



Efficiently Trimming the Fat: Streamlining Software Dependencies with Java Reflection and Dependency Analysis

Xiaohu Song
Northeastern University
Shenyang, China
2010511@stu.neu.edu.cn

Ying Wang*
Northeastern University
Shenyang, China
wangying@swc.neu.edu.cn

Xiao Cheng
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China
chengxiao5@huawei.com

Guangtai Liang
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China
liangguangtai@huawei.com

Qianxiang Wang
Huawei Cloud Computing
Technologies Co., Ltd.
Beijing, China
wangqianxiang@huawei.com

Zhiliang Zhu
National Frontiers Science Center for Industrial
Intelligence and Systems Optimization, and Key
Laboratory of Data Analytics and Optimization
for Smart Industry, Northeastern University
Shenyang, China
ZHUZhiLiang_NEU@163.com

ABSTRACT

Numerous third-party libraries introduced into client projects are not actually required, resulting in modern software being gradually bloated. Software developers may spend much unnecessary effort to manage the bloated dependencies: keeping the library versions up-to-date, making sure that heterogeneous licenses are compatible, and resolving dependency conflict or vulnerability issues.

However, the prior debloating techniques can easily produce false alarms of bloated dependencies since they are less effective in analyzing Java reflections. Besides, the solutions given by the existing approaches for removing bloated dependencies may induce new issues that are not conducive to dependency management. To address the above limitations, in this paper, we developed a technique, SLIMMING, to remove bloated dependencies from software projects reliably. SLIMMING statically analyzes the Java reflections that are commonly leveraged by popular frameworks (e.g., *Spring Boot*) and resolves the reflective targets via parsing configuration files (*.xml, *.yaml and *.properties). By modeling string manipulations, SLIMMING fully resolves the string arguments of our concerned reflection APIs to identify all the required dependencies. More importantly, it helps developers analyze the debloating solutions by weighing the benefits against the costs of dependency management. Our evaluation results show that the static reflection analysis capability of SLIMMING outperforms all the other existing techniques with 97.0% of *Precision* and 98.8% of *Recall*. Compared with the prior debloating techniques, SLIMMING can reliably remove the bloated dependencies with a 100% test passing ratio and improve the rationality of debloating solutions. In our large-scale study in

the Maven ecosystem, SLIMMING reported 484 bloated dependencies to 66 open-source projects. 38 reports (57.6%) have been confirmed by developers.

CCS CONCEPTS

• Software and its engineering → Software libraries and repositories.

KEYWORDS

Bloated Dependencies, Java Reflection, Dependency Management

ACM Reference Format:

Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. 2024. Efficiently Trimming the Fat: Streamlining Software Dependencies with Java Reflection and Dependency Analysis. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24)*, April 14–20, 2024, Lisbon, Portugal. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3597503.3639123>

1 INTRODUCTION

“Redundant dependencies in software are like weeds in a garden.”

- Robert C. Martin

Dependency management tools (e.g., Maven for Java and Cargo for Rust) boost software reuse by automatically fetching all the direct and transitive third-party dependencies required to compile, test and deploy a client project. In practice, however, numerous introduced third-party libraries are not actually used by client code, resulting in gradually bloating software projects. A recent study [40] on 435 open-source projects showed that the number of transitive dependencies in 2011 was 1,695, and by the end of 2020, this number grew up to 286,228 (increase > 250×). Bloated software negatively affects code performance on resource-constrained devices with limited memory. Additionally, it poses a direct security threat since unused libraries increase security attack surfaces and introduces an extra burden for dependency management. Software developers may spend much effort to manage unnecessary dependencies:

*Ying Wang is the corresponding author.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICSE 2024, April 2024, Lisbon, Portugal

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 979-8-4007-0217-4/24/04

<https://doi.org/10.1145/3597503.3639123>

keeping the library versions up-to-date, making sure that heterogeneous licenses are compatible, resolving dependency conflicts or vulnerability issues.

