereas all the code that belongs to the components in red, which includes all the functionalities for parsing other types of files than properties and json, are bloated with respect to JPROJECT. This represents a considerable number of unnecessary classes from the commons-configuration2 library included in the project. In addition, by declaring a dependency towards commons-configuration2, JPROJECT has to include the classes of a total of 7 transitive dependencies in its classpath. Some classes in the dependency B are used for the two file formats supported by JPROJECT, and parsing json files requires functions from dependencies C and D. Notice that all the classes in the dependencies E, F, G, and H, are bloated with respect to JPROJECT.

In summary, a small part of JPROJECT is in charge of handling configuration files. The project declares a dependency to a third-party library that facilitates this task. Consequently, the

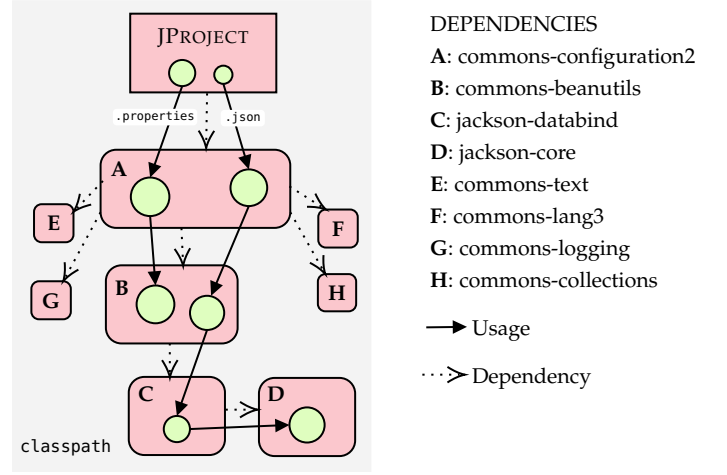


Fig. 1: Typical code reuse scenario in the Java ecosystem. The Java project, JPROJECT, uses functionalities provided by the library commons-configuration2. The square node represents the bytecode in JPROJECT, rounded nodes are dependencies, circles inside the nodes are API members called in the bytecode of libraries and dependencies. Used elements are in green, bloated elements are in red.

project imports additional code that is not necessary. The Java packaging mechanism includes all the unnecessary bytecode in the JAR file of the application at each release, regardless of the functionalities that the project actually uses.

This simplified example illustrates the typical 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 are packaged in the project’s JAR; the existence of disjoint execution paths makes Java projects susceptible to include unnecessary functionalities from libraries declared as dependencies.

In this paper, we focus on removing unnecessary functionalities from compiled Java projects and their dependencies in order to mitigate software bloat. This involves the detection and removal of the reachable bytecode instructions that do not provide any functionalities to the project at run-time, both in its own classes and the classes of its dependencies.

2.2 Definitions

We define the key concepts used throughout this work and build upon these to introduce the definition of trace-based debloat.

Definition 1. Maven project: A Maven project is a collection of source code files and configuration files organized to build with Maven.

Maven projects must include a particular build file, called pom.xml, which defines all the dependencies and build instructions necessary to produce an artifact. Artifacts are typically packaged as JAR files. These files are then deployed to a binary code repository to facilitate reuse by other projects. In this work, the compiled Maven project is referred to as a *library*, and the project that reuses the library is called a *client*.

Definition 2. Input space: The input space of a compiled Maven project is the set of all valid inputs for its public Application Programming Interface (API).

1. <https://commons.apache.org/proper/commons-configuration2>

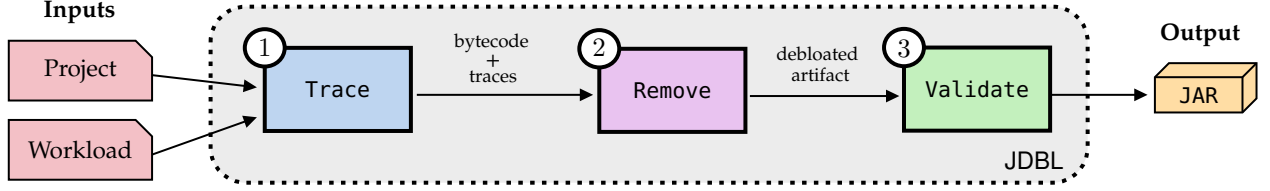


Fig. 2: High-level architecture of JDBL, a three-phase trace-based debloat tool: Trace, Remove, and Validate.

Maven projects provide API members, abstracting implementation details to facilitate external reuse. Libraries generally provide public members for external reuse. However, there exist other dynamic reuse mechanisms that can be utilized by Java clients (e.g., through reflection, dynamic proxies, or the use of unsafe APIs).

Definition 3. Workload: A workload is a set of valid inputs belonging to the input space of a compiled Maven project.

Workloads are particularly useful for performing dynamic analysis. For example, workloads are used to detect distinct execution paths in software applications, e.g., through profiling, and observability tasks. This type of technique aims at using monitoring tools to analyze how the application reacts to different workloads at run-time.

Definition 4. Execution trace: An execution trace is a sequence of calls between bytecode instructions in a compiled Maven project, obtained as a result of executing the project with a valid workload.

Given a valid workload for a project, one can obtain dynamic information about the program’s behavior by collecting execution traces. In this work, we consider a trace as a sequence of calls, at the level of classes and methods, in compiled Java classes. These traces include the bytecode of the project itself, as well as the classes and methods in third-party libraries.

Definition 5. Trace-based debloat: Given a project and an execution trace collected when running a specific workload on the project, trace-based debloat consists of removing the bytecode constructs that are not necessary to run the workload. Trace-based debloat takes a project and workload as input and produces a valid compiled Java project as output. The generated project, called the debloated project, is executable and has the same behavior as the original, modulo the workload.

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 $\mathcal{F}_{\mathcal{P}} \subset \mathcal{S}_{\mathcal{P}}$. Trace-based debloat transforms \mathcal{P} into a correct program \mathcal{P}' , where $|\mathcal{S}_{\mathcal{P}'}| < |\mathcal{S}_{\mathcal{P}}|$ and \mathcal{P}' preserves the same behavior as \mathcal{P} when executing the workload.

In the next section, we present the details of JDBL, the first end-to-end tool to perform automated trace-based debloat for Java bytecode.

3 JDBL

In this section, we describe our main technical contribution in detail: an end-to-end pipeline for performing automatic trace-based debloat of Java programs, implemented in the JDBL tool.

Figure 2 illustrates the main architecture of JDBL. It receives as input a Java project that builds correctly with Maven and a workload that exercises the project. JDBL yields as output a debloated version of the packaged project that builds correctly and preserves all the functionalities necessary to run that particular workload. The debloat procedure consists of three

main phases: the Trace phase for collecting usage information based on dynamic execution traces, the Remove phase for modifying the bytecode of the artifact based on the traces, and the Validate phase for assessing the correctness of the debloated artifact (i.e., syntactic and semantic correctness).

Algorithm 1 details the end-to-end debloat procedure of JDBL. The algorithm contains three subroutines, corresponding to each debloating phase. The Trace phase (lines 1 to 11) starts compiling the source files of the project \mathcal{P} that is given as input, resolving all its direct and transitive dependencies \mathcal{D} , and adding the bytecode to the classpath CP of the project (line 1). JDBL proceeds to instrument the whole bytecode contained in CP (line 2), and initializes a map USG of pairs $class \rightarrow [method_1, method_2, \dots, method_n]$, in which the classes and methods used when executing \mathcal{W} will be stored (line 3). Then, JDBL executes \mathcal{W} on the instrumented bytecode: each element w in the workload is executed (line 5), and the classes and methods executed are added to USG (lines 8 and 11). Note that the main challenge here is to get an accurate implementation of the function `isExecuted`, which is in charge of determining the actual usage status of classes and methods. Then, JDBL moves forward to the Remove phase (lines 12 to 18). Each class and method in the original bytecode that is not traced is removed (lines 14 and 18). At this step, JDBL can be configured to stop the debloat at the class level, or can go further with the removal of bloated methods. In the Validate phase (lines 19 to 23), the workload \mathcal{W} is executed again on the original version of \mathcal{P} , to collect the program’s original behavior in the variable OBS (line 19). Then, JDBL performs two checks in line 20: a syntactic check that passes if the build of the debloated program is successful, and a behavioral check that passes if the debloated artifact preserves the original behavior of \mathcal{P} . Finally, the bytecode is packaged as a JAR file and returned (lines 22 and 23).

In the following subsections, we describe each of the three phases in more detail. We discuss the key technical challenges for trace-based debloat in Java and how JDBL overcomes them through its implementation.

3.1 Trace phase

The goal of the Trace phase in JDBL is to collect a set of execution traces that are both accurate and complete, capturing the dependencies, classes, and methods that are used during the execution of a Java application. The Trace 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. For example, the workload can be a set of test scenarios, a production workload that can be replayed. The output of the Trace phase is the original, unmodified, bytecode of the compiled sources and a set of execution traces that includes the minimal set of classes and methods required to execute the workload.

Algorithm 1: Trace-based debloat procedure for Java.

