Automatic Specialization of Third-Party Java Dependencies

César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry
KTH Royal Institute of Technology, Stockholm, Sweden
Email: {cesarsv, deepikat, baudry}@kth.se; toady@eecs.kth.se

Abstract—Large-scale code reuse significantly reduces both development costs and time. However, the massive share of third-party code in software projects poses new challenges, especially in terms of maintenance and security. In this paper, we propose a novel technique to specialize dependencies of Java projects, based on their actual usage. Given a project and its dependencies, we systematically identify the subset of each dependency that is necessary to build the project, and we remove the rest. As a result of this process, we package each specialized dependency in a JAR file. Then, we generate specialized dependency trees where the original dependencies are replaced by the specialized versions. This allows building the project with significantly less third-party code than the original. As a result, the specialized dependencies become a first-class concept in the software supply chain, rather than a transient artifact in an optimizing compiler toolchain. We implement our technique in a tool called DEPTRIM, which we evaluate with 30 notable open-source Java projects. DEPTRIM specializes a total of 343 $(86.6\,\%)$ dependencies across these projects, and successfully rebuilds each project with a specialized dependency tree. Moreover, through this specialization, DEPTRIM removes a total of $57{,}444$ $(42.2\,\%)$ classes from the dependencies, reducing the ratio of dependency classes to project classes from $8.7\,\times$ in the original projects to $5.0\,\times$ after specialization. These novel results indicate that dependency specialization significantly reduces the share of third-party code in Java projects.

Index Terms—Software specialization, Software debloating, Maven, Software supply chain, Dependency trees

1 Introduction

SOFTWARE projects are developed by assembling new features and components provided by reusable third-party libraries. Software reuse at large is a known best practice in software engineering [1]. Its adoption has rocketed in the last decade, thanks to the rapid growth of repositories of reusable packages, along with the development of mature package managers [2]. These package managers let developers declare a list of third-party libraries that they want to reuse in their projects. The libraries declared by developers form the set of direct dependencies of the project. Then, at build time, the package manager fetches the code of these libraries, as well as the code of transitive dependencies, declared by the direct dependencies. This forms a dependency tree that the build system bundles with the project code into a package that can be released and deployed.

The large-scale adoption of software reuse [3] is beneficial for software companies as it reduces their delivery times and costs [4]. Meanwhile, reuse today has reached a point where most of the code in a released application actually originates from third-party dependencies [5]. This massive presence of third-party code in application binaries has turned software reuse into a double-edged sword [6]. Recent studies have highlighted the new challenges that third-party dependencies pose for maintenance [7], [8], performance [9], code quality [10], and security [11], [12].

Several techniques have emerged to address the challenges of dependency management. The first type of approach consists of supporting developers in maintaining a correct and secure dependency tree. Software composition analysis [13] and software bots [14] suggest dependency updates and warn about potential vulnerabilities among dependencies. Integrity-checking tools aim at preventing packaging a dependency with code that may have been tempered with. For example, the Go community maintains a global database for authenticating module content [15] and sigstore facilitates the procedure of signing third-party libraries [16]. A second type of approach to maintain healthy dependency trees consists in reducing it, removing the dependencies that are completely unused. Examples of such techniques include package debloating for Linux applications [17] dependency debloating or shading for Java applications [18], or tree shaking for JavaScript applications [19].

In this paper, we aim at advancing the state-of-the-art of dependency tree reduction with a novel technique that specializes dependency trees to the needs of an application. Mishra and Polychronakis first introduced the concept of API specialization [20] to reduce the attack surface of third-party libraries. In this work, we advance the state-of-the-art on library specialization with a combination of static and dynamic code analysis to generate (i) a specialized version of an application's dependency tree and (ii) specialized versions of each third-party library that is partially used by an application, which can be deployed in a library registry. We implement our new technique in DEPTRIM, a tool that automatically specializes third-party libraries in the dependency tree of Java applications.

DEPTRIM analyses the bytecode of a Java project, as well as all its direct and transitive third-party dependencies. First, it removes the dependencies that are completely bloated, and identifies the non-bloated ones. Next, for each non-bloated dependency, DEPTRIM builds a static call graph through all non-bloated dependencies, to identify the classes for which at least one member is reachable from the project. DepTrim then removes the unused classes and produces one specialized jar for each dependency. Finally, DepTrim modifies the dependency tree of the project, replacing the original dependencies with the specialized versions in the build file. The output of DepTrim is a specialized dependency tree of the project, with the maximum number of specialized dependencies such that the project builds correctly, *i.e.*, the project correctly compiles and all its tests pass, providing evidence that the expected behavior of the project, as specified within the test suite, is preserved. DepTrim simplifies the reuse of specialized dependencies by generating reusable `JAR` files, which can be readily deployed to external repositories and can be documented and versioned as part of the project's software bill of material [21].

We demonstrate the capabilities of DepTrim by performing a study with 30 mature open-source Java projects that are configured to build with Maven. DepTrim successfully analyzes 135,343 classes across the 467 dependencies of the projects. For 14 projects, it generates a dependency tree in which all `compile`-scope dependencies are specialized. For the 16 other projects, DepTrim produces a dependency tree that includes all dependencies that can be specialized without breaking the build, while keeping the others intact. In total, DepTrim removes 51,631 (39.9 %) unused classes from 343 third-party dependencies. The specialized dependencies are deployed locally, as reusable `JAR` files. For each project, DepTrim produces a specialized version of the *pom.xml* file that replaces original dependencies with specialized ones, such that the project still correctly builds.

In summary, our contributions are as follows:

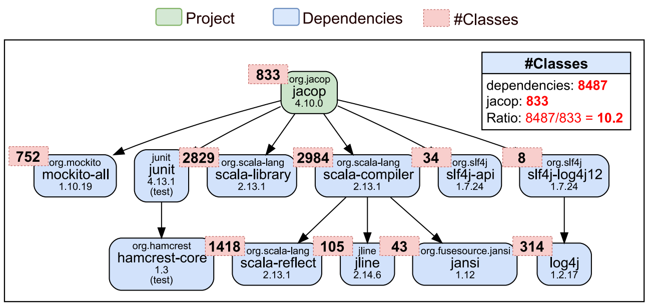
- A fully automated technique to specialize the dependency tree of Java projects at build time.
- A tool called DepTrim, which automatically builds Maven projects with the largest subset of specialized dependencies.
- Novel observations about the ratio of dependency classes compared to project classes collected on 30 mature open-source projects at three stages of the dependency tree: original, debloated, and specialized.
- Empirical evidence that DepTrim successfully specializes the dependency tree of 14 projects in its entirety, and 16 partially, reducing the number of third-party classes by (42.2 %). The project classes to dependency classes ratio is divided by two, from 8.7 × to 5.0 ×.

2 Background

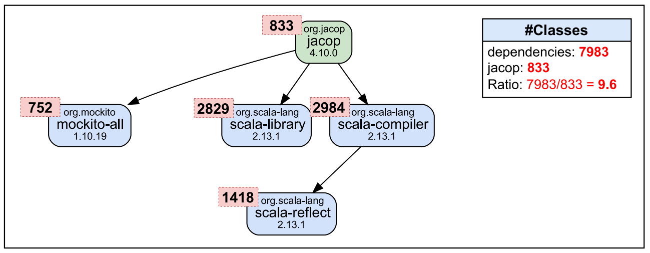
In this section, we introduce the existing techniques to reduce the amount of dependency code. Then we present the opportunities for dependency specialization, for Java projects.

2.1 Terminology

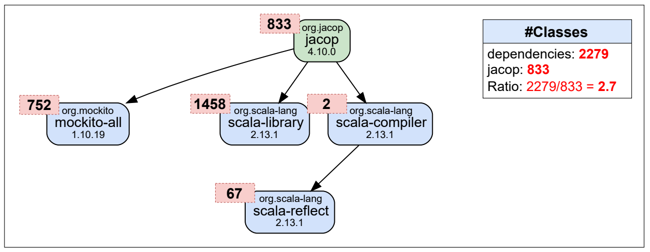
In this work, we consider a software project as a collection of Java source code files and configuration files organized to be built with Maven [22]. Maven is a build automation tool for Java-based projects. It is primarily used for managing the dependencies of a project, testing it, and packaging it, as specified in the Project Object Model (POM) expressed


Project
Dependencies
#Classes
#Classes
dependencies: 8487
jacop: 833
Ratio: 8487/833 = 10.2
833 org.jacop jacop 4.10.0
752 org.mockito mockito-all 1.10.19
junit junit 4.13.1 (test)
2829 org.scala-lang scala-library 2.13.1
2984 org.scala-lang scala-compiler 2.13.1
34 org.slf4j slf4j-api 1.7.24
8 org.slf4j slf4j-log4j12 1.7.24
org.hamcrest hamcrest-core 1.3 (test)
1418 org.scala-lang scala-reflect 2.13.1
105 jline jline 2.14.6
43 org.fusesource.jansi jansi 1.12
314 log4j 1.2.17

(a) Maven dependency resolution (default)


#Classes
dependencies: 7983
jacop: 833
Ratio: 7983/833 = 9.6
833 org.jacop jacop 4.10.0
752 org.mockito mockito-all 1.10.19
2829 org.scala-lang scala-library 2.13.1
2984 org.scala-lang scala-compiler 2.13.1
1418 org.scala-lang scala-reflect 2.13.1

(b) DepClean dependency debloating (state-of-the-art)


#Classes
dependencies: 2279
jacop: 833
Ratio: 2279/833 = 2.7
833 org.jacop jacop 4.10.0
752 org.mockito mockito-all 1.10.19
1458 org.scala-lang scala-library 2.13.1
2 org.scala-lang scala-compiler 2.13.1
67 org.scala-lang scala-reflect 2.13.1

(c) Classes actually used in the dependencies (determined by static analysis)

Figure 1: Example of transformations to reduce the share of dependency code in the project `jacop v4.10.0`, and the impact of such transformations on the dependency classes to project classes ratio. Dependencies have `compile`-scope by default if not specified. Note that `jacop` reuses only a portion of its third-party dependencies.

in a file called *pom.xml*. This file, located at the root of the project, includes additional information such as the project name and version. We now define the key concepts about dependencies in the Maven ecosystem.

Definition 1. ***Maven dependency:*** A Maven dependency defines a relationship between a project and another compiled project. Dependencies are compiled `JAR` files, uniquely identified with a triplet (`G:A:V`) where `G` is the `groupId`, `A` is the `artifactId`, and `V` is the `version`. Dependencies are defined in the *pom.xml* within a scope, which determines the phase of the Maven build cycle at which the dependency is required. Maven distinguishes 6 dependency scopes: `compile`, `runtime`, `test`, `provided`, `system`, and `import`.

For example, the constraint programming solver `jacop` (`6ed0cd0`) is a Maven project. As illustrated in Figure 1a, `scala-library` is one of its 11 dependencies. This is a `compile`-scope dependency, which means that `jacop` can use some functionalities of `scala-library` at compile time, and will include all the code of `scala-library` within the packaged binary of `jacop`. The testing framework `junit` is also declared as a dependency of `jacop` within the `test` scope, indicating that this dependency is required only for executing unit tests.

Definition 2. ***Dependency tree:*** The dependency tree of a MAVEN project is a directed acyclic graph that includes all the direct dependencies declared by developers in the project *pom.xml*, as well as all the transitive dependencies, *i.e.* dependencies in the transitive closure of direct dependencies. For a MAVEN project, there exists a dependency resolution mechanism that fetches both direct and transitive dependency `JAR` files not present locally from external repositories such as Maven Central [23]. The project becomes the root node of the tree, while the edges represent dependency relationships between its direct and transitive dependencies.

For example, in Figure 1a, `scala-compiler` is a direct dependency of `jacop` because it is declared by developers in the *pom.xml*. It depends on `scala-reflect`, which makes `scala-reflect` a transitive dependency of `jacop`. The 6 direct and 5 transitive dependencies of `jacop` constitute its dependency tree.

Definition 3. ***Bloated dependency:*** A dependency is said to be bloated if none of the elements in its API are used, directly or indirectly, by the project [24]. This means that, although they are present in the dependency tree of software projects, bloated dependencies are entirely unused. Developers are therefore encouraged to remove them [25].

For example, Figure 1b presents the `compile`-scope dependencies of `jacop` after the removal of its bloated dependencies. The dependencies `jline`, `jansi`, `sl4j-api`, `sl4j-log4j12`, and `log4j` are bloated and have been safely removed, as no member of their APIs is exercised by `jacop`.