To promote dependency management, developers urgently need an effective technique to help them remove the bloated third-party libraries from dependency configurations. Many existing debloating techniques [23, 24, 27, 32, 34–38, 41, 44] are proposed for reducing the size of software at fine granularity (i.e., method/statement level), to improve the efficiency of installations and executions on resource-constrained hardware. Soto-Valero [40] et al. develop a coarse-grained debloating technique, DEPCLEAN, which constructs the complete dependency tree of a project and determines the presence of bloated libraries by statically analyzing the bytecode. However, the prior approaches have the following limitations:

- *Less effective in analyzing Java reflections.* Dynamic language features are widely adopted in developing modern software projects, which are also leveraged by projects to facilitate flexible integration with the code from third-party libraries. To avoid false alarms of bloated dependencies, a debloating technique should be empowered by effective reflection analysis. However, reflection analysis is a long-standing hard open problem. Previous debloating techniques consider Java reflection as a separate assumption, and they either handle it partially based on dynamic analysis techniques [41] or cannot resolve it well based on the existing static analysis tools [23, 27, 32, 42]. The dynamic analysis techniques heavily depend on the test coverage rate of the given projects, which is not scalable for all the subjects. The static analysis techniques cannot resolve the reflection code well since dynamic behaviors of reflective calls are mainly specified by their string arguments, with some string values being intra-procedurally and inter-procedurally manipulated, read from configuration files, or even retrieved from the Internet. The challenges of static reflection analyses are illustrated in Section 2.1.
- *Remove bloated libraries may induce new issues that are not conducive to dependency management.* In many cases, although the client project does not actually use a bloated dependency, the libraries transitively introduced by the bloated dependency can be invoked by client code via certain direct dependencies. After removing the bloated library from a configuration file, the referenced transitive dependencies should be declared as the client project's direct dependencies to ensure the program can run correctly. At the cost of the above changes, developers would spend much effort to keep their versions up-to-date. However, their updated version can easily induce incompatibility issues with the libraries depending on them. There is no existing technique to help developers analyze the solutions for removing bloated dependencies by weighing the benefits against the costs of dependency management. The challenges of removing bloated dependencies are illustrated in Section 2.2.

To address the above limitations, in this paper, we developed a technique, SLIMMING, to help developers reliably remove bloated dependencies from software projects. Our work focuses on the Maven ecosystem, the most popular package manager and automatic build system for Java programming language. SLIMMING is empowered by effective static reflection analyses, which can precisely resolve: (1) the reflections that are synthesized by dependency injections,

annotations, and configurations in popular Java frameworks (e.g., *Java EE Servlets*, *Java Beans*, *Apache Struts*, *Spring*, *Spring Boot* and *Tomcat*); (2) non-constant arguments of reflection APIs that are undergone a series of complex string manipulations. Leveraging static reflection analyses, SLIMMING captures all the bloated dependencies not actually used by the client projects. Furthermore, our tool derives debloating solutions by making a trade-off between the benefits (e.g., removal of bloated dependencies can also resolve the issues of vulnerabilities/dependency conflicts/incompatible licenses/outdated dependencies) and the costs (e.g., potentially inducing more maintenance effort) of dependency management.

To evaluate SLIMMING, we conduct a systematic study to assess the effectiveness of reflection analysis and the reliability of debloating solutions. We first construct a high-quality benchmark including 3,520 reflective calls captured by dynamic techniques from a collection of framework-based Java projects. SLIMMING achieves a *Precision* of 97.0% and a *Recall* of 98.8% on reflection resolution, which significantly outperforms all the state-of-the-art reflection analysis techniques. By collecting a set of Java projects with a 100% test coverage for their byte code instructions, we further validate whether the derived debloating solutions will induce new issues. The experiment results indicate that SLIMMING can reliably remove bloated dependencies with a 100% test passing ratio and 0 false alarm. Moreover, leveraging SLIMMING, we conducted a large-scale study on Java projects to reveal the landscapes of bloated dependencies in the Maven ecosystem. To evaluate the usefulness of our tool, we configure SLIMMING to report 484 bloated dependencies to 66 open-source projects. 38 reports (57.6%) have been confirmed by developers. The above feedback demonstrates the rationality of debloating solutions, which conform to developers' viewpoints on software maintenance.

In summary, the main contributions of this paper are as follows:

- **An effective debloating tool.** We develop SLIMMING as a Maven plugin to help users remove unused dependencies and reduce the frequent false alarms warned by build tools for vulnerability issues and license/version conflicts between dependencies.
- **Comprehensive datasets.** We provide (1) a high-quality benchmark including 3,520 reflective calls captured by dynamic techniques from a collection of framework-based Java projects for evaluating the effectiveness of reflection analysis techniques; (2) a dataset of 40 projects with 100% test coverage for Java byte code instructions for evaluating the reliability of debloating solutions.
- **Thorough comparisons.** We conduct systematic and thorough comparisons between SLIMMING and five state-of-the-art tools from different perspectives.
- **Large-scale analysis.** We leverage SLIMMING to conduct a large-scale study on 4,035 Java projects (including 8,337 modules) and identify 152,711 bloated dependencies from 2,503 projects. The investigation reveals the landscapes of bloated dependencies in the Maven ecosystem.
- **Data Availability.** We provided a reproduction package, including the above datasets, an available tool, and experiment raw data, on the website (<https://slimming-fat.github.io/>) for facilitating future research.