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} .

```

// ① Trace phase
1   $CP \leftarrow \text{compileSources}(\mathcal{S}, \mathcal{P}) \cup \text{getDependencies}(\mathcal{D}, \mathcal{P})$ ;
2   $INST \leftarrow \text{instrument}(CP)$ ;
3   $USG \leftarrow \emptyset$ ;
4  foreach  $w \in \mathcal{W}$  do
5       $\text{execute}(w, INST)$ ;
6      foreach  $class \in INST$  do
7          if  $\text{isExecuted}(class)$  then
8               $USG \leftarrow \text{addKey}(class, USG)$ ;
9              foreach  $method \in class$  do
10                 if  $\text{isExecuted}(method)$  then
11                      $USG \leftarrow \text{addVal}(method, class, USG)$ ;

// ② Remove phase
12 foreach  $class \in CP$  do
13     if  $class \notin \text{keys}(USG)$  then
14          $CP \leftarrow CP \setminus class$ ;
15     else
16         foreach  $method \in class$  do
17             if  $method \notin \text{values}(class, USG)$  then
18                  $CP \leftarrow CP \setminus method$ ;

// ③ Validate phase
19  $OBS \leftarrow \text{execute}(\mathcal{W}, \mathcal{P})$ ;
20 if  $! \text{buildSuccess}(CP) \mid \text{execute}(\mathcal{W}, CP) \neq OBS$  then
21     return ALERT;
22  $\mathcal{P}' \leftarrow \text{package}(CP)$ ;
23 return  $\mathcal{P}'$ ;

```

3.1.1 Leveraging coverage techniques for trace collection

To collect accurate usage information in the form of execution traces for a Maven project, JDBL leverages JaCoCo,² a modern code coverage tool that operates on the bytecode. JaCoCo is a mature, actively maintained, efficient, and widely adopted coverage tool for Java that provides the necessary infrastructure for bytecode instrumentation, monitoring, and trace collection. Similar to other code coverage products that rely on bytecode transformations [21], JaCoCo is originally designed to perform the following steps when computing the code coverage of a project under test:

- 1) *Bytecode instrumentation.* The bytecode is enriched by the injection of probes at particular locations of the control flow, depending on the granularity level of the coverage.
- 2) *Artifact execution.* The instrumented bytecode is executed in order to collect the information on which probes are activated at run-time.
- 3) *Coverage report.* The activated regions of the bytecode are mapped with the source code, and a coverage report is given to the user.

JDBL leverages the bytecode instrumentation and the artifact execution capabilities of JaCoCo to identify the bytecode elements that are executed by a particular workload. One major technical contribution of JDBL is to adapt the instrumentation and observability capabilities of JaCoCo to serve its debloating objective. These adaptations address two shortcomings of JaCoCo for debloating: 1) extend coverage monitoring to the complete dependency tree, and 2) augment the coverage capabilities in order to build a more accurate trace than the default trace obtained with JaCoCo. In the following sections, we explain how we overcome these technical challenges in JDBL.

3.1.2 Code coverage for the complete dependency tree of a Java project

JDBL is designed to trace and remove the unnecessary bytecode in the compiled project as well as in its dependencies. To do so, JDBL extends the coverage information provided by JaCoCo to the level of dependencies down the dependency tree of the project. This requires modifying the way JaCoCo interacts with Maven during the artifacts' build life-cycle.

Resolving all the dependencies from external repositories is a fundamental step before performing bytecode instrumentation. JDBL relies on the automated build infrastructure of Maven for the compilation of the Java project and the resolution of its dependencies. Maven provides dedicated plugins for fetching and storing all the direct and transitive dependencies of the project. Therefore, JDBL relies on the Maven dependency management mechanisms, which are all based on the use of a *pom.xml* file that declares the direct dependencies of the project towards third-party libraries. These dependencies are JAR files hosted in external repositories (e.g., Maven Central³).

Only dependencies with *runtime* scope are packaged by Maven at the end of the building process. Therefore, JDBL does not need to instrument dependencies required for other build tasks (e.g., JUnit for testing). Once the dependencies have been downloaded and stored locally, JDBL compiles the Java sources and unpacks all the bytecode of the project and its dependencies into the same local directory. Then, it proceeds to inject probes at the beginning and end of the Java bytecode methods. JDBL instruments the bytecode of classes and dependencies in off-line mode, before the execution and trace-collection tasks.

As for the trace collection, during the execution of the instrumented application, JaCoCo receives an event every time the execution hits an injected probe. JDBL executes the application with the given workload, captures the covered classes and methods, and combines them into a single execution trace. In contrast with code coverage tools, the report of JDBL is not the covered source code, but the set of bytecode elements in the execution trace. This set is the output of the Trace phase.

3.1.3 Complete coverage reports

The collection of complete execution traces involves several challenges related to compilation and bytecode instrumentation. First, the bytecode instrumentation must be safe, i.e., it should not alter the behavior of the application at run-time. Second, the trace must be complete, i.e., all the bytecode that is necessary to execute the workload should be reported as covered. Third, the instrumentation must be efficient, and the injected probes should not impose a significant performance overhead during the execution of the application.

From the challenges described above, the second one is the most relevant for our debloat purposes: missing a class in

2. <https://www.eclemma.org/JaCoCo>

3. <https://mvnrepository.com/repos/central>

```

1 public class Foo {
2     public void m1() {
3         m2();
4     }
5     public void m2() {
6         throw new IllegalArgumentException();
7     }
8 }
9
10 public class FooTest {
11     @Test(expected = IllegalArgumentException.class)
12     public void test() {
13         Foo foo = new Foo();
14         foo.m1();
15     }
16 }

```

Listing 1: Example of an incomplete coverage report given by JaCoCo. The method `m2` is executed when running the method `test` in `FooTest`. However, this method is not considered as covered by JaCoCo.

the trace means that JDBL will remove a necessary piece of bytecode, causing the debloated application to be syntactically or semantically incorrect. In this regard, we observe that the completeness of execution traces may be affected due to two main reasons. First, different coverage tools have divergent coverage strategies to handle the variety of existing bytecode constructs [22]. Consequently, these tools provide different reports for the same build setup, posing a challenge for debloat usage. Second, 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 [14]. In this case, it is not possible for coverage tools to collect information missing in the original bytecode. The following examples illustrate these limitations.

Listing 1 shows an example of an incorrect coverage report caused by a design limitation of JaCoCo. The method `m2` (lines 5 to 7) is not considered as covered by JaCoCo. However, it is clear that, if we remove it, the test in class `FooTest` fails (lines 11 to 15). The reason for this unexpected behavior is that the JaCoCo probe insertion strategy does not consider implicit exceptions thrown from invoked methods. These types of exceptions are subclasses of the classes `RuntimeException` and `Error`, and are expected to be thrown by the JVM itself. 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 considered as not covered because JaCoCo misses the instrumentation probe on the exit point of the method. To mitigate this issue, JaCoCo adds an additional probe between the instructions of two lines whenever the subsequent line contains at least one method invocation. This limits the effect of implicit exceptions from method invocations to single lines of source. However, the approach only works for class files compiled with debug information (line numbers) and does not consider implicit exceptions from other instructions than method invocations (e.g., `NullPointerException` or `ArrayIndexOutOfBoundsException`). In conclusion, JaCoCo does not consider methods with a single-line invocation to other methods that throw exceptions as covered, which has a negative impact on the precision of the execution trace in this particular scenario.

Listing 2 shows an example of incorrect coverage due to the inability of JaCoCo to trace 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 12. However, `FooEnum` is not covered according to JaCoCo. The reason is that, in Java, every enumerated type im-

```

1 public enum FooEnum {
2     MAGIC("forty two");
3     public final String label;
4     private FooEnum(String label) {
5         this.label = label;
6     }
7 }
8
9 public class FooEnumTest {
10     @Test
11     public void test() {
12         assertEquals("forty two", FooEnum.valueOf("MAGIC").label);
13     }
14 }

```

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 `FooEnum` is reported as not covered.

PLICITLY extends the class `java.lang.Enum`, which implements the methods `Enum.values()` and `Enum.valueOf()`. These are compiler-generated methods, i.e., the `javac` compiler injects them at compile-time. These methods are not traced by various coverage tools, which degrades the overall completeness of the produced execution trace.

Listing 3 shows another example of an incorrect coverage report. The variable `MAGIC`, initialized with a final static integer literal in line 2, is used in the `FooTest` class as `Foo.MAGIC` (line 8). However, the class `Foo` is not detected as covered by any code coverage tool based on bytecode instrumentation. The cause is a bytecode optimization implemented in the `javac` compiler, which is designed to inline constants at compilation time. Listing 4 shows the bytecode generated after compiling the sources of the `FooTest` class in Listing 3. As we observe in lines 4 to 5, the value of the constant `MAGIC` is directly substituted by its integer value, and hence the reference to the class `Foo` is lost in the bytecode. Note that, if we remove the class `Foo`, the program will not compile correctly.