2.2 Example

We now illustrate the MAVEN dependency resolution mechanism and the concept of bloated dependencies. Figure 1 shows the example of the transformations of the dependency tree of the project `jacop`. In Figure 1a, we see the dependency tree of `jacop` as generated by the MAVEN dependency resolution mechanism: it fetches `JAR` files from external repositories while omitting duplication, avoiding conflicts, and constructing a tree representation of the dependencies [26]. `jacop` has a total of 11 third-party dependencies: 6 are direct and 5 are transitive. Direct dependencies are explicitly declared by the developers in the *pom.xml* file of `jacop`, while transitive dependencies are resolved automatically via the MAVEN dependency resolution mechanism. MAVEN uses the concept of scope to determine the visibility and lifecycle of a dependency, *i.e.*, whether it should be included in the classpath of a certain build phase, as well as what the classpath of an artifact should be during the execution of a build phase. For example, `jacop` has 9 `compile`-scope dependencies (the default) and 2 `test` scope dependencies. When `jacop` is packaged for deployment as a `jar-with-dependencies`, its `JAR` file will include the bytecode of all its 9 `compile`-scope dependencies. These `compile`-scope dependencies include 8,487 `class` files, while the number of classes within `jacop`, written and tested by its developers, is 833. As observed, the number of classes contributed by third-party dependencies is one order of magnitude (i.e., $10.2\times$) more than the number of classes written by the `jacop` developers.

When we run DEPCLEAN, a state-of-the-art MAVEN plugin that identifies and removes bloated dependencies [24], [18], we find that 5 dependencies of `jacop` are never used, and are therefore marked as bloated. Figure 1b shows the dependency tree of `jacop` after `test`-scope dependencies and bloated dependencies are removed. In this case, the number of nodes in the tree is reduced from 11 to 4. The reduction in the number of `compile`-scope dependencies represents a removal of 504 (5.9 %) third-party classes (*e.g.*, removing `sl4j-api` leads to the removal of 34 classes). For `jacop`, the removal of bloated dependencies has a minimal impact on the reduction of third-party classes. Consequently, while complete dependency debloating drastically reduces the number of dependencies in `jacop`, it only leads to a modest reduction in the ratio of dependency classes to project classes, from the original $10.2\times$ in Figure 1a to $9.6\times$ in Figure 1b.

To assess the opportunities of further reducing the number of dependency classes, we analyze the `JAR` of each non-debloated dependency of `jacop`. We compute the static call graph of method calls between the classes in the `JAR` files. Based on this graph, we get the list of dependency classes that are reachable from the project at build time. Figure 1c shows the number of reachable classes for each dependency of `jacop`. Consider the direct dependency `scala-compiler`. Of its 2,984 classes, only two are reachable from `jacop`. This confirms that `scala-compiler` is not a bloated dependency for `jacop`, and that it includes way more features than what `jacop` actually needs. This is evidence of the opportunity to specialize this dependency in the context of `jacop`. Similar opportunities exist for 2 other non-bloated dependencies. In fact, we find that 5,704/7,983 (71.5 %) of the third-party classes in these dependencies can be removed, and `jacop` can still build successfully. After dependency specialization, the ratio of the number of dependency classes to `jacop`'s classes is $2.7\times$. This is a drastic reduction from $9.6\times$ which was the ratio after debloating (Figure 1b), and even more significant if we consider the original ratio of $10.2\times$ in Figure 1a.

The number of classes actually used in the dependencies is significantly lower than the original number of classes provided. This observation motivates us to extend the state-of-the-art of Java dependency management with a novel technique to specialize non-bloated dependencies, by identifying and removing unnecessary classes through bytecode removal. In the next section, we present our approach and provide details on DEPTRIM, a tool that automatically specializes the dependencies of MAVEN projects.

3 DEPENDENCY SPECIALIZATION WITH DEPTRIM

This section presents DEPTRIM, an end-to-end tool for the automated specialization of third-party Java dependencies. We define the concept of dependency specialization, followed by an explanation of the key phases of DEPTRIM.

3.1 Dependency Specialization

This work introduces the concept of specialized dependencies and specialized dependency trees. We define them below.

Definition 4. ***Specialized dependency:*** A dependency is said to be specialized with respect to a project if all the

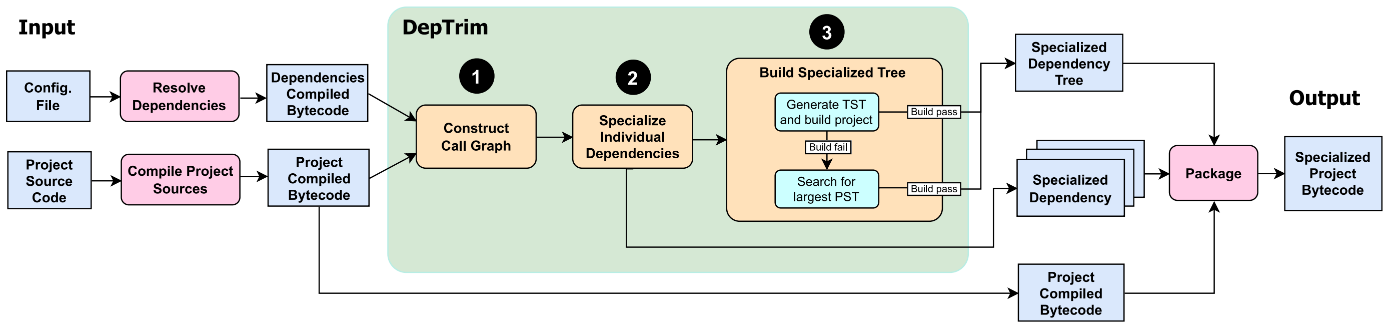


Figure 2: Overview of the dependency specialization approach implemented in DEPTRIM. Blue boxes are software artifacts, pink rounded boxes are actions performed by the build engine, and each of the three main phases of DEPTRIM are indicated within the green rounded box.

classes within the dependency are used by the project, and all unused classes have been identified and removed. Consequently, there is no class in the API of a specialized dependency that is unused, directly or indirectly, by the project or any other dependency in its dependency tree.

Given a project, DEPTRIM creates a set of specialized dependencies, such that the dependency tree of the project is composed of dependencies that contain only classes that the project uses. Recalling the example in Figure 1b and Figure 1c, `jacop` uses 2 of the 2,984 classes in `scala-compiler`. Therefore, `scala-compiler` could be specialized with respect to `jacop`, by removing the 2,982 unused classes.

Definition 5. ***Specialized dependency tree:*** A specialized dependency tree is a dependency tree where at least one dependency is specialized and the project still correctly builds with that dependency tree. This means that in at least one of the used dependencies, unused classes have been identified and removed. A specialized dependency tree may be one of the following two types:

- *Totally Specialized Tree* (TST): A dependency tree where all used dependencies are specialized and the project build is successful.
- *Partially Specialized Tree* (PST): A dependency tree with the largest possible number of specialized dependencies, such that the project build is successful.

We discuss our approach for building a project with a TST or PST with DEPTRIM in the following subsections. DEPTRIM identifies unused classes within the non-bloated `compile`-scope dependencies of MAVEN projects, and removes them in order to produce specialized dependencies. Using these, DEPTRIM prepares a specialized dependency tree for the project such that the project still correctly builds. The following subsections explain this technique in detail.

3.2 DEPTRIM

Figure 2 illustrates the complete pipeline of the dependency specialization approach implemented in DEPTRIM. DEPTRIM receives as inputs the source code and the *pom.xml* file of a Java MAVEN project. The project must successfully build. DEPTRIM outputs three elements: (i) a specialized version of the *pom.xml* file, which removes the bloated dependencies and includes the largest possible number of specialized dependencies that keep the build passing; (ii) the set of specialized dependencies as reduced JAR files; (iii) the project compiled from its source can be packaged with the specialized dependencies in order to have a smaller `jar-with-dependencies` for release and deployment.

DEPTRIM validates that the project still builds correctly with the specialized dependency tree. Note that DEPTRIM only transforms the bytecode of the third-party dependencies, while the original project source and its compiled bytecode remain intact.

As illustrated in Figure 2, the specialization procedure of DEPTRIM consists of three main phases. First ❶, DEPTRIM leverages state-of-the-art Java static bytecode analysis to construct a static call graph of the class members in the third-party dependencies that are reachable from the project binaries. The completeness of this reachability analysis is critical for the identification of unused third-party classes. Second ❷, DEPTRIM transforms the bytecode in the dependencies to remove unused classes. This task requires integration with the MAVEN build engine to resolve and deploy the modified dependencies to the local repository. Finally ❸, DEPTRIM specializes the dependency tree of the project by modifying its original *pom.xml* file. The modified *pom.xml* should preserve the original configurations, except the dependency declarations, which point to the specialized dependencies instead of the original ones. Moreover, DEPTRIM must validate that dependency specialization does not break project build. We provide more details of these three phases in the following subsections.

3.2.1 ❶ Call Graph Construction

Before it can specialize dependencies, DEPTRIM determines their API usage, based on static analysis of the project binary. To do so, DEPTRIM constructs a call graph using two inputs: the compiled dependencies as resolved by MAVEN (Line 1 of Algorithm 1), and the compiled project sources (Line 2). Then, using the bytecode class members of the project as entry points to this graph (Line 3), DEPTRIM infers and reports class usage information from the bytecode directly, without loading or initializing classes. The report captures the set of dependencies, classes, and methods that are actually used by the project, *i.e.*, that are reachable via static analysis. The output of this phase is a data structure that identifies the minimal set of classes in each of the dependencies that are required to build the project.

The collection of accurate and complete call graphs is essential for specialization. If a necessary class member is not reachable statically, then DEPTRIM will consider it as unused and proceed to remove it in a subsequent phase. To mitigate this limitation, DepTrim relies on state-of-the-art static analysis of Java bytecode to capture invocations between classes, methods, fields, and annotations from the project and its direct and transitive dependencies. Furthermore, it parses the constant pool of `class` files in order to capture dynamic invocations from string literals (*e.g.*, when loading a class using its fully qualified name via reflection).

3.2.2 ❷ Individual Dependency Specialization

The dependency specialization phase receives the call graph as input to specialize individual dependencies. During this phase, DepTrim determines which dependencies are bloated (*i.e.*, there is no path from the project bytecode toward any of the class members in the unused dependencies), and removes them from the original *pom.xml* (Line 4 of Algorithm 1). Next, DepTrim proceeds to remove the unused classes within non-bloated dependencies (Lines 5 to 10). Any dependency `class` file that is not present in the call graph is deemed unreachable and removed. Note that a Java source file can contain multiple classes, thus resulting in multiple `class` files after compilation. DepTrim considers this case as well by design, as it downloads, unzips, and removes the unused compiled classes directly from the project dependencies at build time (*i.e.*, during the Maven `package` phase). Once all the unused `class` files in a dependency are removed, DepTrim qualifies the dependency as specialized. Moreover, to facilitate reuse, DepTrim deploys each specialized dependency in the local Maven repository along with its *pom.xml* file and corresponding `MANIFEST.MF` metadata (Line 11).

The output of the second phase is a set of specialized `JAR` files for the dependencies of the project. These files include all the bytecode and resources that are necessary to be shared and reused by the other packages within the dependency tree. In particular, DepTrim takes care of keeping the classes in dependencies that may not be directly instantiated by the project, but are accessible from the used classes in the dependencies, with regard to the project.

3.2.3 ❸ Dependency Tree Specialization

After specializing each non-bloated dependency, DepTrim produces a specialized version of the project *pom.xml* file that removes the bloated dependencies and points to the specialized dependencies instead of their original versions. This results in a `TST` or a `PST` for the project, as described in Definition 5.

First, DepTrim builds the totally specialized dependency tree (`TST`) of the project (Lines 12 to 15 of Algorithm 1). All specialized dependencies replace their original version in the project *pom.xml*. Then, in order to validate that the specialization did not remove necessary bytecode, DepTrim builds the project, *i.e.* its sources are compiled and its tests are run. If the build is a `SUCCESS`, DepTrim returns this `TST`.