Figure 1: An illustrative example of challenges in resolving Java reflections

2 MOTIVATION AND CHALLENGES

2.1 Resolving Java Reflections

To precisely locate bloated dependencies, project developers should identify all the usage of third-party libraries. However, modern Java projects heavily leverage frameworks (e.g., *Spring Boot*) in their development. These frameworks commonly employ dynamic techniques (e.g., dependency injection), based on Java reflections, annotations, and configurations in *.xml, *.yml and *.properties files. Such dynamic patterns resist static reflection analysis, leading to false alarms of bloated dependencies.

Figure 1 shows an illustrative example of challenges in resolving Java reflections. Figure 1(a) extracts a code snippet from the open-source project Mall, which is implemented for sending messages to servers. As shown in Figure 1(b), the *SpringBoot* framework leverage API *SpringApplication.run()* (Line 14) to inject the information from annotation (Line 10) and configuration file *messageBean.xml* (Line 11) to the container instance *application* via *dependency injection* technique. The call chain triggered by *SpringApplication.run()* involves Java reflection, which accesses *class-retrieving method* *Class.forName* to return a set of class objects. The reflective target *io.minio.messages.CloudConfig* is returned by *SpringBoot* framework API *ApplicationContext.getBean()* (Line 15) via argument "CldConfig" from the container instance *application*. Unfortunately, the values of return types and API arguments can only be captured from data flow at runtime, rather than resolved by static reflection analysis. Since class *io.minio.messages.CloudConfig* belongs to the third-party library *io.minio:minio:8.5.3*, missing resolving this reflective target can lead to a false alarm of bloated dependency.

In addition, the call chain triggered by API *NotifConfig.getMessage()* (Line 16) also involves reflective calls. As shown in Figure 1(c), the *class-retrieving method* *Class.forName* (Lines 05) returns a class object representing a class that is specified by the value of its string argument "className". The reflective target *io.jsonwebtoken.Codec* has undergone two string operations "+" and "substring()" in Lines 02 and 04, respectively, which can only

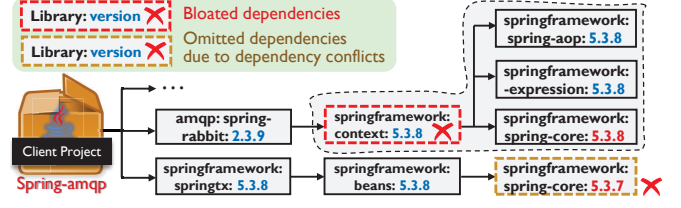


Figure 2: An illustrative example of challenges in removing bloated dependencies

be resolved by precise string analysis. However, existing static reflection analysis techniques built on top of Doop [22], model simple string operations including "+" and "append()" to resolve reflective targets, but they cannot deal with the other complex string operations. Missed inferring such a reflective target from a third-party library *io.jsonwebtoken:jjwt:0.9.1* can also result in a false alarm of bloated dependency.

Challenge #1: The frameworks (e.g., *SpringBoot*) commonly employ dynamic techniques combining Java reflections with annotations/configuration files. The reflective targets can involve the usage of third-party libraries, which are difficult to be resolved statically.

Challenge #2: Existing reflection analyses resolve class-level reflective targets by statically analyzing the string arguments of class-retrieving method calls. The arguments can be non-constant strings that undergo both intra-procedurally and inter-procedurally string operations (e.g., "append()"), which cannot be fully resolved by the existing techniques.