To overcome the limitations of JaCoCo, JDBL augments its coverage capabilities through the combination of information harvested by diverse coverage and monitoring tools. This approach allows consolidating the collection of a more complete and accurate execution trace. For example, we develop *Yajta*,⁴ a customized tracing agent that handles the case presented in Listing 1 by inserting the probe at the beginning of methods, including the default constructor. We also combine the coverage reports of *JCov*,⁵ a mature coverage tool that handles the compiled-generated method presented in Listing 2. To obtain the list of classes that are loaded dynamically, we parse the output of the JVM class loader, this approach facilitates the handling of corner cases such as the example presented in Listing 3. In summary, JDBL relies on the diversity of implementations of the following coverage tools in order to consolidate the coverage report for debloat:

- *JaCoCo*: A mature and widely used Java code coverage tool. JaCoCo relies on ASM⁶ for performing bytecode instrumentation. It supports statements as well as branch coverage.
- *JCov*: A pure Java implementation of a code coverage tool. JCov 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.

4. <https://github.com/castor-software/yajta>

5. <https://wiki.openjdk.java.net/display/CodeTools/jcov>

6. <https://asm.ow2.io>

```

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 name();
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.

- *Yajta*: A customized tracing agent developed by us to cover some corner cases, e.g., synthetic, and compiled generated methods. Yajta is engineered to work offline and uses Javassist⁷ for bytecode instrumentation.
- *JVM class loader*: The JVM dynamically links classes before executing them. The `-verbose:class` option of the JVM enables logging of class loading and unloading. We use this mode as a complement of the bytecode coverage tools for collecting the classes loaded by the JVM during the workload execution.

3.2 Remove phase

The goal of the Remove phase is to eliminate all the unnecessary bytecode instructions in the original class files of the project. Once JDBL analyzes the traces collected in the Trace phase, it proceeds to remove the non-traced class files in two passes. First, all the unused classes are directly removed from the classpath of the project. After this pass, all the bytecode of the methods in the classes that belong to the collected execution traces are visited and, for those that were not traced, i.e., methods that are not part of the execution path, JDBL replaces the body of the method to throw in an `UnsupportedOperationException`. 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.

In summary, JDBL removes the following bloated bytecode elements from Java artifact:

- *Bloated methods*. A method is considered bloated if it is not triggered after executing the artifact with a given workload.
- *Bloated classes*. A class is considered bloated if it has not been instantiated or called via reflection and none of its fields or methods are used.
- *Bloated dependencies*. A dependency is considered bloated if none of its classes or methods are used when executing the artifact with a given workload. In this work, we refer, particularly to Maven dependencies.

7. <https://www.javassist.org>

3.3 Validate phase

The goal of the Validate phase is to assess the syntactic and semantic correctness of the debloated artifact with respect to the workload utilized. This phase is a fundamental step before packaging the debloated JAR because it helps to detect errors introduced by JDBL during the previous debloat phases.

To assess syntactic correctness, JDBL verifies the integrity of the resulting bytecode in the debloated version. This implies checking the validity of the bytecode that the JVM has to load at run-time, and also checking that no dependencies or other resources were incorrectly removed from the classpath of the Maven project. This validation is performed using the Maven tool stack, which includes dedicated plugins to perform several validation checks at each step of the build process. 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, JDBL executes the same workload used for the debloat and asserts that the original behavior of the artifact is not modified. In other words, it treats the workload as an oracle to check the correctness of the debloated application. This guarantees behavior preservation for the necessary features in the debloated artifact before the final bytecode packaging task.

3.4 Implementation details

Our key technical contributions consist of extending JaCoCo to build accurate traces over the complete dependency tree of a Java project and in implementing a novel, precise code removal procedure that executes throughout the software build pipeline. We have embedded these contributions into JDBL.

The core implementation of JDBL consists in the orchestration of our novel techniques with mature code coverage tools, and bytecode transformation techniques. The trace-based debloat process is integrated into the different Maven building phases. Maven 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 navigate through all the JDBL debloat phases during the project build life-cycle.

JDBL gathers direct and transitive dependencies by using the `maven-dependency` 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 3.1.2. The instrumentation of methods and probe insertion is performed by integrating JaCoCo, JCov, Yajta, and the JVM class loader, as described in Section 3.1.3. For bytecode analysis, the collection of non-removable classes, and the whole Remove phase (Section 3.2), we rely on ASM, a lightweight, and mature Java bytecode manipulation and analysis framework.

JDBL is implemented as a multi-module Maven project with a total of 5K lines of code written in Java. It can be used as a Maven plugin that executes during the `package` Maven phase. Thus, JDBL 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, as shown in Listing 5. 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>.