In cases where the build with the `TST` fails, DepTrim proceeds to build the project with one specialized dependency at a time (Lines 17 to 24). Thus, rather than attempting to improve the soundness of the static call graph, which is proven to be challenging in Java [27], DepTrim performs an exhaustive search of the dependencies that are candidates for specialization. At this step, DepTrim builds as many

Algorithm 1 Third-party dependency specialization

```
Input: $\mathcal{P}_{src}$: Project source code
Input: $\mathcal{P}_{obf}$: Project original build file (pom.xml)
Output: $\mathcal{P}_{\mathsf{TST}} \vee \mathcal{P}_{\mathsf{PST}}$
    /** Call graph construction **/
 1: $\mathcal{P}_{deps} \leftarrow resolve\_dependencies(\mathcal{P}_{obf})$
 2: $\mathcal{P}_{bin} \leftarrow compile(\mathcal{P}_{src})$
 3: $\mathcal{CG} \leftarrow analyze(\mathcal{P}_{deps}, \mathcal{P}_{bin})$
 4: $\mathcal{P}_{dbf} \leftarrow debloat(\mathcal{P}_{obf}, \mathcal{CG})$
    /** Individual dependency specialization **/
 5: $\mathcal{P}_{deps\_specialized} \leftarrow \emptyset$
 6: for each $dep \in \mathcal{P}_{dbf}$ do
 7:     reachable_classes $\leftarrow analyze(dep, \mathcal{CG})$
 8:     dep_specialized $\leftarrow specialize(dep,$ reachable_classes$)$
 9:     $\mathcal{P}_{deps\_specialized} \leftarrow \mathcal{P}_{deps\_specialized} \cup$ dep_specialized
10: end for
11: $deploy\_locally(\mathcal{P}_{deps\_specialized})$
    /** Dependency tree specialization **/
12: $\mathcal{P}_{\mathsf{TST}} \leftarrow \emptyset$
13: $\mathcal{P}_{\mathsf{TST}} \leftarrow create\_config\_file(\mathcal{P}_{deps\_specialized})$
14: if $build(\mathcal{P}_{\mathsf{TST}}, \mathcal{P}_{bin}) ==$ SUCCESS then
15:     return $\mathcal{P}_{\mathsf{TST}}$
16: else
17:     $\mathcal{P}_{\mathsf{PST}} \leftarrow \emptyset$
18:     for each $dep \in \mathcal{P}_{deps\_specialized}$ do
19:         $\mathcal{P}_{dep} \leftarrow create\_config\_file(dep)$
20:         if $build(\mathcal{P}_{dep}, \mathcal{P}_{bin}) ==$ SUCCESS then
21:             $\mathcal{P}_{\mathsf{PST}} \leftarrow \mathcal{P}_{\mathsf{PST}} \cup dep$
22:         end if
23:     end for
24:     return $\mathcal{P}_{\mathsf{PST}}$
25: end if
```

versions of the dependency tree as there are specialized dependencies, each containing a single specialized dependency. DepTrim attempts to build the project with each of these single specialized dependency trees. If the project build is successful, DepTrim marks the dependency as safe to be specialized. In case the dependency is not safe to specialize, DepTrim keeps the original dependency entry intact in the specialized *pom.xml* file. Finally, DepTrim constructs a partially specialized dependency tree (`PST`) with the union of all the dependencies that are safe to be specialized. Then, the project is built with this `PST` to verify that the build is successful. If all build steps pass, DepTrim returns this `PST`.

3.3 Implementation Details

DepTrim is implemented in Java as a Maven plugin that can be integrated into a project as part of the build pipeline, or be executed directly from the command line. This design facilitates its integration as part of the projects' CI/CD pipeline, leading to specialized binaries for deployment. At its core, DepTrim reuses the state-of-the-art static analysis of DepClean [18], located in the `depclean-core` module [28]. DepTrim adds unique features to this core static Java analyzer by modifying the bytecode within dependencies based on usage information gathered at compilation time, which is different from the complete removal of unused dependencies performed by DepClean. It uses the ASM Java bytecode analysis library to build a static call graph of `class` files of the compiled projects and their dependencies. The call graph registers usage towards classes, methods, fields, and annotations. For the deployment of the specialized dependencies, DEPTRIM relies on the `deploy-file` goal of the official `maven-deploy-plugin` from the Apache Software Foundation. For dependency analysis and manipulation, DEPTRIM relies on the `maven-dependency-plugin`. DEPTRIM provides dedicated parameters to target or exclude specific dependencies for specialization, using their identifier and scope. DEPTRIM is open-source and reusable from the Maven Central repository. Its source code is publicly available at https://github.com/castor-software/deptrim.

4 EVALUATION

Depending on the outcome of specialization, DEPTRIM potentially removes large portions of the `compile`-scope dependencies of the project. The output is a specialized distribution that developers should be ready to distribute to users. The evaluation described in this section is intended to assess that experience: we run DEPTRIM on a project, build the project again with specialized dependencies to confirm that its behavior is not negatively impacted (*i.e.*, we use the test suite of the projects as a proxy for checking functional integrity), and evaluate the extent to which our technique is effective. Our evaluation is guided by the following research questions:

RQ1. What is the impact of removing bloated dependencies on reducing the ratio of third-party code?
RQ2. To what extent can all the used dependencies be specialized and the project built correctly?
RQ3. How does the number of classes decrease in the dependency tree of the project after specialization?
RQ4. In what contexts is static dependency specialization not applicable?

4.1 Study Subjects

We evaluate DEPTRIM with 30 open-source projects collected from two data sources. The first source is the dataset of single-module Java projects made available by Durieux *et al.* [29]. This dataset contains 395 popular projects that build successfully with MAVEN, *i.e.* all their tests pass, and a compiled artifact is produced as a result of the build. We analyze the dependency tree of the projects in this dataset and select those that have at least one `compile`-scope dependency. This results in 13 projects. Additionally, we derive a second set of projects through the advanced search feature of GitHub. We filter repositories with a *pom.xml* file and rank the resulting Java MAVEN projects in descending order according to the number of stars. We rely on the number of stars as a proxy for popularity [30]. Then, we curate these projects to have 17 projects that meet the following criteria: (i) build successfully with MAVEN, *i.e.* compile and all their tests pass, (ii) declare at least one `compile`-scope dependency, and (iii) have at least one test executed by the `maven-surefire-plugin`. We build the projects at least two times to avoid including projects with flaky tests. At the end of this curation process, we have a set of 30 study subjects with at least one `compile`-scope dependency, and an executable test suite with tests that pass.

Table 1 presents descriptive statistics for the 30 study subjects. For multi-module projects, we specify the MODULE we use for our experiments. Furthermore, we link to the COMMIT SHA of the version that we consider for the evaluation. The explicit documentation of PROJECT, MODULE, and COMMIT SHA ensure the reproducibility of our evaluation. The projects are well-known in the Java community and have between 155 and 20,488 STARS, for `commons-validator` and `flink` respectively. The median number of stars is 2,751. `flink` also has the maximum number of COMMITS, at 32,667, while the median number of commits across the study subjects is 2,544. Next, we report the number of lines of Java code (LOC) in each project, computed with the Unix command `cloc`. In total, the projects have more than 2 M LOC. The two projects with the largest number of lines of code are `CoreNLP` (605,561) and `checkstyle` (342,795), while the median LOC across the projects is 32,965. In the TESTS column we give the number of tests executed by the official `maven-surefire-plugin` in the projects. The median number of tests is 599. The two projects with the most tests are `jimfs` (5,834) and `checkstyle` (3,887). In the COV. column we provide the code coverage of the test suite of the project, as measured with `JaCoCo`[1]. The median coverage for the 30 study subjects is 64%. When we study one specific module of a multi-module project, the LOC, TESTS, and COV. numbers are for the specific module under study.

The last 4 columns of Table 1 provide dependency-specific information. First, the number of `compile`-scope dependencies as resolved by MAVEN (#CD). There are 467 `compile`-scope dependencies across the 30 projects, with a median number of 9 CDs and at least 2 CDs in each project. The maximum number of `compile`-scope dependencies is 56, in `Recaf`. The following columns present the number of CLASSES that are written by the developers of the PROJECT, and the number of third-party classes that come from its `compile`-scope dependencies (CD). The bytecode of each of these classes is analyzed by DEPTRIM in order to construct a static call graph of APIs usages between the projects and dependencies, as described in Section 3.2.1. In total, DEPTRIM analyzes the bytecode of 15,594 project classes, and 135,343 classes from third-party dependencies. `CoreNLP` has 3,932 project classes, the maximum in the dataset. The largest number of third-party classes is 17,512, in `OpenPDF`. In the last column of the table, we present the dependency classes to project classes ratio (RATIO$_\mathcal{O}$ in Equation 1).

$$\text{RATIO}_\mathcal{O} = \frac{\#\text{CD CLASSES}}{\#\text{PROJECT CLASSES}} \qquad (1)$$

We find that, for 27 of these 30 notable projects, most of the code actually belongs to third-party dependencies. In fact, this ratio is as high as 206.7 × for the project `tablesaw`. Across our dataset, the ratio of the project classes to the dependency classes is 8.7 ×.

Recalling the example of `jacop` introduced in Figure 1, the corresponding row in Table 1 reads as follows: we select its latest release for our evaluation (SHA `6ed0cd0`), which has 1,302 commits, 93,170 lines of Java code, 210 tests, and has been starred by 202 users on GitHub. When `jacop` is compiled, the number of classes from `jacop` is 833. On the other hand, its 9 `compile`-scope dependencies contribute 10.2 × more classes (*i.e.*, 8,487) compared to the classes in the project (*i.e.*, 833).

1. https://www.eclemma.org/jacoco/

Table 1: Description of the study subjects considered for the evaluation of DepTrim. The table links to the project repository and SHA on GitHub, and lists the number of commits, stars, lines of Java code (LoC), tests, test coverage (Cov), and the original number of `compile`-scope dependencies in the project (#CD). Also indicated are the number of project classes in each study subject, the total number of classes contributed by CDs, as well as the ratio between them ($\text{Ratio}_{\mathcal{O}}$).

Project	Module	Commit SHA	Commits	Stars	LoC	Tests	Cov. (%)	#CD	Classes		
									Project	CD	$\text{Ratio}_{\mathcal{O}}$
`checkstyle`	-	`6ec5122`	12,066	7,455	342,795	3,887	79	17	863	6,493	7.5 ×
`Chronicle-Map`	-	`63c1c60`	3,298	2,539	55,178	1,231	49	35	375	7,595	20.3 ×
`classgraph`	-	`8a24786`	5,307	2,366	32,151	170	61	2	261	224	0.9 ×
`commons-validator`	-	`33ecc88`	1,742	155	16,781	576	85	4	64	780	12.2 ×
`CoreNLP`	-	`013556a`	17,012	8,802	605,561	1,374	18	32	3,932	9,121	2.3 ×
`flink`	`flink-java`	`1d6e2b7`	32,667	20,488	36,455	836	56	16	277	6,175	22.3 ×
`graphhopper`	`core`	`bff8747`	6,211	3,978	66,119	2,460	81	18	631	5,474	8.7 ×
`guice`	`core`	`5f73d8a`	2,026	11,730	49,697	979	90	10	460	2,474	5.4 ×
`helidon-io`	`openapi`	`070f2bb`	2,707	2,929	7,729	30	76	36	32	4,002	125.1 ×
`httpcomponents`	`httpclient5`	`cb8bdf7`	3,424	1,269	42,920	669	50	5	493	1,153	2.3 ×
`immutables`	`gson`	`413aa37`	2,588	3,218	16,448	37	60	2	31	307	9.9 ×
`jacop`	-	`6ed0cd0`	1,302	202	93,170	210	34	9	833	8,487	10.2 ×
`java-faker`	-	`b0b9e6e`	834	3,914	8,429	579	98	4	107	503	4.7 ×
`jcabi-github`	-	`462d724`	2,764	276	33,542	684	44	20	312	3,921	12.6 ×
`jimfs`	`jimfs`	`9ef38d1`	508	2,234	15,558	5,834	91	9	124	3,560	28.7 ×
`jooby`	`jooby`	`1c78357`	4,702	1,523	20,154	122	31	22	320	6,945	21.7 ×
`lettuce`	`core`	`fc94fcb`	2,280	4,861	89,468	2,600	42	44	1,302	10,364	8.0 ×
`modelmapper`	`core`	`03663ee`	721	2,090	21,769	618	84	6	210	2,700	12.9 ×
`mybatis-3`	-	`2655970`	4,436	18,065	61,849	1,699	86	8	480	1,345	2.8 ×
`OpenPDF`	-	`bd0d458`	1,296	2,573	76,397	35	29	35	484	17,512	36.2 ×
`pdfbox`	`pdfbox`	`af1ff57`	11,147	1,852	97,175	654	79	7	754	6,836	9.1 ×
`pf4j`	-	`fd00c63`	692	1,901	7,199	151	73	3	93	115	1.2 ×
`poi-tl`	-	`71b5969`	732	3,063	20,882	125	79	36	255	12,143	47.6 ×
`Recaf`	-	`a30dce0`	2,275	4,530	31,277	274	35	56	538	10,769	20.0 ×
`RxRelay`	-	`09428b5`	81	2,473	2,405	64	93	2	16	1,758	109.9 ×
`scribejava`	-	`7a6185b`	1,259	5,317	5,769	82	45	8	116	1,278	11.0 ×
`tablesaw`	`json`	`80d5334`	2,501	3,101	508	9	75	9	7	1,447	206.7 ×
`tika`	`tika-core`	`dd04a3e`	6,823	1,584	32,388	305	50	2	435	253	0.6 ×
`undertow`	`core`	`cce54c6`	5,517	3,284	106,711	682	59	5	1,581	742	0.5 ×
`woodstox`	-	`58bd89e`	325	180	60,476	868	67	5	208	864	4.2 ×
Total	14	30	139,243	127,952	2,056,960	27,844	(Med.) 64	467	15,594	135,343	8.7 ×

4.2 Protocol for RQ1

With this research question, we quantify the potential for dependency specialization in the 30 projects described in Table 1. In order to do so, we use DepClean to identify and remove bloated dependencies from each project, ensuring that the project still builds. We report the number of `compile`-scope dependencies that are non-bloated, denoted as NBCD. Next, we present the total number of classes removed through dependency debloating (Classes Removed), and compute the ratio ($\text{Ratio}_{\mathcal{D}}$) between the remaining dependency classes and the project classes (per Equation 2). This data provides quantitative insights regarding the impact of dependency debloating to reduce the share of third-party code, and on the opportunity to reduce this share further via dependency specialization.

$$\text{Ratio}_{\mathcal{D}} = \frac{\#\text{CD Classes} - \#\text{Classes Removed by DepClean}}{\#\text{Project Classes}} \quad (2)$$

4.3 Protocol for RQ2

In order to answer RQ2, we attach DepTrim to the Maven build lifecycle of each of our study subjects. DepTrim is implemented as a Maven plugin, which facilitates this integration, as described in Section 3.3. This means that the non-bloated `compile`-scope dependencies in the dependency tree of each project are resolved, specialized, and deployed to the local Maven repository. DepTrim then attempts to build the project, *i.e.*, compile it and run its tests, with the goal of preparing a specialized dependency tree with the maximum number of specialized dependencies. Per Algorithm 1, DepTrim constructs either a totally specialized tree (`TST`), or a partially specialized tree (`PST`) that includes the largest number of specialized dependencies that preserve the build correctness. For each project, we report whether it builds with a `TST`. If it does not, we report the number of dependencies that DepTrim successfully specializes to prepare a `PST` (through the metric NBCD Specialized). The findings from this research question highlight the applicability of DEPTRIM on real-world MAVEN projects, and its ability to prepare minimal versions of these projects, by removing unused classes within non-bloated dependencies while passing the build.