2.2 Removing Bloated Dependencies

Removing the identified bloated dependencies from configuration files (POM.xml) may induce new issues, since the solutions are not conducive to long-term dependency management for client projects. Figure 2 illustrates the dependency graph of a popular open-source project *Spring-amqp* (566 stars on GitHub). In issue report *Spring-amqp#65* [10], developers noticed that library *springframework:context:5.3.8* is a bloated dependency, which is not actually invoked by the client project. However, three libraries *spring-aop:5.3.8*, *expression:5.3.8*, and *spring-core:5.3.8* introduced by *springframework:context:5.3.8*, are used by project *Spring-amqp* through its direct dependency *amqp:spring-rabbit:2.3.9*. To ensure that the program can run correctly, after removing the bloated library *springframework:context:5.3.8* from *POM.xml*, developers should declare its three dependencies as project *Spring-amqp*'s direct dependencies. In the subsequent maintenance process, potential issues come with the above changes:

- After declaring the three libraries *spring-aop:5.3.8*, *expression:5.3.8*, and *spring-core:5.3.8* as direct dependencies, developers should spend much effort to keep their versions up-to-date. However, their updated versions may be incompatible with the asynchronously evolved version of library *amqp:spring-rabbit*.
- As shown in Figure 2, two versions of library *spring-core* are introduced in the dependency graph. The build tool loads version 5.3.8 while shadows version 5.3.7. Due to such a dependency conflict issue, library *Springframework:beans:5.3.8* actually invokes the loaded version *spring-core:5.3.8*. If developers declare library *spring-core* as a direct dependency, its updated version may also induce incompatibility issues with *springframework:beans*.

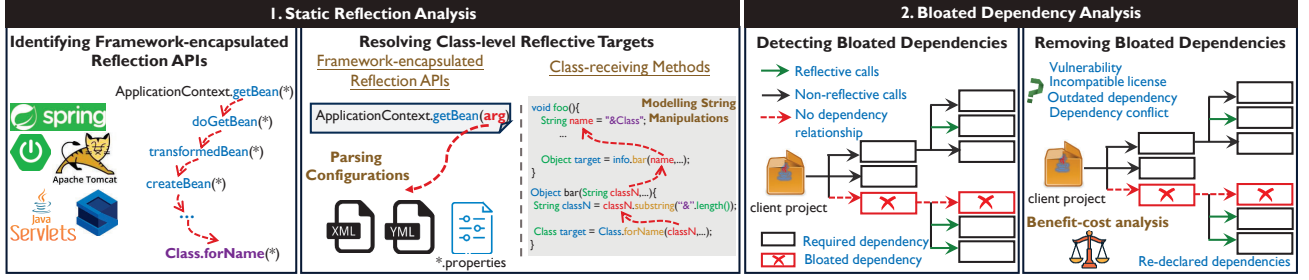


Figure 3: The overall architecture of SLIMMING

In this scenario, developers abandoned such a solution, since they thought the removal of bloated dependency would bring greater maintenance costs to their project. Interestingly, by inspection, we noticed similar cases happened in project *Jax-rs2-guide* (Jax-rs2-guide#203 [9]). However, since the bloated dependency exposed a high-severity level vulnerability (CVE-2022-22976), Jax-rs2-guide’s developer finally decided to remove it at the expense of a higher maintenance burden, after making a trade-off on dependency management.

Challenge #3: Removing bloated dependencies can easily induce new issues that are not conducive to dependency management. There is no existing technique to help developers analyze the solutions of removing bloated dependencies, by weighing the benefits (e.g., removal of bloated dependencies can also resolve the issues of vulnerabilities/dependency conflicts/incompatible licenses/outdated dependencies) against the costs (e.g., potentially induce more maintenance effort) on dependency management.

3 SLIMMING APPROACH

To address the limitations of prior debloating approaches, we developed a technique, SLIMMING, to help developers reliably remove the bloated dependencies. Figure 3 shows the overall architecture of SLIMMING. To precisely identify bloated dependencies, our tool is empowered by effective *static reflection analyses* that can overcome *Challenges#1* and *#2*. As illustrated in Section 2.1, the frameworks encapsulate a set of publicly accessible APIs to deal with dynamic language features, which transitively reach reflective call sites (we refer to such public APIs as **framework-encapsulated reflection APIs**). In this paper, to address *Challenge #1*, SLIMMING leverages a scalable technique to identify all the framework-encapsulated reflection APIs from popular frameworks. Based on static analysis, our tool further parses the arguments of framework-encapsulated reflection APIs combined with framework configuration files (*.xml, *.yaml and *.properties) to resolve the class-level reflective targets. To address *Challenge #2*, SLIMMING models all the string operations defined by JDK (e.g., *String.substring()*) for inferring non-constant strings of class-receiving methods.