```

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

```

Listing 5: Using JDBL as a Maven plugin, which facilitates its execution and integration into the project’s build life-cycle.

4 EMPIRICAL STUDY

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

4.1 Research Questions

To evaluate our trace-based debloat approach, we study the *correctness*, *effectiveness*, and *impact* of JDBL. Our study is guided by the following research questions:

- RQ1.** *To what extent can a generic, fully automated trace-based debloat technique produce a debloated version of Java projects?*
- RQ2.** *To what extent do the debloated versions preserve their original behavior?*

RQ1 and RQ2 focus on assessing the *correctness* of JDBL. 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 debloated in the compiled projects and their dependencies?*
- RQ4.** *What is the impact of trace-based debloat approach on the size of the packaged artifacts?*

RQ3 and RQ4 investigate the *effectiveness* of JDBL at producing a smaller artifact by removing the unnecessary bytecode. We measure this effectiveness with respect to the amount of debloated dependencies, classes, and methods, as well as with the reduction of the size of the bundled JAR file.

- RQ5.** *To what extent do debloated libraries break the compilation of their clients?*
- RQ6.** *To what extent do debloated libraries affect the behavior of their clients?*

In RQ5 and RQ6, we go one step further than any previous work on software debloating and investigate how trace-based debloat of Java libraries impacts the clients of these libraries. Our goal is to determine the ability of dynamic traces at capturing the behaviors that are relevant for the users of the debloated libraries.

4.2 Study Subjects

In this section, we describe the methodology that we follow to construct a new dataset of open-source Maven Java projects extracted 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 allows us to determine if our trace-based debloat approach, implemented in JDBL, 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 benchmark is performed in 5 steps. First, we identified the 147,991 Java projects on GitHub that have at least five stars. We use the stars as an indicator of interest. Second, we selected the 34,560 (23.4%) single-module Maven projects. We chose single-module projects because they generate a single JAR. We identify them by listing all the files for each project and consider the projects that have a single Maven build configuration file, (i.e., `pom.xml`). Third, 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`. During this step, we identify 155 (0.4%) libraries, and 25,557 (73.9%) clients that use 2,103 versions of the libraries. Fourth, we identify the commit associated with the version of the libraries, e.g., `commons-net:3.4` is defined in the commit `74a2282b7e4c6905581f4f1b5a2ec412310cd5e7`. 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. As the fifth step, 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 this point, we consider each library version as a unique library to improve the clarity.

Table 1 summarizes the descriptive statistics of the dataset. The number of LOC and the coverage are computed with JaCoCo. Our data contains 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. The clients have 211,116 test cases that cover 20.24% of the 140,910,102 LOC. The dataset and the scripts to generate the dataset are available in our experiment repository: <https://github.com/castor-software/jdbl-experiments/tree/master/dataset>.

4.3 Protocol

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

Trace-based debloat requires running the program to be debloated. For our experiments, we use the test suite of projects as a workload. Test suites are widely available while obtaining a realistic workload for hundreds of libraries is extremely difficult. Our choice of test suites is also motivated by novelty: as far as we know, no previous work uses the test suite for debloating. The third motivation for test suites is to use JDBL directly in the build process and deploy the debloated version, which can be directly used by the clients without additional steps. Therefore, we investigate the correctness and effectiveness of JDBL using all the existing tests as entry-points to execute the library and produce the traces that steer the debloating process.

Our experiments are based on performing trace-based debloat on 395 different versions of 94 libraries, with their test suites as workload. An original step in our debloat experimental protocol consists of further validating the utility of the debloated libraries with 2,874 clients. This way, we check the extent to which trace-based debloat preserves the elements that are required to compile and successfully run the test suites of the clients. In our benchmark, the libraries have a mean coverage of 80.83%, where the clients only have a mean coverage of 20.24%.

TABLE 1: Descriptive statistics of the studied libraries and their associated clients.

		Min	1st Qu.	Mean	3rd Qu.	Max	Avg.	Total
395 Libraries	# 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
	Coverage	0.1 %	61.7 %	80.8 %	89.8 %	100.0 %	73.7 %	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
	Coverage	0.0 %	2.1 %	20.24 %	57.7 %	100.0 %	31.4 %	N.A

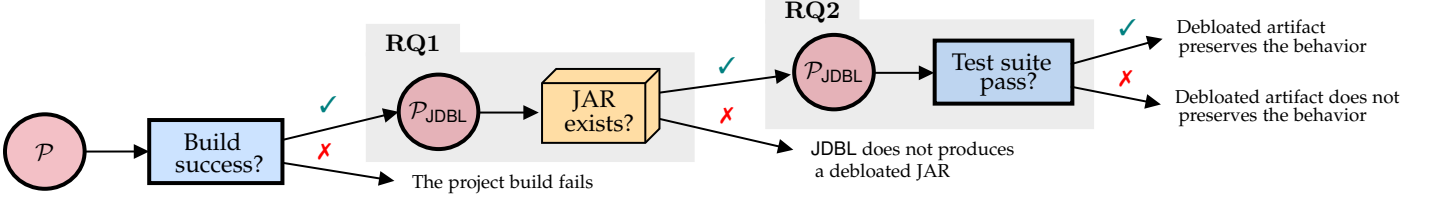


Fig. 3: Pipeline of our experimental protocol to answer RQ1 and RQ2.

4.3.1 JDBL 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 mentioned in Section 3.4, JDBL is implemented as a Maven plugin, hence 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 library.* During the first step, we build the original library (i.e., using `mvn package`) to ensure that it builds correctly, i.e., it does not contain any failing test case before executing JDBL, and to generate a JAR file that contains all the binaries of the project. This verification is similar to the last step of the creation of the dataset (see Section 4.2). Still, the major difference with this previous step is that we configure the project to generate a JAR file. This change of configuration may be in conflict with the project configuration and therefore fail. A failing test case could introduce incorrect or limited execution traces and, therefore, negatively impact the effectiveness of JDBL. At the end of the execution of the test suite, a JAR file is produced, which contains the bytecode of the library and all its dependencies. We also store the reports of 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 JDBL.* The second step consists of adding JDBL as a plugin inside the Maven configuration (`pom.xml`) and resetting the configuration of the surefire Maven plugin.⁸ We reset the surefire to ensure that its original configuration is not in conflict with the execution of the Trace phase of JDBL. A manual configuration of JDBL could prevent this problem, but, in order to scale-up the evaluation, we decided to standardize the execution for all the libraries.
- 3) *Execute JDBL.* The third and final step is to execute JDBL on the library. This consists in running `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 a i9-10900K CPU (16 cores) and 64 GB of RAM. 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 debloat 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 use JDBL commit SHA: `c57396a5739e6ac3b0fa434342eb57b6f945914b`. The execution framework is publicly available on GitHub: `https://github.com/castor-software/jdbl-experiments`, and the raw data obtained from the execution is available on Zenodo: `10.5281/zenodo.3975515`. The JDBL execution framework is composed of 3K lines of Python code.

4.3.2 Debloat correctness protocol (RQ1 & RQ2)

To answer RQ1 and RQ2, we run JDBL on each of the 395 versions of 94 libraries. Those two research questions assess the correctness of JDBL at two different levels: RQ1 assesses the ability of JDBL to produce a debloated JAR file; RQ2 analyzes if the test suite of the library has the same behavior before and after the debloat.

Figure 3 illustrates the pipeline of RQ1 and RQ2. First, we check that the library compiles correctly before the debloat. If it does, then we verify that JDBL has generated a JAR (RQ1). If no JAR file is generated, the debloat is considered as having 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 debloat. To do so, we compare the test execution reports produced during the first step of JDBL execution (see Section 4.3.1) and the test report generated during the verification step of JDBL (see Section 3.3). 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 surefire configuration to run JDBL (see Section 4.3.1). 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 debloat executions to understand what happened.

4.3.3 Debloat effectiveness protocol (RQ3 & RQ4)

In the third and fourth research questions, we study the effectiveness of JDBL in producing a debloated version of the libraries. We focus on two different aspects. The first aspect is 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.

8. <https://maven.apache.org/surefire/maven-surefire-plugin>

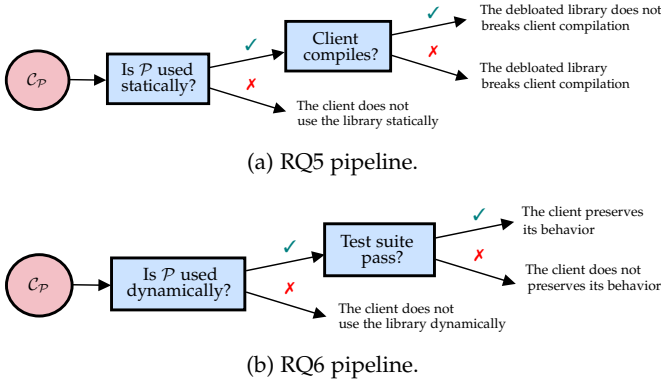


Fig. 4: Pipelines of our experimental protocol to answer RQ5 and RQ6.

To answer those research questions, we use the debloat reports of the original and debloated JAR files generated in Section 4.3.1. These reports contain the list of all the methods and classes of the libraries (including the dependencies), and if the element is debloated. We answer RQ3 by analyzing the ratio of methods and classes that are debloated. For RQ4, we extract the original and debloated JAR, and we collect the size in bytes of all the extracted files. We only consider the size of the bytecode files since JDBL only debloats bytecode.

For those research questions, we consider the 220 library versions that successfully pass the debloat correctness. We separate the 143/220 (65.0%) libraries that do not have dependencies and the 77/220 (35.0%) libraries that have at least one dependency. We decide to do so because we observe that the libraries that have dependencies contain many more elements (bytecode and resources) that impact the analysis when compared to libraries that do not have a dependency.

4.3.4 Debloat impact on clients protocol (RQ5 and RQ6)

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 debloat technique is validated with the clients of the debloated artifacts.

In RQ5, we aim at verifying that the clients still compile when the original library is replaced by the debloated one in their configuration file. This way, we check that JDBL did not remove classes, methods, or interfaces that are necessary for a client. Figure 4a illustrates the pipeline for this research question. First, we check that the client, denoted as C_P in the figure, is using the library statically in the source code. 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.