4.4 Protocol for RQ3

After building each project successfully with a `TST` or a `PST` in RQ2, we report the total number of classes that are removed by DEPTRIM through the specialization of its non-bloated `compile`-scope dependencies (as CLASSES REMOVED). We also report the ratio of the remaining number of specialized dependency classes to the number of project classes (RATIO$_{\mathcal{S}}$ in Equation 3). We compare RATIO$_{\mathcal{S}}$ with RATIO$_{\mathcal{O}}$, *i.e.*, we evaluate the reduction in the original ratio after specialization. This research question demonstrates the practical advantages of dependency specialization with DEPTRIM, specifically the reduction in the original proportion of third-party classes within the compiled project binary.

$$\text{RATIO}_{\mathcal{S}} = \frac{\#\text{NBCD CLASSES} - \#\text{CLASSES REMOVED BY DEPTRIM}}{\#\text{PROJECT CLASSES}} \quad (3)$$

4.5 Protocol for RQ4

While processing each project, DEPTRIM records the project build logs after dependency specialization (RQ2), as well as the number of classes removed from each non-bloated dependency (RQ3). We derive the answer for RQ4 by analyzing these logs. In some cases, all the classes in a non-bloated dependency are used by the project, leaving no room for specialization. We refer to such a dependency as a totally used dependency (TUD). We report the number of TUDs for each project, where DEPTRIM is not applicable by design. Another situation where DEPTRIM is not applicable is when a project uses dynamic features to access dependency classes (Section 3.2.1). While computing the `PST` for RQ2, DEPTRIM builds the project multiple times, each time with a single specialized dependency. In case of a failure when building the project with a specialized dependency, we report a compilation error or a test failure. For the assessment of the compilation results, we rely on the official `maven-compiler-plugin`. We consider the execution of the test suite to fail if there is at least one test reported within the sets of `Failures` or `Errors`, as reported by the official `maven-surefire-plugin`. With this research question, we gain insights regarding the existing challenges of dependency specialization with DEPTRIM. More generally, it contributes to the understanding of the limitations of static analysis with respect to specializing dependencies, in view of the dynamic features of Java.

4.6 Evaluation Framework

In order to run our experiments, we have designed a fully automated framework that orchestrates the execution of DEPTRIM, the creation of specialized dependency trees, the building of the projects with the specialized dependency trees, as well as the collection and processing of data to answer our research questions. Since DEPTRIM is implemented as a MAVEN plugin, it integrates within the MAVEN build lifecycle and executes during the `package` phase. The execution was performed on a virtual machine running Ubuntu Server with 16 cores of CPU and 32 GB of RAM. It took one week to execute the complete experiment with the 30 study subjects. This execution time is essentially due to multiple executions of the large test suites of our subjects: once with the original project; once after debloating dependencies with DEPCLEAN; once with the `TST` generated by DEPTRIM, and if we generate a `PST` for a project, we run the test suite once with each individually specialized tree and once with the final `PST`. The execution framework is publicly available on GitHub at castor-software/deptrim-experiments, and the raw data obtained from the complete execution is available on Zenodo at 10.5281/zenodo.7613554.

5 EXPERIMENTAL RESULTS

This section presents the results from our evaluation of DEPTRIM with the 30 Java projects described in Section 4.1. We evaluate the effectiveness of DEPTRIM in automatically specializing the dependency tree of these projects. The answers to the four RQs are summarized in Table 2.

5.1 RQ1: What is the impact of removing bloated dependencies on reducing the ratio of third-party code?

With this first research question, we set a baseline to assess the impact of dependency specialization regarding the reduction of the number of classes in third-party dependencies. To do so, we report the number of classes removed through state-of-the-art dependency debloating with DEPCLEAN, as described in Section 4.2. We report the ratio of third-party classes remaining after debloating, with respect to the number of classes in each project presented in Table 1.

For our 30 study subjects, the column NBCD in Table 2 denotes the number of `compile`-scope dependencies that remain after identifying and removing bloated dependencies with DEPCLEAN, over the original number of `compile`-scope dependencies in the project (column #CD in Table 1). In total, DEPCLEAN removes 71 bloated dependencies, with a median of 8 dependencies, across the 30 projects. DEPCLEAN removes 23 bloated dependencies from `OpenPDF`, which is the largest number of bloated dependencies for one project in our dataset. In total, DEPCLEAN removes 5,813 third-party classes. It is interesting to note that, for all the projects, dependency debloating removes 4.3 % of the total number of classes.

All projects have at least 2 NBCDs, while `Recaf` has the maximum number of NBCDs at 41. In 13 projects, such as `classgraph` and `commons-validator`, all the dependencies are used. Therefore, executing DEPCLEAN does not contribute to the removal of any class on those projects. On the other hand, we find that in 5 projects, the bloated dependencies do not contain `class` files at all, such as in the case of `flink`. This happens when a bloated dependency only contains assets, such as resource files, or is explicitly designed to avoid conflicts with other dependencies [31]. For example, the dependency `com.google.guava:listenablefuture` is present in the dependency tree of 5 projects, and it is intentionally empty to avoid conflicts with `guava` [32]. Another dependency, called `batik-shared-resources`, is included in the dependency tree of 2 projects, and only

Table 2: Results from the evaluation of DEPTRIM with the case studies described in Table 1. For RQ1, the table indicates the number of non-bloated `compile`-scope dependencies (NBCD). These are the dependencies that are identified as used in the project by DEPCLEAN. The number of classes removed via debloating is listed in the column CLASSES REMOVED. RATIO$_{\mathcal{D}}$ represents the number of remaining third-party classes after debloating over the number of classes in the project. For RQ2, we highlight whether DEPTRIM builds a project with a `TST` or a `PST`, as well as the number of specialized NBCDs contained in the built project (NBCD SPECIALIZED). RQ3 presents the reduction in the number of classes in the NBCDs as a result of specialization with DEPTRIM, and correspondingly, in the RATIO$_{\mathcal{S}}$ of the third-party classes to project classes. For RQ4, we report the three cases where specialization with DEPTRIM is not applicable: (i) if all the classes in a non-bloated dependency are totally used by the project (TUD); (ii) if specializing a dependency causes a compilation error (COMP. ERROR) during project build; and (iii) if a test fails when building a project with a specialized dependency (TEST FAIL.).

PROJECT	RQ1			RQ2			RQ3		RQ4		
	NBCD	CLASSES REMOVED (DEBLOATING)	RATIO$_{\mathcal{D}}$	TST	PST	NBCD SPECIALIZED	CLASSES REMOVED (SPECIALIZATION)	RATIO$_{\mathcal{S}}$	TUD	COMP. ERROR	TEST FAIL.
`checkstyle`	15/17	2 (0.0 %)	7.5 ×	✗	✓	12/15	1,015 (15.6 %)	6.3 ×	0/15	2/3	1/3
`Chronicle-Map`	28/35	373 (4.9 %)	19.3 ×	✗	✓	22/28	2,573 (35.2 %)	12.4 ×	4/28	1/2	1/2
`classgraph`	2/2	0 (0.0 %)	0.9 ×	✓		2/2	10 (4.5 %)	0.8 ×	0/2	–	–
`commons-validator`	4/4	0 (0.0 %)	12.2 ×	✓		4/4	625 (80.1 %)	2.4 ×	0/4	–	–
`CoreNLP`	30/32	364 (4.0 %)	2.2 ×	✓		29/30	3,648 (41.7 %)	1.3 ×	1/30	–	–
`flink`	15/16	0 (0.0 %)	22.3 ×	✗	✓	12/15	2,594 (42.0 %)	12.9 ×	1/15	1/2	1/2
`graphhopper`	13/18	1,309 (23.9 %)	6.6 ×	✗	✓	12/13	1,661 (39.9 %)	4.0 ×	0/13	1/1	0/1
`guice`	9/10	0 (0.0 %)	5.4 ×	✓		7/9	1,327 (53.6 %)	2.5 ×	2/9	–	–
`helidon-io`	34/36	38 (0.9 %)	123.9 ×	✗	✓	32/34	987 (24.9 %)	93.0 ×	1/34	1/1	0/1
`httpcomponents`	5/5	0 (0.0 %)	2.3 ×	✓		4/5	432 (37.5 %)	1.5 ×	1/5	–	–
`immutables`	2/2	0 (0.0 %)	9.9 ×	✓		2/2	48 (15.6 %)	8.4 ×	0/2	–	–
`jacop`	4/9	504 (5.9 %)	9.6 ×	✗	✓	3/4	5,704 (71.5 %)	2.7 ×	0/4	1/1	0/1
`java-faker`	4/4	0 (0.0 %)	4.7 ×	✗	✓	3/4	211 (41.7 %)	2.8 ×	0/4	1/1	0/1
`jcabi-github`	17/20	9 (0.2 %)	12.5 ×	✗	✓	16/17	2,415 (61.7 %)	4.8 ×	0/17	1/1	0/1
`jimfs`	8/9	0 (0.0 %)	28.7 ×	✓		6/8	1,741 (48.9 %)	14.7 ×	2/8	–	–
`jooby`	20/22	0 (0.0 %)	21.7 ×	✗	✓	19/20	730 (10.5 %)	19.4 ×	0/20	1/1	0/1
`lettuce`	39/44	0 (0.0 %)	8.0 ×	✗	✓	36/39	1,865 (18.0 %)	6.5 ×	2/39	0/1	1/1
`modelmapper`	6/6	0 (0.0 %)	12.9 ×	✗	✓	4/6	66 (2.4 %)	12.5 ×	1/6	0/1	1/1
`mybatis-3`	8/8	0 (0.0 %)	2.8 ×	✗	✓	7/8	414 (30.8 %)	1.9 ×	0/8	0/1	1/1
`OpenPDF`	12/35	2,336 (13.3 %)	31.4 ×	✗	✓	11/12	9,155 (60.3 %)	12.4 ×	0/12	1/1	0/1
`pdfbox`	6/7	63 (0.9 %)	9.0 ×	✓		6/6	5,070 (74.9 %)	2.3 ×	0/6	–	–
`pf4j`	3/3	0 (0.0 %)	1.2 ×	✓		2/3	10 (8.7 %)	1.1 ×	1/3	–	–
`poi-tl`	33/36	258 (2.1 %)	46.6 ×	✗	✓	27/33	5,192 (43.7 %)	26.2 ×	5/33	0/1	1/1
`Recaf`	49/56	518 (4.8 %)	19.1 ×	✓		41/49	2,952 (28.8 %)	13.6 ×	8/49	–	–
`RxRelay`	2/2	0 (0.0 %)	109.9 ×	✗	✓	1/2	9 (0.5 %)	109.3 ×	0/2	0/1	1/1
`scribejava`	7/8	39 (3.1 %)	10.7 ×	✓		6/7	353 (28.5 %)	7.6 ×	1/7	–	–
`tablesaw`	9/9	0 (0.0 %)	206.7 ×	✓		7/9	379 (26.2 %)	152.6 ×	2/9	–	–
`tika`	2/2	0 (0.0 %)	0.6 ×	✓		2/2	187 (73.9 %)	0.2 ×	0/2	–	–
`undertow`	5/5	0 (0.0 %)	0.5 ×	✓		5/5	224 (30.2 %)	0.3 ×	0/5	–	–
`woodstox`	5/5	0 (0.0 %)	4.2 ×	✗	✓	3/5	34 (3.9 %)	4.0 ×	0/5	1/2	1/2
TOTAL	396/467	5,813 (4.3 %)	8.3 ×	14/30	16/30	343 (86.6 %)	51,631 (39.9 %)	5.0 ×	32/396	12/21	9/21

contains resource files. We investigate the nature of these resource files and find that they are dependency license statements and build-related metadata. Thus, the removal of such dependencies does not result in a build failure within the projects.