Based on static reflection analysis, SLIMMING captures all the bloated dependencies not actually used by the client project. To address the *Challenge #3* regarding removing bloated dependencies, our tool generates the solutions by weighing the benefits (e.g., removal of bloated dependencies can also resolve the issues of vulnerabilities/dependency conflicts/incompatible licenses/outdated dependencies) against the costs (e.g., potentially inducing more maintenance effort) of dependency management.

3.1 Static Reflection Analysis

SLIMMING performs the static reflection analysis in two steps:

Step 1: Identifying Framework-encapsulated Reflection APIs.

This step involves two tasks:

- **Constructing Complete Call Graph for Popular Frameworks.** To comprehensively identify framework-encapsulated reflection APIs, SLIMMING leverages Soot [18] to statically analyze six frameworks that are commonly used by Java projects, including *Java EE Servlets*, *Java Beans*, *Apache Struts*, *Spring*, *Spring Boot* and *Tomcat*. Based on Soot’s CHA algorithm, SLIMMING constructs a complete method-level call graph for each framework by configuring a set of public APIs as entry points to find all the reachable methods (setting `all-reachable` option of Soot).
- **Locating Framework-encapsulated Reflection APIs.** From the constructed call graph, SLIMMING identifies the call chains that can reach the reflective call sites (i.e., class-retrieving methods, such as `Class.forName` and `ClassLoader.loadClass`). Furthermore, on these identified call chains, SLIMMING locates the framework APIs that satisfy three criteria as framework-encapsulated reflection APIs: (1) they are publicly accessible by the client projects; (2) they contain at least one argument of string type (framework-encapsulated reflection APIs communicate with configuration files *.xml, *.yaml and *.properties via string arguments); (3) their return types are abstract classes, interfaces, or superclasses in the inheritance hierarchy (the instantiated class objects are determined by the reflective targets).

Based on the above process, SLIMMING eventually identifies 294 framework-encapsulated reflection APIs from the six popular Java frameworks. Among them, 40, 32, 42, 65, 79, and 36 APIs are located from *Java EE Servlets*, *Java Beans*, *Apache Struts*, *Spring*, *Spring Boot* and *Tomcat*, respectively.

Step 2: Resolving Class-level Reflective Targets. Since we aim to identify the bloated dependencies at the library level from configurations, for static analysis, SLIMMING resolves the reflective targets at the class level. Resolving reflective targets mainly relies on analyzing the string arguments to Java reflection APIs. For a given Java project, SLIMMING statically analyzes its call graph and performs the following tasks to deal with framework-encapsulated reflection APIs and class-retrieving methods separately.

a. Analyzing Framework-encapsulated Reflection APIs. On the call graph of the project under analysis, SLIMMING first locates the framework-encapsulated reflection APIs. Furthermore, leveraging framework configuration files, it analyzes the API arguments to resolve the class-level reflective targets in the following two steps:

- *Parsing Configuration Files.* Typically, framework configuration files (*.xml, *.yaml and *.properties) record the fully qualified class name as the reflective targets and each class name corresponds to a string literal. Framework-encapsulated reflection APIs return the reflective targets using the string literals as their arguments to map the class names from configuration files. For example, in the code snippet of Figure 1(a), the framework-encapsulated reflection API `ApplicationContext.getBean(*)` communicates with the container instance `application` (whose information is injected by configuration file `messageBean.xml`) via the string argument "CldConfig" to return the corresponding reflective target `io.minio.messages.CloudConfig`. The mapping of reflective target and string literal "CldConfig" is configured in `messageBean.xml` file. Therefore, to perform static reflection analysis, SLIMMING collects and parses three types of configuration files in a given project to extract the mappings of the fully qualified class name and its corresponding string literal. The parsing patterns of SLIMMING for configuration files are described in Figure 4. Each pattern identifies the concerned mappings $attr_1 \rightarrow attr_2$, according to the syntax rules of *.xml, *.yaml and *.properties, where $attr_1$ is a string attribute extracted from the configuration files, and $attr_2$ is $attr_1$'s attribute value in the format of class name.

- *Resolving Reflective Targets.* For each identified framework encapsulated reflection API, SLIMMING checks if its string arguments map a fully qualified class name in the configuration file. Our tool considers the mapped class name as a reflective target returned by the framework-encapsulated reflection API.

b. Analyzing Class-Retrieving Methods. In addition to the framework-encapsulated reflection APIs, SLIMMING deals with four class-retrieving methods that are most widely used by modern Java projects [30]: `Class::forName`, `ClassLoader::loadClass`, `Object::getClass` and `.class`. Java reflection begins with class objects, which are returned by calling class-retrieving methods. Note that the class object returned by `O.getClass()` and `A.class` represents the dynamic type (class) of `O` and `A`, respectively. SLIMMING can efficiently resolve the reflective targets by statically analyzing these two class-retrieving methods. However, `Class::forName` and `ClassLoader::loadClass` returns a class object representing a class that is specified by the value of their string arguments. Therefore, resolving reflective targets depends on analyzing the string arguments of the above two class-retrieving methods.