In RQ6, the goal is to determine if the tests of the clients still pass after the debloat, i.e., that JDBL preserves the functionalities that are necessary for the clients. Figure 4b illustrates the pipeline for this research question. First, we execute the test suite of the client, denoted as C_P in the figure, with the original version of the library to check that the library is covered by at least one test of the client. If yes, we replace the library by 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 for the clients.

To ensure the validity of this protocol, we perform additional checks on the clients. The clients have to use one of the 220 libraries. We only consider the 1,066/1,370 (77.8%) clients that either have a direct reference to the library in their source code or which test suite covers at least one class of the library (static or dynamic usage). The 1,001/1,066 (93.9%) clients that statically use the library serve as the study subjects to answer RQ5. The 283/1,066 (26.5%) clients that have at least a test that covers one of the classes of the dependency serve as the study subjects to answer RQ6.

5 RESULTS

In this section, we present our experimental results on the correctness, effectiveness, and impact of our trace-based debloat approach for automatically removing unnecessary bytecode from Java projects with JDBL.

5.1 Debloat 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.

5.1.1 RQ1. To what extent can a generic, fully automated trace-based debloat technique produce a debloated version of Java projects?

In the first research question, we evaluate the ability of JDBL at performing automatic trace-based debloat for the 395 libraries in our initial benchmark. Here we consider the debloat 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 pass through all the build phases of the Maven build life-cycle, i.e., compilation, testing, and packaging. Therefore, the goal is to assess the correctness of JDBL to execute all its debloat phases without breaking the projects' build, according to the protocol described in Section 4.3.2.

Figure 5 shows a bar plot of the number of successfully debloated libraries, and the number of libraries for which JDBL does not produce a debloated JAR file. The absence of the JAR may occur due to failures during the build process, as a result of modifying the `pom.xml` to execute the experiments with JDBL.

For the 395 libraries of our dataset, JDBL succeeds in producing a debloated JAR file for a total of 311 libraries, and fails to debloat 84. Therefore, the overall debloat success rate of JDBL for producing a debloated version of Java projects is 78.7%. However, when considering only the libraries that are originally compiling, JDBL succeed in debloating 87.9% of the libraries. We identify and classify the causes of failure 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 modifying 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 we consider that 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 may cause 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 to instrument the bytecode of the project and its dependencies to

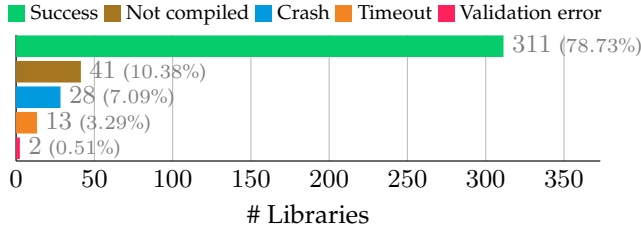


Fig. 5: Number of libraries for which JDBL succeeds or fails to produce a debloated JAR file.

collect complete execution traces. This process induces an additional overhead to 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. Since JDBL alters the behavior of the project build, some of these plugins may not be compatible with JDBL, triggering errors during the build life-cycle.

We manually investigate the causes of the validation errors for the two libraries that fall into this category:

- *org.apache.commons:collection:4.0:* the MANIFEST.MF file is not included in the debloated JAR due to an incompatibility with library plugins. Therefore, Maven fails to package the debloated bytecode.
- *org.yaml:snakeyaml:1.17:* the Maven build fails, triggering a LifecycleExecutionException. The reason is a failure during the library instrumentation with Yajta. This tool relies on Javassist for inserting probes in the bytecode. In this case, JDBL tries to change a class that was frozen by Javassist when it was loaded. Consequently, Javassist crashes because further changes in a frozen class are prohibited.

Answer to RQ1. JDBL successfully produces a debloated version of 311 libraries in our benchmark, which represents 78.7% of the libraries that compile correctly. This is the largest number of debloated subjects in the literature, and the experiment demonstrates the feasibility of trace-based debloat for Java.

5.1.2 RQ2. To what extent do the debloated versions preserve their original behavior?

Our second research question involves evaluating 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 removal operations performed by JDBL affect the results of the tests of the 311 libraries for which JDBL produces a valid JAR file. This semantic correctness assessment constitutes the last phase in the execution of JDBL, as described in Section 3.3.

Figure 6 summarizes the comparison between the test suite executed on the original and the debloated libraries. We observe that, from the 311 libraries that reach the last phase of the debloat procedure, 220 (70.7%) preserve the original behavior (i.e., all the 335,222 tests pass), whereas the number of libraries that have at least one test failure is 30 (9.6%). This high test success rate is a fundamental result to ensure that the debloated version of the artifact preserves semantic correctness.

We excluded 61 (19.6%) libraries because the number of executed tests before and after the debloat do not match.

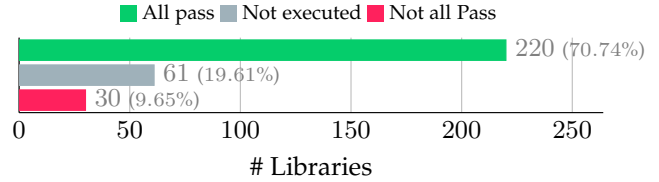


Fig. 6: Number of debloated libraries for which the test suite passes; number of debloated libraries for which the number of executed tests does not match the original test execution (ignored for the research question); number of debloated libraries that have at least one failing test case.

This indicates that the configuration of the tests has changed once JDBL has been injected into the build procedures for 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 because of the scale of this experiment.

In total, we execute 344,596 unique tests, from which 343,191 pass, and 1,405 do not pass (973 fail, and 432 result in error). This represents an overall debloat correctness ratio of 99.59%, considering the total number of tests. This result shows that JDBL is able to capture most of the project behavior, as observed by the tests, while removing the unnecessary bytecode.

We investigate the causes of test failures in the 30 libraries that contain at least one test that does not pass. To do so, we manually analyze the logs of the tests, as reported by Maven. We find the following 5 causes that explain the errors:

- *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.
- *NoClassDefFound (NCDF):* JDBL mistakenly removes a necessary class.
- *Other:* The tests are failing for another reason than the ones previously mentioned.

Table 2 categorizes the tests failures for the 30 libraries. 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 5 causes of test failure according to our manual analysis of the tests' logs: TAF, UOE, NPE, NCDF, and Other. Column 8 (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, the library with the largest number of tests that do not pass is *equalsverifier:3.4.1*, with a total of 605 test failures out of 921 tests in total (283 TAF, 221 NCDF, and 1 Other). These tests failures represent 65.7% of the total number of tests in *equalsverifier:3.4.1*. 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 TAF (592), followed by NCDF (735). We found that these two types of failures are related to each other: when the test uses a non-traced class, the log shows a NCDF, and the test assertion fails consequently. We notice that NCDF and UOE are directly related to the removal

TABLE 2: Classification of the tests that fail for the 30 libraries that do not pass all the tests. We identified five causes of failures through the manual inspection of the Maven test’s logs: TestAssertionFailure (TAF), UnsupportedOperationException (UOE), NullPointerException (NPE), NoClassDefFound (NCDF), and Other.

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 %)
jline:2.2.14.3	2					2/141 (1.4 %)
jackson-annotations:2.7.5		1				1/77 (1.3 %)
commons-beel:6.0	1					1/103 (1.0 %)
commons-beel: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 %)
jline:2.2.13	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 %)

procedure during the debloat process, 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 debloat 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. While these constructs are important for humans to design and program in an object-oriented manner, 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 traced dynamically.

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

More generally, this reveals the challenges of trace-based debloat for real-world Java applications, using the test suite as 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 due to aggressive debloat transformations.

Answer to RQ2. JDBL generates a debloated JAR that preserves the original behavior of 220 (70.7 %) libraries. A total of 343,191 (99.59 %) tests pass on 250 libraries. This original behavioral assessment of trace-based debloat demonstrates that, although not perfect, JDBL preserves a large majority of the libraries’ expected behavior.

5.2 Debloat effectiveness (RQ3 and RQ4)

In this section, we report on the effects of debloating Java libraries with JDBL.

5.2.1 RQ3. How much bytecode is debloated in the compiled projects and their dependencies?

To answer our third research question, we compare the status (kept or removed) of dependencies, classes, and methods in the 220 libraries correctly debloated with JDBL. The goal is to evaluate the effectiveness of JDBL to remove these bytecode elements in the libraries through trace-based debloat.

Figure 7 shows area charts representing the distribution of kept and removed classes and methods in the 220 correctly debloated libraries. We separate the libraries into two sets for better analysis of the impact of dependencies in the bloat: the libraries that have no dependency (Figures 7a and 7c), and the libraries that have at least one dependency (Figures 7b and 7d). 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 (Figures 7a and 7b) or methods (Figures 7c and 7d) kept and removed. The order of the libraries is the same vertically for each figure.