The column RATIO$_{\mathcal{D}}$ in Table 2 presents the ratio of the number of classes in the NBCDs to the original number of classes in the project (column PROJECT in Table 1). For 11 of the 30 projects, RATIO$_{\mathcal{D}}$ is less than RATIO$_{\mathcal{O}}$ from Table 1. This corresponds to cases where debloating results in fewer third-party classes in the compiled project. For example, the removal of the 23 bloated dependencies from `OpenPDF` results in the maximum reduction in the number of classes (2,336). Consequently, RATIO$_{\mathcal{D}}$ for `OpenPDF` is 31.4 ×, which is 4.8 less than its RATIO$_{\mathcal{O}}$. The project with the highest percentage of classes removed is `graphhopper`, for which the removal of 5 bloated dependencies leads to a 23.9 % reduction in the number of third-party classes. RATIO$_{\mathcal{D}}$ for `graphhopper` is 6.6 ×, down from its original RATIO$_{\mathcal{O}}$ of 8.7 ×. However, despite debloating dependencies, the total RATIO$_{\mathcal{D}}$ across the 30 projects is 8.3 ×, which is only 0.4 less than the total original RATIO$_{\mathcal{O}}$.

Of the 9 `compile`-scope dependencies in `jacop` (column #CD in Table 1), DEPCLEAN identifies 5 dependencies as bloated and removes them. This leads to the removal of 504 classes, and 4 remaining NBCDs with a total of 7,983 classes. Correspondingly, $\text{RATIO}_{\mathcal{D}}$ reduces to 9.6 ×, down from the original $\text{RATIO}_{\mathcal{O}}$ of 10.2 ×. The 4 NBCDs are the target for specialization with DEPTRIM, which will remove unused classes within these dependencies while ensuring that `jacop` still correctly builds.

Answer to RQ1: State-of-the-art dependency debloating with DEPCLEAN contributes to the removal of 71 bloated dependencies from 30 real-world Java projects. This corresponds to the removal of 5,813 (4.3 %) third-party classes in total. Yet, the dependency classes to project classes ratio is reduced by only 0.4 (from 8.7 × to 8.3 ×). This calls for more extensive code removal to reduce the dependencies to the strictly necessary parts.

Table 3: Dependencies specialized in the `OpenPDF` project

DEPENDENCY	TYPE	CLASSES REMOVED
`xmlgraphics-commons`	Transitive	366/375 (97.6 %)
`bcutil-jdk18on`	Transitive	532/579 (91.9 %)
`fop-core`	Transitive	2,278/2,547 (89.4 %)
`bcpkix-jdk18on`	Direct	697/841 (82.9 %)
`bcprov-jdk18on`	Direct	5,696/7,149 (79.7 %)
`xml-apis`	Transitive	234/346 (67.6 %)
`fontbox`	Transitive	100/157 (63.7 %)
`xalan`	Transitive	942/1,501 (62.8 %)
`icu4j`	Direct	578/1,555 (37.2 %)
`serializer`	Transitive	69/108 (63.9 %)
`commons-logging`	Transitive	6/18 (33.3 %)
`fop`	Transitive	N/A
TOTAL	3 D/9 T	9,155/15,176 (60.3 %)

5.2 RQ2: To what extent can all the used dependencies be specialized and the project built correctly?

This research question evaluates the ability of DEPTRIM to perform automatic dependency specialization for the study subjects described in Section 4.1. We consider the specialization procedure to be successful if DEPTRIM produces a valid set of specialized dependencies, with a corresponding specialized dependency tree captured in a *pom.xml*, and for which the project builds correctly. To reach this successful state, the project to be specialized must pass through all the build phases of the MAVEN build lifecycle, *i.e.*, compilation, testing, and packaging, according to the protocol described in Section 4.2.

Columns `TST`, `PST`, and NBCD SPECIALIZED in Table 2 present the results obtained. First, we observe that for a total of 14 (46.7 %) projects, DEPTRIM produces a totally specialized tree (`TST`), *i.e.*, the project builds successfully with a specialized version of all its non-bloated `compile`-scope dependencies. For these projects, DEPTRIM successfully identifies and removes unused classes within the dependencies. Moreover, DEPTRIM updates the dependency tree of each project by replacing original dependencies with specialized ones. The projects correctly compile, and their original test suite still passes, indicating that their behavior is intact *w.r.t.* the tests, despite the dependency tree specialization. Overall, these results confirm that dependency specialization is feasible for real-world projects.

We illustrate `TST`s with the example of `pdfbox`, a utility library and tool to manipulate PDF documents. DEPTRIM specializes the 6 NBCDs of `pdfbox`, and builds its `TST` successfully. Of these 6 dependencies, 4 are direct: `fontbox`, `commons-logging`, `pdfbox-io`, and `bcprov-jdk18on`; whereas 2 are transitive: `bcutil-jdk18on` and `bcpkix-jdk18on`. Another interesting example is `guice`, a popular dependency injection framework from Google branded as a "lightweight" alternative to existing libraries, as stated in its official documentation. DEPTRIM builds `guice` with a `TST`, thus making it even smaller. The project that builds with a `TST` and has the largest number of specialized dependencies is `Recaf` with 41 specialized dependencies. Note that, when specializing transitive dependencies, DEPTRIM keeps all the classes in the direct dependencies that are necessary to access the APIs in transitive dependencies, whether directly or indirectly. For example, the transitive dependency `commons-lang3` in `Recaf` is resolved and used from the direct dependency `jphantom`, a Java library for program complementation [33]. Thus, DEPTRIM keeps the bytecode in `commons-lang3` that is necessary to access the used features provided by `jphantom`.

On the other hand, projects that do not build with a `TST` signify cases where at least one `compile`-scope dependency relies on dynamic Java features that make static analysis unsound. This observation is in line with previous work showing that Java reflection and other dynamic features impose limitations on performing static analysis in the Java ecosystem [34]. However, even for these projects, DEPTRIM successfully builds a partially specialized tree (`PST`) by targeting dependencies that could be specialized, and discarding the ones that cause the build to break. In total, 16 (53.3 %) of the projects build successfully with a `PST`. In these cases, DEPTRIM successfully identifies the subset of dependencies that are safe for specialization, and validates that the projects still correctly build with a `PST`.

For example, DEPTRIM successfully specializes 4 dependencies in the project `modelmapper`, an object mapping library that automatically maps objects to each other. DEPTRIM creates a `PST` with which `modelmapper` builds successfully. Note that none of the 6 `compile`-scope dependencies of `modelmapper` are bloated, and hence debloating the project with DEPCLEAN has no impact on it. However, after executing DEPTRIM, the direct dependencies `objenesis` and `asm-tree` are specialized. Moreover, the transitive dependencies `asm-commons` and `asm`, resolved from `asm-tree` are also specialized. This example illustrates the impact of specialization beyond dependency debloating, for projects that build with a `PST`. Indeed, across our study subjects, there are 8 projects that successfully build with a `TST`, and 5 that build with a `PST`, and yet for which no classes are removed through DEPCLEAN.

It is interesting to notice that some of our study subjects share dependencies that are specialized. For example, `slf4j-api` is specialized in 12 different projects, `jsr305` in 8 projects, and `commons-io` in 4 projects. The projects `jcabi-github`, `jooby`, and `Recaf` include all these three dependencies in their dependency tree. After investigating the contents of the specialized versions of `slf4j-api` prepared by DEPTRIM, we find that there are three sets of variants for which this dependency contains the same number of classes. Thus, deploying multiple specialized versions of `slf4j-api` to external repositories can contribute to reducing its attack surface for projects that reuse the exact same features. This specialized form of code reuse also increases software diversity. Furthermore, the dependency `bcprov-jdk18on`, which contains the largest number of classes among the dependencies (3,768), is successfully specialized in 2 projects, `OpenPDF` and `pdfbox`. Our findings suggest that specializing dependencies with a large number of classes yields a greater reduction of third-party code. To confirm this hypothesis, additional investigation is required.

Our experiments show that, despite the challenges of specializing the dependency trees of our 30 real-world study subjects, DepTrim is capable of specializing 343 of the 396 non-bloated `compile`-scope dependencies across them. A key aspect of our evaluation is that we validate that each project builds successfully using its specialized dependency tree. We manually analyze and classify the cases where specialization is not achievable for a dependency, in RQ4. The specialized dependencies contribute to the deployment of smaller project binaries, to reduce their attack surface, and to increase dependency diversity when deployed to external repositories.

Answer to RQ2: DepTrim successfully builds 14 real-world projects with a totally specialized dependency tree. For the other 16 projects, DepTrim finds the largest subset of specialized dependencies that do not break the build. In total, DepTrim specializes 343 (86.6 %) of the non-bloated `compile`-scope dependencies. This is evidence that a large majority of dependencies in Java projects can be specialized without impacting the project build.

5.3 RQ3: How does the number of classes decrease in the dependency tree of the project after specialization?

To answer our third research question, we count the number of classes removed by DepTrim in the 343 successfully specialized dependencies. The goal is to evaluate the effectiveness of DepTrim in removing unused `class` files through specialization, as described in Section 4.3. We also report the impact of this reduction on the ratio of third-party classes to project classes, *i.e.*, $\text{Ratio}_{\mathcal{S}}$.

The column Classes Removed in Table 2 shows the number of classes removed by DepTrim from the NBCDs of each project, in order to build its `TST` or `PST`. DepTrim removes a total of 51,631 classes, with a median removal of 858 third-party classes for each project. This represents 39.9 % of the total number of classes in the third-party dependencies for all the projects (*i.e.*, 135,343 per Table 1). For example, the project `tika` has 2 dependencies specialized: `slf4j-api` with 9/52 (17.3 %) classes removed, and `commons-io` with 178/201 (88.6 %) classes removed. This represents a removal of 187 (73.9 %) third-party classes in `tika`, as a result of which its $\text{Ratio}_{\mathcal{S}}$ is 0.2 × . Thus, the ratio of dependency classes to project classes in `tika` decreased by 0.4 compared to $\text{Ratio}_{\mathcal{D}}$ (*i.e.*, 0.6 ×).

The project with the highest percentage of dependency classes removed is `commons-validator` with 80.1 %, *i.e.*, 625 of the 780 original third-party classes.

Table 4: Summary of global impact of DepTrim: number of classes in the compile-scope dependencies (CD) and size of the bytecode of the compile-scope dependencies (CD) in the original projects, and impact of DepTrim in reducing the number of classes and the size of bytecode for dependencies.

	Min.	Max.	Med.	Total
Original				
# CD classes	115	17,512	3,130	135,343
CD bytecode size (MB)	0.4	49.5	9.7	455.9
Impact of DepTrim				
# CD classes removed	9	11,491	873.5	57,444 (42.2 %)
CD bytecode reduction (MB)	0.02	37.1	1.9	186.1 (35.7 %)

This drastic reduction is the result of successfully removing unused classes from all its `compile`-scope dependencies: `commons-beanutils`, `commons-collections`, `commons-digester` and `commons-logging`. In particular, DepTrim identifies that only 7 of the 460 classes in `commons-collections` are required by `commons-validator`. These classes support an implementation of the `HashMap` data structure for multi-threaded operations, which are used by `commons-validator` for processing form fields. The other 453 types within `commons-collections` correspond to data structures that are not required by `commons-validator`, and are consequently removed by DepTrim.

The project with the largest number of classes removed (9,155) is `OpenPDF`. Table 3 shows the dependencies specialized in `OpenPDF`, of which 3 are direct and 9 are transitive. DepTrim builds `OpenPDF` with a `PST`, excluding the dependency `fop` from the specialized dependency tree. Looking at the 11 successfully specialized dependencies, we observe that `OpenPDF` depends transitively on a family of dependencies from the Apache XML Graphics Project, including `fop-core` and `xmlgraphics-commons`, from which DepTrim removes 2,278 (89.4 %) and 366 (97.6 %) unused classes, respectively. `OpenPDF` also depends directly on `bcpkix-jdk18on` and `bcprov-jdk18on`, which are dependencies from the Bouncy Castle family of cryptographic libraries, from which 697 (82.9 %) and 5,696 (79.7 %) classes are removed, respectively. Moreover, DepTrim systematically identifies functionalities that are used transitively through direct dependencies. For example, two classes within `OpenPDF`, called `PdfPKCS7` and `TSAClientBouncyCastle`, use classes from the direct dependency `bcpkix-jdk18on`. In turn, these classes of `bcpkix-jdk18on` depend on 4 classes within `bcutil-jdk18on` that are responsible for supporting the encoding of the Time Stamp Protocol. Therefore, DepTrim marks these 4 classes within the transitive dependency as necessary for `OpenPDF`, and does not remove them. Note that `OpenPDF` built with a specialized dependency tree may be deployed to an external repository, which reduces the attack surface of the clients that rely on the features that are provided by `OpenPDF` when used as a library.

Overall, the specialization of non-bloated dependencies can significantly reduce the share of third-party classes, beyond state-of-the-art dependency debloating techniques such as DepClean [24]. This is evidenced in the last row of Table 2, where we report the removal of 51,631 classes, just through specialization, which represents 39.9 % of the classes in non-bloated dependencies. While these observations are compelling evidence of the benefits of specialization, we now reflect upon the overall effect of DEPTRIM.

Table 4 summarizes the key metrics about the impact of DEPTRIM on dependency trees. First, we provide the distribution of the number of classes in compile-scope dependencies (CD), as well as the distribution of the size of the bytecode for these dependencies. Second, we provide the global reduction of the number of classes and the size of the bytecode for third-party dependencies. While we focused on the number of classes so far, here we also include the impact on the size of the bytecode, as it is an important performance metric for some applications. Specifically, the table highlights the minimum, maximum, median, and total values of CD classes and bytecode sizes. For example, DEPTRIM removes 11,491 CD classes in total for `OpenPDF`, which is the maximum number of CD classes removed. Per Table 2, we see that 2,336 classes were removed through dependency debloating, while 9,155 classes were removed the specialization of 11 dependencies. In `RxRelay`, DEPTRIM removes only 9 classes, exclusively by specializing one dependency. In total, DEPTRIM succeeds in removing 42.2 % of the CD classes. This is a significant reduction in the prevalence of third-party code in the Java projects under study. For `jacop`, DEPTRIM achieves the largest reduction in the size of third-party bytecode, removing 37.1 MB. This is essentially due to large dependencies in `jacop` such as `scala-compiler`, for which DEPTRIM removes 2,982 classes (see Figure 1c). Note that DEPTRIM removes more than 1.9 MB of third-party bytecode for half the study subjects. In total, DEPTRIM removes 186.1 MB of third-party bytecode, which corresponds to a reduction in 35.7 %.

Answer to RQ3: DEPTRIM reduces the number of classes in the dependency tree of each of the 30 projects. Overall, by adding bytecode specialization to dependency debloating, DEPTRIM reduces dependency classes by a total of 42.2 % and the size of dependency bytecode by 35.7 %. The dependency classes to project classes ratio reduces from 8.7 × in the original project to 5.0 × . Dependency specialization drastically reduces the share of third-party bytecode in Java projects.

5.4 RQ4: In what contexts is static dependency specialization not applicable?

With this research question we report on the cases where there is no scope for specialization in a non-bloated dependency, as well as cases where projects do not build successfully with a specialized dependency in the dependency tree.

First, we observe that 14 projects include at least one dependency that is totally used. A total of 32 dependencies are totally used by their respective client projects, as presented in the column TUD in Table 2. A dependency is a TUD for a project if all its `class` files are exercised by the project. Consequently, there is no scope for the specialization of a TUD. TUDs represent 8.1 % of the non-bloated dependencies. `Recaf` has 8 TUDs, the largest number in the study subjects. Note that a project with TUDs in its dependency tree can still successfully build with a `TST`, as is true for 8 projects, including `Recaf`. We observe that the dependencies `asm-tree`, `failureaccess`, and `minimal-json` are TUDs in 2 projects. For example, `minimal-json` is totally used by both `tablesaw` and by `Recaf`, which is evidence of a minimal API that is completely used by these projects. As far as we know, this is the first time in the literature that totally used dependencies are identified and quantified.

Table 5: Number of unique failing tests, and the specialized dependency that causes these failures, in the 9 projects with tests failures.

PROJECT	DEPENDENCY	# TEST FAIL
`checkstyle`	`Saxon-HE`	1/3,887 (0.0 %)
`Chronicle-Map`	`chronicle-wire`	3/1,231 (0.2 %)
`flink`	`commons-math3`	1/820 (0.1 %)
`lettuce-core`	`micrometer-core`	6/2,600 (0.2 %)
`modelmapper`	`byte-buddy-dep`	4/618 (0.6 %)
`mybatis-3`	`slf4j-api`	1/1,699 (0.1 %)
`poi-tl`	`commons-io`	107/125 (85.6 %)
`RxRelay`	`rxjava`	6/64 (9.4 %)
`woodstox`	`msv-core`	1/868 (0.1 %)
TOTAL	9	130/11,912 (1.1 %)

DEPTRIM builds 16 projects with a `PST`. For example, DEPTRIM marks 1 of the 4 non-bloated `compile`-scope dependencies in `java-faker` as not suitable for specialization. This is because building `java-faker` with the specialized version of `org.yaml:snakeyaml` prevents the compilation of the project, which includes the `Lebowski` class. Consequently, `org.yaml:snakeyaml` is excluded from the specialized dependency tree of `java-faker` and DEPTRIM outputs a partially specialized tree that successfully builds `java-faker` and includes three specialized dependencies.

Column COMP. ERROR in Table 2 shows the number of specialized dependencies for which the build fails due to compilation errors. This occurs for 11 projects and 12 dependencies. We investigate the causes of compilation errors by manually analyzing the logs of the `maven-compiler-plugin`. We find the following 4 causes for compilation to fail:

- Some classes are not found during compilation. For example, attempting to build `checkstyle` with specialized versions of `commons-beanutils` and `guava` fails due to the missing classes, `BasicDynaBean` and `ClassPath`. Both classes enable dynamic scanning and loading of classes at runtime.
- The project has a plugin that fails at compile time. For example, the plugin `snakeyaml-codegen-maven-plugin` in the project `helidon` adds code to the project's compiled sources automatically [35], and fails when building with the specialized dependency `smallrye-open-api-core` because the specialization process changes the expected dependency bytecode.
- The project has a plugin that checks the integrity of the specialized dependency. For example, the dependency `commons-io` in project `jcabi-github` uses the `maven-enforcer-plugin` to check for certain constraints, including checksums, on the dependency bytecode.
- The specialized dependency is not found in the local repository. For example, the specialized dependency `snakeyaml` in project `java-faker` is not deployed correctly due to a known issue in this dependency when

using the `android` MAVEN tag classifier [36].

We now discuss the number of specialized dependencies for which the build reports test failures (column TEST FAIL. in Table 2). For 9 projects, one specialized dependency has at least one test failure. DEPTRIM preserves the original tested behavior (*i.e.*, all the tests pass) of 387 (97.7 %) specialized, non-bloated compile-scope dependencies. This high rate of test success is a fundamental result to ensure that the specialized version of the dependency tree preserves the tested behavior of the project.

In total, we execute 27,844 unique tests across all projects (per Table 1). Of these, 130 do not pass. DEPTRIM produces specialized dependency trees that break a few test cases. These cases reveal the challenges of dependency specialization concerning static analysis. For example, DEPTRIM can miss some used classes, resulting in the removal of bytecode that is necessary at runtime. This is a general constraint for static analysis tools when processing Java applications that rely on dynamic features to load and execute code at runtime. As a result of the absence of bytecode from a specialized dependency, 9 projects report test failures, *e.g.*, an unreachable class loaded at runtime causing a failing test that stops the execution of the build.

Table 5 shows the number of unique test failures (column #TEST FAIL.) in the 9 projects that have at least one `PST` with test failures, as well as the specialized dependency that causes the failure (column DEPENDENCY). For example, the project `Chronicle-Map` has 3 tests that fail when specializing the dependency `chronicle-wire`, from a total of 1,231 executed tests, which represents 0.2 % of the total. The project with the largest number of test failures is `poi-tl`, with 107 (85.6 %) tests failures when specializing its dependency `commons-io`. Overall, the number of test failures accounts for 1.1 % of the total tests executed in the 9 projects, and only 0.5 % across the 30 projects.

We further investigate the causes of the failures. To do so, we manually analyze the logs of the tests, as reported by the `maven-surefire-plugin`. We find the following 3 causes:

- The tests load dependency classes dynamically. For example, `poi-tl` relies on the method `byte[]` in class `IOUtils` of `commons-io` to check the size of a file. This method is loaded via reflection through an external configuration file and causes the failure of 107 tests.
- Some tests rely on Java serialization to manipulate objects at runtime, and the input stream is not closed properly because DEPTRIM removes a class responsible for closing the input stream. For example, the project `Chronicle-Map` uses the dependency `chronicle-wire` for serialization, and 3 tests fail due to a `ClosedIORuntimeException`.
- The project has tests that rely on dependencies that use Java Native Interfaces (JNI) to execute machine code at runtime. For example, the test `TestWsdlValidation` in project `woodstox` relies on dependency `msv-core` which uses JNI to validate XML schemas. DEPTRIM's static analysis is limited to Java bytecode, and therefore native code executed in third-party dependencies is not considered as used when building the call graph.

Our results reveal the challenges of dependency specialization based on static analysis (see Section 3.2.1) for real-world Java projects. Handling these cases to achieve 100 % correctness requires specific domain knowledge of the project, and of the reachable code in the dependencies that exercise some form of dynamic Java features. To facilitate this task, we provide a dedicated parameter `ignoreDependencies` in DEPTRIM so that developers can declare a list of dependency coordinates to be ignored by DEPTRIM during the call graph analysis. Nevertheless, we recommend always checking that the build passes to avoid semantic errors when performing bytecode removal transformations.

Answer to RQ4: Of the 396 dependencies that are targets for specialization, 32 are not specialized because they are totally used, 12 (3.3 %) dependencies cause a compilation failure when specialized, and 9 (2.5 %) lead to a failure at runtime. For the latter, the test failures represent only 0.5 % of the total number of tests executed. This behavioral assessment of DEPTRIM demonstrates that the specialized dependency trees preserve a large majority of syntactic and semantic correctness for the 30 projects.

6 DISCUSSION

In this section, we discuss the state-of-the-art and the current challenges of code specialization in Java, as well as the implications of specialization for software integrity. We also discuss the threats to the validity of our results.

6.1 Specialization in the Modern Java Ecosystem

The Java community is currently making substantial efforts to reduce the amount of bloated code deployed in production. The GraalVM native image compiler [37] is perceived by many as an important step in this direction. GraalVM relies on static analysis to build a native executable image that only includes the elements reachable from an application entry point and its third-party dependencies [38]. To do so, GraalVM operates under a closed-world assumption [39]: all the bytecode that can be called at runtime, must be known at build time, *i.e.*, when the native-image tool in GraalVM is building the standalone executable [40]. Thanks to this condition, GraalVM is able to perform a set of aggressive optimizations such as the elimination of unused code from third-party dependencies. The self-contained native executable image includes only code that is actually necessary to build and execute a Java project. This reduces the size of container images, making Java applications easy to ship and deploy directly in a containerized environment, as microservices for example.

A big challenge is that many legacy applications are not designed according to the closed-world assumption. In this case, the reachability of some bytecode elements (such as classes, methods, or fields) may not be identified due to the Java dynamic features, *e.g.*, reflection, resource access, dynamic proxies, and serialization [41], [42], [34], [43]. For example, the popular dependency `netty`, an asynchronous event-driven framework, heavily relies on dynamic Java features to perform blocking and non-blocking sockets between servers and clients. The closed-world constraint of GraalVM imposes strict limits on the natural dynamism of Java upon which libraries and frameworks like `netty` depend.

There is a risk of violating the close world assumption if at least one of the dependency in a project relies on some dynamic Java feature.

To bridge the gap between the requirements of GraalVM and the current state of Java systems, the community is creating new versions of libraries that adhere to the closed-world assumption. In the long run, Java developers will have the option to embrace the full closed-world constraint in order to produce fully-static images. Between now and then, however, the community works on developing and delivering incremental improvements which developers can use sooner rather than later. DepTrim contributes to this effort, offering a specialization solution for projects that have dependencies potentially conflicting with the closed-world assumption. With the creation of a partially specialized tree (PST), DepTrim effectively achieves dependency specialization without jeopardizing the success of the build, making it a practical option. Note that the successful build of a project does not guarantee that its behaviour is unchanged. Indeed, test suites can never prove the absence of bugs, and must generally concentrate on specific issues, since it is impossible to test everything. More research is needed to precisely ascertain the extent to which the usage dynamic of features affects dependency specialization.

6.2 Specialized Projects in Production

In this work, we assess the validity of the specialized projects with respect to their tests suite. In practice, developers who wish to deploy their specialized project in production, might consider one more validation step to assess how specialization may impact their users. This additional step depends on how the specialized project is used: either declared as dependency within client projects, or deployed as an application with which end-users interact.

If the specialized project is mainly used as a library, an additional validation step consists in assessing the specialized JAR with respect to representative client projects. By running the test suite of these clients, the developers can verify that the specialization does not result in unexpected behaviors. This mechanism, termed *reverse dependency compatibility testing*, has been applied previously in the literature to identify breaking updates in libraries [44]. This requires curating a list of relevant client projects, which usage of the library is meaningful and which test suites actually exercise the library. If the specialized project essentially faces end users, through a graphical or command-line interface, an additional validation step consists in running a production-like workload on the specialized application.