Figure 7a shows the comparison between the percentages of kept and removed classes in the 143 libraries that have no dependency. A total of 119 libraries have at least one removed class. The library with the largest percentage of removed classes is *apache-sling-api:2.16.0* with the 99.1 % of its classes bloated. On the other hand, Figure 7b shows the percentage of removed classes for the 77 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 dependency. 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 on the importance of reducing the number of dependencies to mitigate software bloat.

Figure 7c shows the percentage of kept and removed methods in the 143 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.i4d: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: used classes may still contain a significant number of bloated methods. On the other hand, Figure 7d shows the percentage of kept methods in libraries that have 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 reuse through the declaration of dependencies. Figure 8 shows a beanplot [23] comparing the distribution of the percentage of bloated classes in libraries, with respect to the bloated classes in

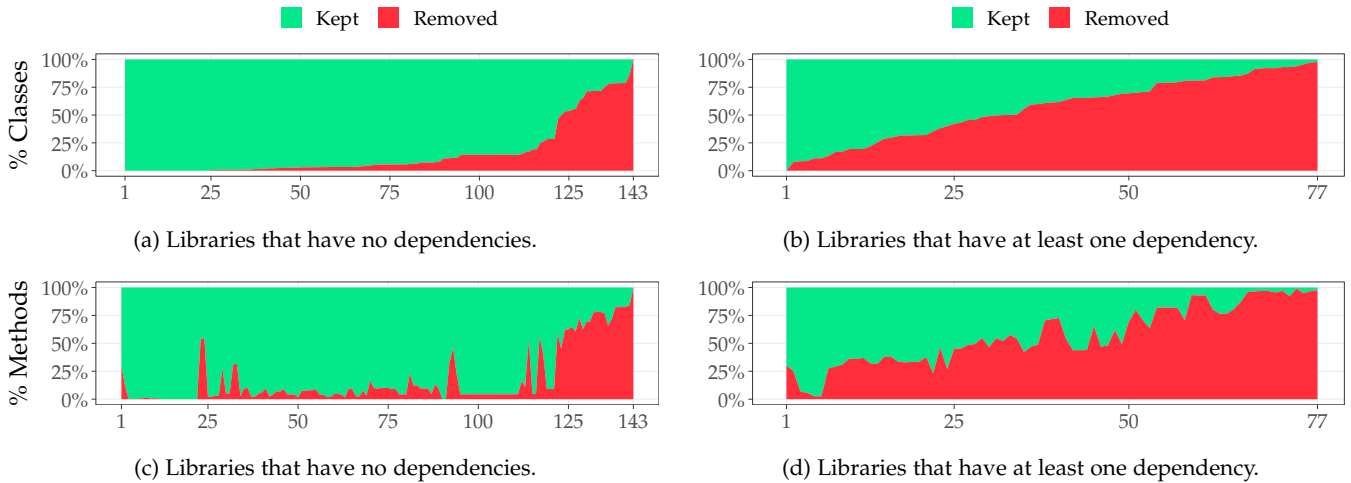


Fig. 7: Percentage of classes kept and removed in (a) libraries that have no dependencies, and (b) libraries that have at least one dependency. Percentage of methods kept and removed in (c) libraries that have no dependencies, and (d) libraries that have at least one dependency.

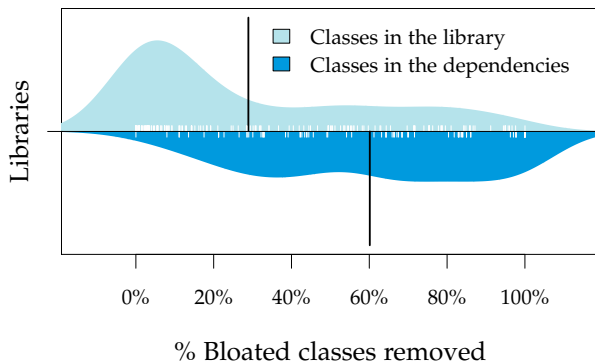


Fig. 8: Distribution of the percentage of bloated classes that belong to libraries, and bloated classes that belong to dependencies. The strip chart (white marks) in between represents the libraries that belong to each of the two groups. The two vertical bars represent the mean value for each group.

TABLE 3: Summary of debloat results for the 220 libraries correctly debloated with JDBL.

		Bloated (%)
Dependencies	52/254 (20.5%)	
Classes	75,273/121,055 (62.2%)	
Methods	505,268/829,015 (60.9%)	

dependencies. The density shape at the top of the plot shows the distribution of the percentage of bloated classes that belong to the 220 libraries. The density shape at the bottom shows this percentage for the classes in the dependencies of the 77 libraries that have at least one dependency. The mean bloat in libraries is 28.9%, whereas in the dependencies it is 60.1%. Overall, the mean percentage of bloated classes removed for all the libraries is 37.1%. The two-samples Wilcoxon test 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 debloat results for the dependencies, classes, and methods. Interestingly, JDBL completely removes

the bytecode for 20.5% of the dependencies. In other words, 52/254 (20.5%) dependencies in the dependency tree of the projects are not necessary to successfully execute the workload in our benchmark. With respect to the classes, 62.2% of them are bloated, from which we determine that 44.4% belong to dependencies. JDBL debloats 60.9% of the methods, from which 50.4% belong to dependencies.

Answer to RQ3: JDBL removes bytecode in all libraries. It reduces the number of dependencies, classes, and methods by 20.5%, 62.2%, and 60.9%, respectively. This result confirms the relevance of the trace-based debloat approach for reducing the unnecessary bytecode of Java projects, while preserving their correctness.

5.2.2 RQ4. What is the impact of trace-based debloat approach on the size of the packaged artifacts?

In this research question, we focus on assessing the effectiveness of JDBL in reducing the size of the packaged artifacts. We consider all the elements in the JAR files before the debloat, 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 summarises the main metrics related to the content and size of the JAR files in our benchmark. 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 benchmark represent 610.3 MB/893.7 MB (68.3%) of pure

TABLE 4: Size in bytes of the elements in the JAR files.

Metrics	Size in bytes (%)
Resources	257,999,869 (22.4 %)
Bytecode	893,732,414 (77.6 %)
Non-bloated classes	283,424,921 (31.7 %)
Bloated classes	596,538,866 (66.7 %)
Bloated methods	13,768,627 (1.5 %)
Total size	1,151,732,283

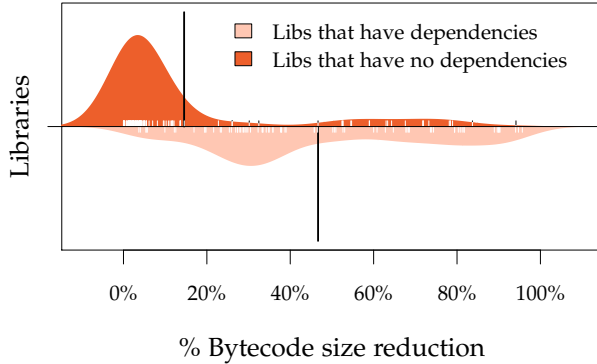


Fig. 9: Distribution of the percentage of reduction of the JAR size in libraries that have no dependencies and libraries that have at least one dependency, with respect to the original bundle.

bytecode: 596.5 MB of bloated classes and 13.8 MB of bloated methods. The used bytecode represents the 31.7 % of the size. 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.

Figure 9 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 mean bytecode size reduction in the libraries that have dependencies (46.7 %) is higher than the libraries with no dependencies (14.5 %). Overall, the mean percentage of bytecode reduction for all the libraries is 25.8 %. The two-samples Wilcoxon test shows that there are significant differences between those two groups (p -value < 0.01). Therefore, we reject the null hypothesis that the trace-based debloat 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.

Answer to RQ4: Trace-based debloat reduces the JAR size of most libraries, while preserving correctness. The percentage of JAR size reduction of pure bytecode after debloat is 68.3 %, which represents a mean reduction of 25.8 % per library. The JAR size reduction is significantly higher in libraries with at least one dependency compared to libraries with no dependency.



Fig. 10: Results of the compilation of the 1,001 clients that use at least one debloated library in the source code.

5.3 Debloat impact on clients (RQ5 and RQ6)

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

5.3.1 RQ5. To what extent do debloated libraries break the compilation of their clients?

In this research question, we investigate how debloating a library with a trace-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, hence having a minimal negative impact on their clients in terms of compilation breakages.

As described in Section 4.3.4, in this research question, we consider the 1,370 clients that use the 220 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 1,001/1,370 (73.1 %) clients that satisfy this condition.

Figure 10 shows the results obtained after attempting to compile the clients with the debloated version of the library. JDBL generates debloated libraries for which 957 (95.6 %) of their clients successfully compile.

From the 1,001 clients that use at least one class of the library, we only observe compilation failures for 44 (4.4 %) clients. Table 5 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, and the number of clients that are affected and the percentage relative to the number of libraries/clients impacted by a compilation error. Note that a client can be impacted by several different errors. Column 4 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 as expected mostly due to errors related to missing packages, classes, methods, and variables (84.0 % of all the compilations errors). Those errors are directly caused by the debloat procedure, the elements in the bytecode are removed, and therefore the clients do not compile. We detect 1,096 errors, but most of them have duplicated causes. Indeed, 20/44 (45.5 %) 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 problems, i.e., by adding the missing element to the debloated library. For example, the 8 clients that are not compiling for the project davidmoten/guava-mini are all failing for the same reasons: the package `com.github.davidmoten.guavamini.annotations` does not exist and the class from the same package `VisibleForTesting` is missing. Those 8 clients would compile if the class `com.github.davidmoten.guavamini.annotations.Visible`

TABLE 5: Frequency of the errors for the 44 unique clients that have compilation errors. Note that a client can have multiple errors from different categories.

Description	# Libraries	# Clients	Occurrence
Cannot find class	15/23 (65.2 %)	25/44 (56.8 %)	722/1,096 (65.9 %)
Package does not exist	8/23 (34.8 %)	16/44 (36.4 %)	116/1,096 (10.6 %)
Unmappable character for encoding UTF8	1/23 (4.3 %)	1/44 (2.3 %)	100/1,096 (9.1 %)
Cannot find variable	8/23 (34.8 %)	10/44 (22.7 %)	81/1,096 (7.4 %)
Cannot find method	1/23 (4.3 %)	1/44 (2.3 %)	28/1,096 (2.6 %)
Static import only from classes and interfaces	2/23 (8.7 %)	2/44 (4.5 %)	18/1,096 (1.6 %)
Method does not override or implement a method from a supertype	3/23 (13.0 %)	3/44 (6.8 %)	14/1,096 (1.3 %)
Processor error	2/23 (8.7 %)	11/44 (25.0 %)	11/1,096 (1.0 %)
Cannot find other symbol	2/23 (8.7 %)	2/44 (4.5 %)	4/1,096 (0.4 %)
UnsupportedOperationException	1/23 (4.3 %)	1/44 (2.3 %)	1/1,096 (0.1 %)
Plugin verification	1/23 (4.3 %)	1/44 (2.3 %)	1/1,096 (0.1 %)
10 Unique errors	23 Unique libraries	44 Unique clients	1,096 Compilation errors

TABLE 6: Frequency of the exceptions thrown during the execution of the tests for the 54 unique clients that have failing test cases. Note that a client can have multiple exceptions from different categories.

Exception	# Libraries	# Clients	Occurrence
UnsupportedOperationException	28/39 (71.8 %)	41/54 (75.9 %)	635/744 (85.3 %)
IllegalStateException	1/39 (2.6 %)	2/54 (3.7 %)	55/744 (7.4 %)
NoClassDefFoundError	7/39 (17.9 %)	7/54 (13.0 %)	26/744 (3.5 %)
Assert	5/39 (12.8 %)	5/54 (9.3 %)	12/744 (1.6 %)
ExceptionInInitializerError	4/39 (10.3 %)	4/54 (7.4 %)	4/744 (0.5 %)
TargetHostsLoadException	1/39 (2.6 %)	1/54 (1.9 %)	4/744 (0.5 %)
NullPointerException	1/39 (2.6 %)	1/54 (1.9 %)	3/744 (0.4 %)
TimeoutException	1/39 (2.6 %)	1/54 (1.9 %)	2/744 (0.3 %)
PushSenderException	1/39 (2.6 %)	1/54 (1.9 %)	2/744 (0.3 %)
AssertionError	1/39 (2.6 %)	1/54 (1.9 %)	1/744 (0.1 %)
10 Unique exceptions	39 Unique libraries	54 Unique clients	744 Failing tests

ForTesting was not debloated and the number of compilation errors would be reduced by 104.

Several clients are also not compiling because of their plugins, in order to force the client to 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 because the compilation should not execute code and therefore should not trigger a run-time exception. It happens during the build of `jenkinsci/warnings-plugin`, which uses the `apache/commons-io:2.6` library. In this case, the compilation itself of `jenkinsci/warnings-plugin` does not fail, but the 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 RQ5: JDBL preserves the syntactic correctness of 957 (95.6 %) clients of debloated libraries. This is the first empirical demonstration that debloat can preserve essential functionalities to successfully compile the clients of debloated libraries, which validates the relevance of this approach to remove unnecessary code.