We have performed a proof of concept of this augmented validation for specialized projects to be used in production. We focused on user facing projects. Three projects in our dataset have a graphical or command-line interface and can be executed with a workload: `graphhopper`, `pdfbox`, and `Recaf`. For each of these projects, we build the original JAR and run it with a workload that is representative of a typical production operation. Next, we build a specialized version of the project with DepTrim, and run the new JAR with the same workload. The analysis of both executions let's us determine if the observable behavior of the specialized project is as intended.

The first project, `graphhopper`,[2] is a routing application based on `OpenStreetMap`. DepTrim removes 1,666 third-party classes from 12 dependencies in `graphhopper` to output a PST. The workload for `graphhopper` consists of running its JAR and fetching the route between four locations in Sweden from its web page. The second project, `pdfbox`,[3] is developed by the Apache Software Foundation. It offers command-line tools for performing common operations on PDF documents. DepTrim produces a TST for `pdfbox` by specializing 6 dependencies within it, which results in the removal of 5,070 classes. As the workload for `pdfbox`, we use 10 of its command-line utilities, on 5 PDF documents sourced from [45]. These operations include text and image extraction, encryption, decryption, and merging and splitting PDF documents. `Recaf`[4] is a code editor that allows developers to manipulate bytecode through a graphical user interface. DepTrim produces a TST for `Recaf`, by removing 2,952 classes from 41 of its dependencies. As the production workload for `Recaf`, we import a compiled `.class` file into its editor, which decompiles it, and renders its source. We then modify the source by adding a statement, and export this new version as a `.java` file.

For all three specialized projects, we do not observe any deviation between the behavior of the original and the specialized version. The route returned by specialized `graphhopper` is identical to the one returned by the original. The `pdfbox` operations also result in the same output files. We successfully modify a decompiled `.class` file with specialized `Recaf` and export it, as with the original `Recaf` JAR file. Additionally, we do not get any unexpected log outputs within `graphhopper` and `pdfbox`. However, `Recaf` outputs a log message during startup about a missing class within the specialized dependency `logback-core`. This non-critical exception can be remediated by adding `logback-core` to the specialization blacklist of DepTrim.

The executions of the three projects under realistic workloads confirm that their high-level features are not impacted by specialization. Developers can leverage dependency specialization to deliver focused versions of their application to end-users, while keeping its behavior intact. An interesting direction for future work is to conduct this evaluation for a larger set of specialized projects.

When we run DepTrim across the three case studies, the execution times are in the order of minutes, taking 18, 10, and 33 minutes for `graphhopper`, `pdfbox`, and `Recaf` respectively. Execution time variations are due to several factors. These factors include the test execution time, the number of dependencies a project has, and the extent to which these dependencies are utilized within the project. Interestingly, within the DepTrim execution pipeline, the most time-consuming step is not the static analysis. Instead, it is the dynamic analysis phase where tests are run with each specialized dependency tree version to validate their usability. Given its relatively expedient execution time and the potential benefits of optimizing dependency trees, we advocate for the integration of DepTrim into the Continuous Integration (CI) pipeline of projects, especially during new

2. https://www.graphhopper.com/
3. https://pdfbox.apache.org/
4. https://www.coley.software/Recaf/

releases. This is particularly beneficial for projects aiming to deploy smaller JAR files for their dependencies, such as those pushing updates to Maven Central, where minimized JAR files could benefit their end users.

6.3 Specialization and Software Integrity

The integrity of software supply chains is a timely research topic [46], [47], [48]. Ensuring the integrity of dependencies involves checking that their code has not been tampered with between the moment they are fetched from a repository and the moment they are packaged in the project. Checksums, such as the SHA family of cryptographic functions, are commonly used to verify the integrity of software dependencies. These checksums are then integrated as part of the project's software bill of materials (SBOM) that lists all the components that compose it, including open-source libraries, frameworks, and tools [49]. A comprehensive, well-maintained SBOM can help ensure software integrity by enabling organizations to identify and track potential security vulnerabilities in their software components and take appropriate action to address them, while also complying with regulations and standards.

The specialization of third-party dependencies modifies the bytecode of the target dependencies, which can break the integrity-checking process. In other words, rehashing a dependency with a different bytecode will produce a different hash value, breaking the integrity check. This is because the checksum of the original bytecode, which was used to verify the integrity of the dependency, will no longer match the checksum of the changed bytecode. For example, Listing 1 shows a JSON file reporting the checksum of the original dependency `commons-io` in one of our study subjects, `jcabi-github`, when using the SHA-256 hashing algorithm. DepTrim specializes `commons-io` by removing unused classes, which constitutes a change in its bytecode, and hence in its checksum, as presented in Listing 2. Therefore, the checksum of the changed bytecode after specialization no longer matches the expected checksum, and the integrity checks fail, as discussed in Section 4.5.

A way to ensure the integrity of specialized dependencies is by deploying them to external repositories at build time. For example, in the previous example, the project `jcabi-github` could deploy the specialized variant of `commons-io` to Maven Central with a custom Maven `groupId`, while updating the checksum in its SBOM accordingly. This way, it could check the integrity of this dependency against the SHA of the specialized variant. This approach provides the benefits of specialization while preserving software integrity. As far as we know, there is currently no tool that implements this technique. Preserving integrity in the light of specialization is a challenge for hardening the software supply chain.

```
{
  "groupId": "commons-io",
  "artifactId": "commons-io",
  "version": "2.11.0",
  "checksumAlgorithm": "SHA-256",
  "checksum": "961b2f6d87dbacc5d54abf45ab7a6e2495f89b755989
       ↪ 62d8c723cea9bc210908"
}
```

Listing 1: SHA checksum of the original dependency `commons-io` in the project `jcabi-github`

```
{
  "groupId": "se.kth.castor.deptrim.spl",
  "artifactId": "commons-io",
  "version": "2.11.0",
  "checksumAlgorithm": "SHA-256",
  "checksum": "c84eaef6b629729c71a70a2513584e7ccacf70cb4df1
       ↪ 3e38b731bb6193c60e73"
}
```

Listing 2: SHA checksum of the specialized dependency `commons-io` in the project `jcabi-github`

6.4 Threats to Validity

Internal validity. The first internal threat relates to the usage of static analysis to determine which parts of the dependency bytecode are reachable from the project. We mitigate this threat, by relying on DepClean, the state-of-the-art tool for debloating Java dependencies [24]. Another threat lies in the thoroughness of the test suite, which may not capture all the dependency API behaviors that can be exercised by the project. This means that there is a risk that some necessary classes would be removed, yet the build would be successful because of insufficient testing. For example, the projects `immutables`, `scribejava`, and `tablesaw` successfully build with a TST but have less than 100 tests each. To mitigate this threat, we curate a set of study subjects that are mature and contain tests (see Table 1). DepTrim is a Maven plugin that modifies the *pom.xml* on-the-fly during the build process. It might introduce conflicts between plugins, causing the build to fail. For example, `maven-enforcer-plugin` or `license-maven-plugin` check the *pom.xml* to ensure that it meets specific requirements and follows the best practices. However, since our approach only modifies the code within the entry `dependencies` in the *pom.xml*, the failures due to misconfigurations are minimized.

External validity. Our results are representative of the Java ecosystem, and our findings are valid for software projects with these particular characteristics. Moreover, our bytecode removal results are influenced by the number of dependencies of these projects. To address this, we found our evaluation on 30 real-world, well-known projects, derived from sound data sources, as described in Section 4.1. Furthermore, the selected projects cover a variety of application domains (*e.g.*, dependency injection, database handling, machine learning, encryption, IO utilities, faking, metaprogramming, networking, etc). To the best of our knowledge, this is the largest set of study subjects used in software specialization experiments.

Construct validity. The threats to construct validity relate to the accuracy and soundness of the results. Our results may not be reproducible if the projects are compiled with a different Java version or have flaky tests. To mitigate this threat, we choose the latest Java version and build the original projects two times in order to avoid including projects with flaky tests. Furthermore, for all RQs, we include logs and automated analysis scripts in our replication package for reproducibility as described in Section 4.6.

7 Related Work

In this section, we position the contribution of our dependency specialization technique with respect to previous work that aims at reducing the size of applications composed of multiple third-party dependencies.

Several previous works focus on reducing the size of Java applications. While all techniques perform code analysis based on the construction of a call graph, they vary in the way they look for code that can be removed: dead-code removal, inlining, and class hierarchy removal [50]; identification and removal of unused optional concerns with respect to a specific installation context [51], unbundling user-facing application features [52] or tailoring the Java standard library [53], [54]. In contrast to these efforts that aim at reducing the size of a packaged application, DepTrim targets reduction while keeping the modular structure of the project and its third-party dependencies. Our technique focuses on reducing each dependency while keeping an explicit dependency tree in the form of a specialized *pom.xml* file as well as maintaining specialized dependencies as distinct, deployable JAR files.

Bruce *et al.* [55] propose JShrink, augmenting static reachability analysis with dynamic reachability analysis. They rely on test cases to find dynamic features, including methods and fields, invoked at runtime, adding them back to amend the imprecision of static call graphs. DepTrim differs from JShrink, as it does not aim to refine reachability analysis to create smaller JAR files of the target project. Instead, DepTrim focuses on specializing the dependency tree of a Java project by removing unused code in third-party dependencies independently, such that each dependency can be deployed to external repositories.

In our previous work, we proposed DepClean, a tool that identifies and removes unused dependencies in the dependency tree [56], [25]. DepClean constructs a call graph of the bytecode class members by capturing annotations, fields, and methods, and accounts for a limited number of dynamic features such as class literals. DepClean produces a variant of the dependency tree without bloated dependencies. DepTrim pushes forward the field of dependency debloating through the removal of unused bytecode from individual dependencies, thereby yielding smaller packaged artifacts.

Table 6 shows a comparison between DepTrim and state-of-the-art debloating tools for Java applications, per the recent study of Ponta *et al.* [18]. We have included a recent tool, JDBL, which produces debloated JAR files based on the usage analysis obtained from code coverage tools [57]. We compare the tools regarding three distinct specialization outcomes: the specialized project JAR, specialized dependency JAR files, and specialized *pom.xml* files. A key observation is that state-of-the-art tools primarily focus on generating a debloated JAR file for the target Java project. More specifically, the Maven Shade Plugin, Proguard, and JDBL aim to build an uber-JAR, which encapsulates all the utilized code from the project's dependencies, resulting in a self-contained JAR file. Only DepClean produces a modified version of the *pom.xml* file. The key novelty of DepTrim is that it is the first tool that specializes individual dependency JAR files and produces specialized *pom.xml* files. The generation and packaging of specialized individual dependency JAR files allows developers to benefit from specialization while maintaining a modular architecture, and they can eventually include the specialized dependencies as part of their project's software bill of materials [49].

Table 6: Comparison between specialization outputs of existing Java debloating tools and DepTrim.

Tool	Specialization Output		
	Project JAR	Dependency JAR	POM file
Maven Shade	✓		
ProGuard	✓		
JDBL	✓		
DepClean	✓		✓
DepTrim	✓	✓	✓

Closely related to DepTrim is the work on code specialization. Mishra and Polychronakis propose Shredder [20], a defense-in-depth exploit mitigation tool that protects closed-source applications against code reuse attacks. They also build Saffire [58] which creates specialized and hardened replicas of critical functions with restricted interfaces to prevent code reuse attacks. These tools target C++ API implementations. They eliminate arguments with static values and restrict the acceptable values of arguments. A key feature of these techniques is to replace the code of API members by a stub function so that, at runtime, only specialized versions of critical API functions are exposed, while any invocation that violates the enforced policy is blocked. Focusing on JavaScript applications, Turcotte *et al.* [59] propose Stubbifier, which replaces unreachable code, identified through static and dynamic call graphs. DepTrim does not remove unused code from the project but rather replaces dependencies that are partially used by the project with smaller and specialized versions.

Previous specialization techniques mitigate the risk of removing code that might be needed for a specific execution, by replacing this code by small stub functions. With DepTrim we address the challenges of dynamic language features with another strategy. We specialize each dependency and then assess whether the completely specialized dependency tree still passes the build. If it does not, we search for a partially specialized tree that does not include the dependencies that rely on the dynamic features of Java. To the best of our knowledge, prior research on software specialization has not addressed the customization of third-party dependencies or the provision of build configuration files to enable the construction of specialized dependency trees. This represents a novel contribution of our work, differentiating it from previous studies in this area.

As part of our experiments with DepTrim, we contribute novel observations to the body of knowledge about library and API usage. Recent work in this area includes the following studies. Huang *et al.* [60] study the usage intensity from Java projects to libraries. They find that the number of libraries adopted by a project is correlated to the project size. However, their study does not provide a more fine-grained analysis of the used components. Hejderup *et al.* [61] investigate the extent to which Rust projects use the third-party packages in their dependency tree. They propose

PRÄZI, a call-based dependency network for CRATES.IO that operates at the function level.

Some studies examine the benefits of debloating from a security standpoint. For instance, Azad *et al.* [62] report that debloating significantly reduces the number of vulnerabilities in web applications, while also making it more difficult for attackers to exploit the remaining ones. Agadakos *et al.* [63] propose NIBBLER to erase unused functions within the binaries of shared libraries at the binary level. This enhances existing software defenses, such as continuous code re-randomization and control-flow integrity, without incurring additional run-time overhead. Ye *et al.* [64] implement a tool that uses NLP and function call graphs to identify and isolate vulnerabilities in NPM packages, effectively reducing software bloat and preventing known vulnerability exploitation in JavaScript applications. Although DEPTRIM's primary function is to specialize dependency trees and enhance their reusability, it is important to note that the removal of third-party code can lead to a reduction in the potential attack surface.

8 CONCLUSION

In this paper, we propose DEPTRIM, a fully automated technique to specialize third-party dependencies of a Java project. DEPTRIM systematically identifies and removes unused classes within each reachable dependency, repackages the used classes into a specialized dependency, and replaces the original dependency tree of a project with a specialized version. DEPTRIM builds a minimal project binary, which only contains code that is necessary for the project.

Our evaluation with 30 MAVEN Java projects demonstrates the capabilities of DEPTRIM to produce minimal versions of the dependencies in these projects while keeping the original build successful. In particular, DEPTRIM builds totally specialized trees for 14 projects and builds the other 16 with the largest number of specialized dependencies such that the project still builds. The ratio of dependency classes to project classes decreases from $8.7\times$ in the original projects to $5.0\times$ in the specialized projects. This represents a reduction of $35.7\,\%$ of the bytecode size in third-party dependencies. Dependency specialization effectively reduces the share of third-party code in Java projects.

As future work we will investigate dependency specialization to increase diversity in software supply chains [49]. DEPTRIM currently generates one specialized dependency tree for each project. However, there exists a multitude of possibilities within the realm of partially specialized trees, which we have yet to explore. We will venture into the forest of dependency trees to let diversity blossom in Java applications.

ACKNOWLEDGEMENTS

This work has been partially supported by the Wallenberg Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, as well as by the Chains project funded by the Swedish Foundation for Strategic Research (SSF).

REFERENCES

[1] C. W. Krueger, "Software reuse," *ACM Computing Surveys (CSUR)*, vol. 24, no. 2, pp. 131–183, 1992.

[2] R. Cox, "Surviving software dependencies," *Communications of the ACM*, vol. 62, no. 9, pp. 36–43, 2019.

[3] T. Gustavsson, "Managing the open source dependency," *IEEE Computer*, vol. 53, no. 2, pp. 83–87, 2020.

[4] N. Harutyunyan, "Managing your open source supply chain-why and how?," *IEEE Computer*, vol. 53, no. 6, pp. 77–81, 2020.

[5] P. Ombredanne, "Free and open source software license compliance: tools for software composition analysis," *IEEE Computer*, vol. 53, no. 10, pp. 105–109, 2020.

[6] 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*, vol. 172, p. 110653, 2021.

[7] A. Decan, T. Mens, and E. Constantinou, "On the evolution of technical lag in the npm package dependency network," in *Proc. of ICSME*, pp. 404–414, 2018.

[8] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transacions on Software Engineering*, vol. 48, no. 10, pp. 3790–3807, 2022.

[9] K. Heo, W. Lee, P. Pashakhanloo, and M. Naik, "Effective program debloating via reinforcement learning," in *Proc. of ACM CCS*, pp. 380–394, 2018.

[10] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?," *IEEE Transacions on Software Engineering*, vol. 48, no. 7, pp. 2295–2316, 2022.

[11] A. Dann, H. Plate, B. Hermann, S. E. Ponta, and E. Bodden, "Identifying challenges for OSS vulnerability scanners - A study & test suite," *IEEE Transacions on Software Engineering*, vol. 48, no. 9, pp. 3613–3625, 2022.

[12] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transacions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, 2022.

[13] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," in *Proc. of ESEM*, pp. 1–11, 2021.

[14] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests," in *Proc. of MSR*, pp. 254–265, 2021.

[15] R. Cox, R. Griesemer, R. Pike, I. L. Taylor, and K. Thompson, "The go programming language and environment," *Communications of the ACM*, vol. 65, no. 5, pp. 70–78, 2022.

[16] Z. Newman, J. S. Meyers, and S. Torres-Arias, "Sigstore: software signing for everybody," in *Proc. of ACM CCS*, pp. 2353–2367, 2022.

[17] P. Pashakhanloo, A. Machiry, H. Choi, A. Canino, K. Heo, I. Lee, and M. Naik, "Pacjam: Securing dependencies continuously via package-oriented debloating," in *Proc. of ACM CCS*, pp. 903–916, 2022.

[18] S. E. Ponta, W. Fischer, H. Plate, and A. Sabetta, "The used, the bloated, and the vulnerable: Reducing the attack surface of an industrial application," in *Proc. of ICSME*, pp. 555–558, IEEE, 2021.

[19] J. Latendresse, S. Mujahid, D. E. Costa, and E. Shihab, "Not all dependencies are equal: An empirical study on production dependencies in NPM," in *Proc. of ASE*, pp. 1 – 12, 2022.

[20] S. Mishra and M. Polychronakis, "Shredder: Breaking exploits through api specialization," in *Proc. of ACSAC*, pp. 1–16, 2018.

[21] B. Xia, T. Bi, Z. Xing, Q. Lu, and L. Zhu, "An Empirical Study on Software Bill of Materials: Where We Stand and the Road Ahead," in *Proceedings of ICSE*, 2023.

[22] A. S. Foundation, "Apache Maven," January 2023. Available at https://maven.apache.org.

[23] Apache Maven, "Introduction to the dependency mechanism," January 2023. Available at https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html.

[24] C. Soto-Valero, N. Harrand, M. Monperrus, and B. Baudry, "A comprehensive study of bloated dependencies in the maven ecosystem," *Empirical Software Engineering*, vol. 26, no. 3, pp. 1–44, 2021.

[25] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated java dependencies," in *Proc. of ESEC/FSE*, pp. 1021–1031, 2021.

[26] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "Do the dependency conflicts in my project matter?," in *Proc. of ESEC/FSE*, pp. 319–330, 2018.

[27] L. Sui, J. Dietrich, A. Tahir, and G. Fourtounis, "On the recall of static call graph construction in practice," in *Proc. of ICSE*, pp. 1049–1060, 2020.

[28] "Depclean." https://github.com/castor-software/depclean/tree/master/depclean-core. Accessed: 2023-05-18.

[29] T. Durieux, C. Soto-Valero, and B. Baudry, "Duets: A dataset of reproducible pairs of java library-clients," in *Proc. of MSR*, pp. 545–549, 2021.

[30] H. Borges and M. T. Valente, "What's in a github star? understanding repository starring practices in a social coding platform," *Journal of Systems and Software*, vol. 146, pp. 112–129, 2018.

[31] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S.-C. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?," *IEEE Transactions on Software Engineering*, vol. 48, no. 7, pp. 2295–2316, 2021.

[32] Google, "Guava listenablefuture," January 2023. Available at https://mvnrepository.com/artifact/com.google.guava/listenablefuture/9999.0-empty-to-avoid-conflict-with-guava.

[33] G. Balatsouras and Y. Smaragdakis, "Class hierarchy complementation: soundly completing a partial type graph," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 515–532, 2013.

[34] Y. Li, T. Tan, and J. Xue, "Understanding and analyzing java reflection," *ACM Transactions on Software Engineering and Methodology*, vol. 28, no. 2, pp. 1–50, 2019.

[35] Helidon, "Helidon snakeyaml helper maven plugin," January 2023. Available at https://github.com/helidon-io/helidon-build-tools/tree/master/maven-plugins/snakeyaml-codegen-maven-plugin.

[36] GitHub, "Issue 327," October 2018. Available at https://github.com/DiUS/java-faker/issues/327.

[37] O. C. and/or its affiliates, "Project leyden: Beginnings," May 2022. Available at https://openjdk.org/projects/leyden/notes/01-beginnings.

[38] GraalVM, "The graalvm native image," January 2023. Available at https://www.graalvm.org/native-image.

[39] C. Wimmer, C. Stancu, P. Hofer, V. Jovanovic, P. Wögerer, P. B. Kessler, O. Pliss, and T. Würthinger, "Initialize once, start fast: Application initialization at build time," *Proc. of OOPSLA*, 2019.

[40] C. Wimmer, "Graalvm native image: large-scale static analysis for java (keynote)," in *Proc. of Workshop on Virtual Machines and Intermediate Languages*, pp. 3–3, 2021.

[41] 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 *Proc. of APLAS*, pp. 69–88, 2018.

[42] D. Landman, A. Serebrenik, and J. J. Vinju, "Challenges for static analysis of java reflection-literature review and empirical study," in *Proc. of ICSE*, pp. 507–518, 2017.

[43] M. Reif, F. Kübler, M. Eichberg, D. Helm, and M. Mezini, "Judge: Identifying, understanding, and evaluating sources of unsoundness in call graphs," in *Proc. of ISSTA*, pp. 251–261, 2019.

[44] L. Ochoa, T. Degueule, and J.-R. Falleri, "Breakbot: Analyzing the impact of breaking changes to assist library evolution," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*, pp. 26–30, 2022.

[45] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *digital investigation*, vol. 6, pp. S2–S11, 2009.

[46] C. Lamb and S. Zacchiroli, "Reproducible builds: Increasing the integrity of software supply chains," *IEEE Software*, vol. 39, no. 2, pp. 62–70, 2021.

[47] M. Ahmadvand, A. Pretschner, and F. Kelbert, "A taxonomy of software integrity protection techniques," in *Advances in Computers*, vol. 112, pp. 413–486, Elsevier, 2019.

[48] W. Enck and L. Williams, "Top five challenges in software supply chain security: Observations from 30 industry and government organizations," *IEEE Security & Privacy*, vol. 20, no. 2, pp. 96–100, 2022.

[49] M. Balliu, B. Baudry, S. Bobadilla, M. Ekstedt, M. Monperrus, J. Ron, A. Sharma, G. Skoglund, C. Soto-Valero, and M. Wittlinger, "Challenges of producing software bill of materials for java," *IEEE Security & Privacy*, pp. 2–13, 2023.

[50] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for java," in *Proc. of OOPSLA*, pp. 292–305, 1999.

[51] S. Bhattacharya, K. Gopinath, and M. G. Nanda, "Combining concern input with program analysis for bloat detection," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 745–764, 2013.

[52] J. B. F. Filho, M. Acher, and O. Barais, "Software unbundling: Challenges and perspectives," *LNCS Trans. Modul. Compos.*, vol. 1, pp. 224–237, 2016.

[53] Y. Jiang, D. Wu, and P. Liu, "Jred: Program customization and bloatware mitigation based on static analysis," in *Proc. of COMPSAC)*, vol. 1, pp. 12–21, IEEE, 2016.

[54] D. Rayside and K. Kontogiannis, "Extracting java library subsets for deployment on embedded systems," *Science of Computer Programming*, vol. 45, no. 2-3, pp. 245–270, 2002.

[55] B. R. Bruce, T. Zhang, J. Arora, G. H. Xu, and M. Kim, "Jshrink: In-depth investigation into debloating modern java applications," in *Proc. of ESEC/FSE*, pp. 135–146, 2020.

[56] C. Soto-Valero, A. Benelallam, N. Harrand, O. Barais, and B. Baudry, "The Emergence of Software Diversity in Maven Central," in *Proc. of MSR*, pp. 1–10, 2019.

[57] C. Soto-Valero, T. Durieux, N. Harrand, and B. Baudry, "Coverage-based debloating for java bytecode," *ACM Transactions on Software Engineering and Methodology*, vol. 32, no. 2, pp. 1–34, 2023.

[58] S. Mishra and M. Polychronakis, "Saffire: Context-sensitive function specialization against code reuse attacks," in *Proc. of (EuroS&P)*, pp. 17–33, IEEE, 2020.

[59] A. Turcotte, E. Arteca, A. Mishra, S. Alimadadi, and F. Tip, "Stubbifier: debloating dynamic server-side javascript applications," *Empirical Software Engineering*, vol. 27, no. 7, pp. 1–36, 2022.

[60] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu, "Characterizing usages, updates and risks of third-party libraries in java projects," *Empirical Software Engineering*, vol. 27, no. 4, p. 90, 2022.

[61] J. Hejderup, M. Beller, K. Triantafyllou, and G. Gousios, "Präzi: from package-based to call-based dependency networks," *Empirical Software Engineering*, vol. 27, no. 5, pp. 1–42, 2022.

[62] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: quantifying the security benefits of debloating web applications," in *Proc. of USENIX Security*, pp. 1697–1714, 2019.

[63] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: debloating binary shared libraries," in *Proc. of ACSAC*, pp. 70–83, 2019.

[64] R. Ye, L. Liu, S. Hu, F. Zhu, J. Yang, and F. Wang, "Jslim: Reducing the known vulnerabilities of javascript application by debloating," in *International Symposium on Emerging Information Security and Applications*, pp. 128–143, Springer, 2021.