5.3.2 RQ6. To what extent do debloated libraries affect the behavior of their clients?

In the previous research question, we investigate the impact of the debloat on the compilation of the clients. In this research question, we analyze another facet of the debloat 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 correct behavior.

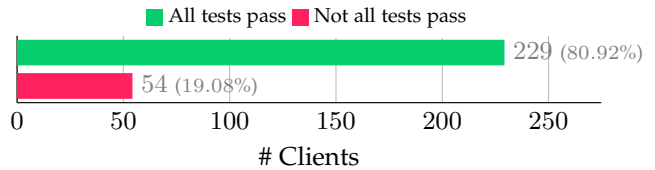


Fig. 11: Results of the tests on the 283 clients that cover at least one debloated library.

If a test in the client is failing after the debloat of the library, then debloat breaks the behavior of the client.

As described in Section 4.3.4, in this research question, we consider the 1,370 clients that use the 220 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 283/1,370 (20.7 %) clients that satisfy this condition.

Figure 11 presents the results obtained after building the clients with the debloated library and running their test suite. In total, 229 (80.9 %) clients pass all the tests, i.e., they behave the same with the original and with the debloated library. There are 54 (19.1 %) clients that have more failing test cases with the debloated library than with the original. However, the number of tests that fail is only 744/44,428 (1.7 %) of the total number of tests in the clients. This result 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 6 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.

From our observations, the most frequent exception is `UnsupportedOperationException` with 635 occurrences in the failing-tests, which affects 75.9 % 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 26 occurrences, is `NoClassDefFoundError`. This exception is similar to `UnsupportedOperationException`, 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`). Therefore, JDBL does not change the behavior of the clients significantly, i.e., only 12 assertions are triggered that verify behavior change in the clients. Having runtime exceptions that are triggered during the executions of the clients is less harmful than having behavior changes that are not verified during the execution of the clients, since the runtime exceptions can be monitored and the execution stops the execution of the client, where a behavior change can stay hidden and corrupt the state of the clients.

Answer to RQ6: JDBL preserves the behavior of 229 (80.9 %) clients of debloated libraries. The failing tests represent only 744 (1.7 %) of the test suite of the clients that have failing tests. Moreover, 99.1 % of the test failures are due to a missing class or method, and not due to a behavioral change which is much harder to detect. These original experiments and observations show, for the first time, that the negative impact of debloated libraries is marginal for their clients.

6 THREATS TO VALIDITY

In this section, we discuss internal, external, and construct threats to the validity of our results.

6.1 Internal validity

The threats to internal validity are related to the accuracy of JDBL to debloat generic real-world Java projects, and how the details of its implementation could influence the results. As explained in Section 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. For example, JaCoCo expects valid so-called "stack map frame" information in class files of Java version 1.6 or higher. Invalid class files are typically created by some frameworks which do not properly adjust stack map frames when manipulating bytecode. However, since we rely on a diverse set of coverage tools, the failures of one specific tool are 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. For example, the Maven shade plugin allows developers to change the produced JAR by including/excluding custom dependencies; or the Maven surefire plugin allows excluding custom test cases, thus altering the validation phase of JDBL. To overcome this threat and to automate our experiments, we set the Maven surefire plugin to its default configuration, and use the Maven assembly plugin to construct the JAR of all the study subjects.

6.2 External validity

The threats to external validity are related to the generalizability of our findings outside the scope of the present experiments. Our observations in Section 5 about bloat are tailored to Java and the Maven ecosystem and hence our findings are restricted to this particular ecosystem. Therefore, the chances are high

that the trace-based debloat approach applied to programs in different languages would yield different conclusions than ours. However, we took care to select all the open-source Java libraries available on GitHub, which cover projects from different domains. To the best of our knowledge, this is the largest set of programs used in software debloat experiments.

6.3 Construct validity

The threats to construct validity are related to the relation between the trace-based debloat approach and the experimental protocol employed. Our analysis is based on a diverse set of real-world open-source Java projects. We minimize the modifications on these projects to run JDBL (only the *pom.xml* file is modified). The assessment of the debloat effectiveness and correctness assumes 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 is not instrumented. Therefore, the usage log is not included in the trace, causing JDBL to remove necessary bytecode. Thus, the quality of the debloat result heavily depends on the effectiveness of the Maven dependency resolution mechanism. Furthermore, the applicability of our trace-based debloat approach heavily depends on the quality of the workload. In the specific case of our experiments, we rely on the projects' test suite; thus, our observations partly depend on the coverage of the projects, which determines the completeness of the debloat result. Since we rely on the developer's written test cases, we cannot guarantee that the test suite exercises the main functionalities of the debloated artifact. However, as explained in Section 4.2, the coverage of the libraries in our benchmark is high.

7 RELATED WORK

In this section, we present the work related to software debloat techniques and dynamic analysis.

7.1 Software debloat

Research interest in software debloat has grown in recent years, motivated by the reuse of large open-source libraries designed to provide several functionalities for different clients [24], [25]. Seminal work on debloat for Java programs was performed by Tip et al. [26], [27]. 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 debloat Java frameworks and Android applications using static analysis. Jiang et al. [28] presented JRED, a tool to reduce the attack surface by trimming redundant code from Java binaries. REDDROID [15] and POLYDROID [29] propose debloat techniques for mobile devices. They found that debloat significantly reduces the bandwidth consumption used when distributing the application, improving the performance of the system by optimizing resources. Other works rely on debloat to improve the performance of the Maven build automation system [30] or removing bloated dependencies [31]. More recently, Haas et al. [9] 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 debloat in practice.

To improve the debloat results of static analysis, recent debloat techniques drive the removal process using information collected at run-time. 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 [32], [33], [34]. Unfortunately, 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. [11] propose TRIMMER, a debloat approach that relies on user-provided configurations and compiler optimization to reduce code size. Qian et al. [12] present RAZOR, a tool to debloat program binaries based on test cases and control-flow heuristics. However, the authors do not provide a thorough analysis on the challenges and benefits of using execution traces to debloat software. These previous works assess the impact of debloat on the size of the programs, yet, they rarely evaluate to what extent the debloat transformations preserve program behavior.

This work contributes to the state of the art of software debloat. We propose a novel approach to debloat Java software based on the collection of execution traces to identify unused software parts. Our tool, JDBL, integrates the debloat 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 debloat literature and we provide the first quantitative investigation of the impact of debloat 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, such as program slicing [35], program comprehension [36], or dynamic taint tracking [37]. 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 [38], [39]. Mururu et al. [40] implement 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. In this work, we employ dynamic analysis for bytecode reduction, as opposed to run-time memory bloat, which was the target of previous works [41], [42], [43], [44], [45].

In Java, dynamic analysis is often used to overcome the limitations of static analysis. Landman [19] 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. [46] utilize execution traces to identify and understand features in Android applications by analyzing its dynamic behavior. In order to leverage dynamic analysis for debloat, we need to collect a very accurate set of execution traces, which guide the debloat procedure.

Our work contributes to the state of the art of dynamic analysis for Java programs. JDBL combines the traces obtained from four distinct code coverage tools through bytecode instrumentation [47]. The composition of these four types of observation allows us to build very accurate and complete traces, which are necessary to know exactly what parts of the code are used at runtime and which ones can be removed. To collect the execution traces, we rely on the test suite of the libraries. This approach is similar to other dynamic analyses, e.g., for finding backward incompatibilities [48].

8 CONCLUSION

In this work, we presented a novel approach to automatically debloat software applications based on dynamic analysis, which we coined as *trace-based debloat*. We created a tool, JDBL, that implements this approach to debloat Java projects. JDBL monitors the execution of a program in order to achieve a unique goal: collect the minimum set of classes and methods that are necessary to execute the program with a given workload. We have extensively evaluated JDBL using an original experimental protocol that assessed the impact of the debloat on the libraries' behavior, on their size, as well as on their clients. Our results indicated that 62.2 % of classes and 60.9 % of methods in the studied libraries can be debloated, which represents a reduction of 68.3 % of the bytecode size.

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

The next step of trace-based debloat is to specialize applications with respect to usage profiles collected in production, extending the debloat to other parts of the program stack (e.g., to the Java RE, program resources, or containerized applications). As for the empirical investigation of debloat impact, we aim at evaluating the effectiveness of trace-based debloat to reduce the attack surface of applications. These are major milestones towards full stack debloat in order to drastically reduce the size of software and achieve significant resource savings.

ACKNOWLEDGMENTS

This work is partially supported by the Wallenberg AI, Autonomous Systems, and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation and by the TrustFull project funded by the Swedish Foundation for Strategic Research.

REFERENCES

- [1] N. Wirth, "A plea for lean software," *Computer*, vol. 28, no. 2, pp. 64–68, 1995.
- [2] G. J. Holzmann, "Code Inflation," *IEEE Software*, vol. 32, pp. 10–13, Mar 2015.
- [3] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST '17*, (New York, NY, USA), pp. 65–70, ACM, 2017.
- [4] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19*, (USA), p. 1697–1714, USENIX Association, 2019.
- [5] H. Koo, S. Ghavamnia, and M. Polychronakis, "Configuration-driven software debloating," in *Proceedings of the 12th European Workshop on Systems Security, EuroSec '19*, (New York, NY, USA), Association for Computing Machinery, 2019.
- [6] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: Automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017*, (New York, NY, USA), p. 476–486, Association for Computing Machinery, 2017.
- [7] H. Z. anfsdd H. Mei, "An empirical study on api usages," *IEEE Transactions on Software Engineering*, vol. 45, no. 4, pp. 319–334, 2019.
- [8] A. Gkortzis, D. Feitosa, and D. Spinellis, "Software reuse cuts both ways: An empirical analysis of its relationship with security vulnerabilities," *Journal of Systems and Software*, p. 110653, 2020.

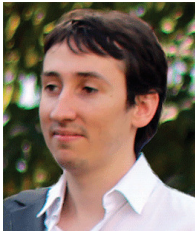
- [9] R. Haas, R. Niedermayr, T. Roehm, and S. Apel, "Is static analysis able to identify unnecessary source code?," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, Jan. 2020.
- [10] Y. Chen, T. Lan, and G. Venkataramani, "Damgate: Dynamic adaptive multi-feature gating in program binaries," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST '17*, (New York, NY, USA), pp. 23–29, ACM, 2017.
- [11] H. Sharif, M. Abubakar, A. Gehani, and F. Zaffar, "Trimmer: Application specialization for code debloating," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018*, (New York, NY, USA), pp. 329–339, ACM, 2018.
- [12] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "[RAZOR]: A framework for post-deployment software debloating," in *28th Security Symposium (USENIX)*, pp. 1733–1750, 2019.
- [13] A. Quach, A. Prakash, and L. Yan, "Debloating software through piece-wise compilation and loading," in *27th USENIX Security Symposium (USENIX Security 18)*, (Baltimore, MD), pp. 869–886, USENIX Association, Aug. 2018.
- [14] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, *The Java virtual machine specification*. Pearson Education, 2014.
- [15] Y. Jiang, Q. Bao, S. Wang, X. Liu, and D. Wu, "Reddroid: Android application redundancy customization based on static analysis," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 189–199, Oct 2018.
- [16] E. Lafortune, "Proguard," <http://proguard.sourceforge.net>, 2004.
- [17] S. Liang and G. Bracha, "Dynamic class loading in the java virtual machine," in *Proceedings of the 13th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '98*, (New York, NY, USA), p. 36–44, Association for Computing Machinery, 1998.
- [18] L. Sui, J. Dietrich, M. Emery, S. Rasheed, and A. Tahir, "On the soundness of call graph construction in the presence of dynamic language features - a benchmark and tool evaluation," in *Programming Languages and Systems (S. Ryu, ed.)*, (Cham), pp. 69–88, Springer International Publishing, 2018.
- [19] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection: Literature review and empirical study," in *Proceedings of the 39th International Conference on Software Engineering, ICSE'17*, p. 507–518, IEEE Press, 2017.
- [20] D. Foo, J. Yeo, H. Xiao, and A. Sharma, "The dynamics of software composition analysis," *Poster at the 34th ACM/IEEE International Conference on Automated Software Engineering*, 2019.
- [21] Q. Yang, J. J. Li, and D. M. Weiss, "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589–597, 2009.
- [22] F. Horváth, T. Gergely, Á. Beszédes, D. Tengeri, G. Balogh, and T. Gyimóthy, "Code coverage differences of java bytecode and source code instrumentation tools," *Software Quality Journal*, vol. 27, no. 1, pp. 79–123, 2019.
- [23] P. Kampstra et al., "Beanplot: A boxplot alternative for visual comparison of distributions," *Journal of Statistical Software*, vol. 28, no. 1, pp. 1–9, 2008.
- [24] S. Eder, H. Femmer, B. Hauptmann, and M. Junker, "Which features do my users (not) use?," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 446–450, 2014.
- [25] Y. Jiang, C. Zhang, D. Wu, and P. Liu, "Feature-based software customization: Preliminary analysis, formalization, and methods," in *2016 IEEE 17th International Symposium on High Assurance Systems Engineering (HASE)*, pp. 122–131, 2016.
- [26] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '99*, (New York, NY, USA), p. 292–305, Association for Computing Machinery, 1999.
- [27] F. Tip, P. F. Sweeney, C. Laffra, A. Eisma, and D. Streeter, "Practical extraction techniques for java," *ACM Trans. Program. Lang. Syst.*, vol. 24, p. 625–666, Nov. 2002.
- [28] Y. Jiang, D. Wu, and P. Liu, "'jred: Program customization and bloatware mitigation based on static analysis'," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 12–21, June 2016.
- [29] B. Heath, N. Velingker, O. Bastani, and M. Naik, "Polydroid: Learning-driven specialization of mobile applications," *arXiv preprint arXiv:1902.09589*, 2019.
- [30] A. Celik, A. Knaust, A. Milicevic, and M. Gligoric, "Build system with lazy retrieval for java projects," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE'16*, (New York, NY, USA), p. 643–654, Association for Computing Machinery, 2016.
- [31] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A Comprehensive Study of Bloating Dependencies in the Maven Ecosystem," *arXiv e-prints*, p. arXiv:2001.07808, Jan. 2020.
- [32] U. P. Schultz, J. L. Lawall, and C. Consel, "Automatic program specialization for java," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 25, no. 4, pp. 452–499, 2003.
- [33] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, (New York, NY, USA), p. 380–394, Association for Computing Machinery, 2018.
- [34] H. Vázquez, A. Bergel, S. Vidal, J. D. Pace, and C. Marcos, "Slimming javascript applications: An approach for removing unused functions from javascript libraries," *Information and Software Technology*, vol. 107, pp. 18–29, 2019.
- [35] H. Agrawal and J. R. Horgan, "Dynamic program slicing," *SIGPLAN Not.*, vol. 25, p. 246–256, June 1990.
- [36] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, pp. 684–702, Sep. 2009.
- [37] J. Bell and G. Kaiser, "Phosphor: Illuminating dynamic data flow in commodity jvms," *ACM Sigplan Notices*, vol. 49, no. 10, pp. 83–101, 2014.
- [38] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani, "A trace-based java jit compiler retrofitted from a method-based compiler," in *International Symposium on Code Generation and Optimization (CGO 2011)*, pp. 246–256, 2011.
- [39] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderma, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz, "Trace-based just-in-time type specialization for dynamic languages," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '09*, (New York, NY, USA), p. 465–478, Association for Computing Machinery, 2009.
- [40] G. Mururu, C. Porter, P. Barua, and S. Pande, "Binary debloating for security via demand driven loading," *arXiv preprint arXiv:1902.06570*, 2019.
- [41] N. Mitchell, E. Schonberg, and G. Sevitsky, "Four trends leading to java runtime bloat," *IEEE Software*, vol. 27, no. 1, pp. 56–63, 2010.
- [42] G. Xu, "Coco: Sound and adaptive replacement of java collections," in *ECOOP 2013 – Object-Oriented Programming (G. Castagna, ed.)*, (Berlin, Heidelberg), pp. 1–26, Springer Berlin Heidelberg, 2013.
- [43] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research, FoSER'10*, (New York, NY, USA), pp. 421–426, ACM, 2010.
- [44] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, (New York, NY, USA), p. 268–278, Association for Computing Machinery, 2013.
- [45] S. Bhattacharya, K. Gopinath, and M. G. Nanda, "Combining concern input with program analysis for bloat detection," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13*, (New York, NY, USA), p. 745–764, Association for Computing Machinery, 2013.
- [46] Q. Xin, F. Behrang, M. Fazzini, and A. Orso, "Identifying features of android apps from execution traces," in *Proceedings of the 6th International Conference on Mobile Software Engineering and Systems, MOBILESoft '19*, p. 35–39, IEEE Press, 2019.
- [47] W. Binder, J. Hulaas, and P. Moret, "Advanced java bytecode instrumentation," in *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java, PPPJ '07*, (New York, NY, USA), p. 135–144, Association for Computing Machinery, 2007.
- [48] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *IEEE/ACM International Conference on Software Engineering*, 2020.



César Soto-Valero is a PhD student in Software Engineering at KTH Royal Institute of Technology, Sweden. His research work focuses on leveraging static and dynamic program analysis techniques to mitigate software bloat. César received his MSc degree and BSc degree in Computer Science from Universidad Central "Marta Abreu" de Las Villas, Cuba.



Thomas Durieux is a Post-doc at KTH Royal Institute of Technology, Sweden. He is currently working on software debloat and software art. Thomas did his PhD on automatic program repair, with a specific focus on the production environment at INRIA Lille, France.



Nicolas Harrand is a PhD student in Software Engineering at KTH Royal Institute of Technology, Sweden. His current research is focused on automatic software diversification. Nicolas studied Computer Science and Applied Mathematics in Grenoble, France.



Benoit Baudry is a Professor in Software Technology at KTH Royal Institute of Technology in Stockholm, Sweden. He received his PhD in 2003 from the University of Rennes and was a research scientist at INRIA from 2004 to 2017. His research is in the area of software testing, code analysis and automatic diversification. He has led the largest research group in software engineering at INRIA, as well as collaborative projects funded by the European Union, and software companies.