If the string arguments of `Class::forName` and `ClassLoader::loadClass` are constant strings, SLIMMING considers the arguments of class names as reflective targets. However, according to an investigation on reflection usage in popular Java projects [30], in practice, nearly 65.3% of reflection API arguments are non-constant strings that are undergone intra-procedural and inter-procedural string manipulations. To precisely resolve the non-constant string arguments, SLIMMING models all the string operations defined by JDK (e.g., `String.length()`, `String.substring()`) for inferring string manipulations, which involves three tasks:

- *Tagging the end location of string manipulations.* SLIMMING identifies each class-receiving method with string arguments and tags its first string argument as the end location of string manipulations. The first string argument of class-receiving methods

`Class::forName` and `ClassLoader::loadClass` is the outcome of string manipulations.

- *Tagging the start location of string manipulations.* From each tagged end location, SLIMMING performs pointer analysis to trace the data flow, and locates a string assignment statement that satisfies two conditions: (1) The value assigned is a string constant; (2) The assigned string variable is not an argument of the function in which it is defined. SLIMMING tags this string variable as the corresponding start location of string manipulations.
- *Modeling string manipulations.* The algorithm of modeling string manipulations is described in Figure 5, which takes the tagged start location (`start_location`) and end location (`end_location`) of string manipulations as inputs and returns the manipulation outcome (`model_result`). The algorithm works as follows. It first initiates `model_result` and assigns the `start_location` to `start_variable` (Lines 1–2). To fully resolve the non-constant string arguments of reflection APIs, our tool collects 72 types of string operations are defined by JDK and assigns the signature array to `StringOperations` (Line 3). SLIMMING iteratively analyzes each statement `tmp_statement` invoked between `start_location` and `end_location` and utilizes `start_variable` to point to the current statement under analysis (Line 5). If `tmp_statement` involves a string operation API (e.g., `String.substring()`), SLIMMING then extracts the API's signature and arguments and the instance object of string manipulation (Lines 6–7). Next, our tool invokes the corresponding string operation API with the above extracted arguments to simulate the string manipulations (the simulated outcome is assigned to `model_result`) (Lines 8–11). Furthermore, SLIMMING assigns the current value of `model_result` to the concerned string argument/variable referenced by the subsequent string manipulation relevant statement (Line 12). If the `tmp_statement` does not involve a string operation API, SLIMMING traces the data flow based on pointer analysis (Lines 13–14). Our tool iteratively performs the above tasks until its pointer moves to the end location of string manipulations.

3.2 Bloated Dependency Analysis

Based on the dependency relationships captured by static reflection analysis, SLIMMING remove the bloated dependencies in two steps: **Step 1: Detecting Bloated Dependencies.** Leveraging Soot, SLIMMING first obtain the class-level dependency graph of a given project based on static analysis. Our tool further adds the dependencies between classes that are captured by reflection analysis to construct a complete dependency graph. SLIMMING considers a library as a bloated dependency if none of its defined classes exists in the dependency graph.

Step 2: Removing Bloated Dependencies. SLIMMING performs a benefit-cost analysis to remove the bloated dependencies reasonably. It takes the following factors into consideration:

- *Factors that bring additional benefit to dependency management:* SLIMMING considers a solution that brings additional benefit to dependency management if removing bloated dependencies can also resolve four types of issues:
 - *Vulnerability issues:* The bloated dependencies contain the vulnerabilities that have been disclosed in vulnerability databases. SLIMMING collects the vulnerable library information from

File Type	Parsing Pattern	Illustrative Example								
*.xml	<p>(1) For each tag of a *.xml file, Slimming extracts all the attribute values of string type.</p> <p>(2) For the extracted string attributes, Slimming constructs the mappings between them in the following format: <i>attr₁</i> → <i>attr₂</i>, where <i>attr₁</i> is a string attribute other than the class name, and <i>attr₂</i> is a string of the class name.</p>	<p>(1) *.xml configuration file</p> <pre><bean id="CldConfig" class="io.minio.messages.CloudConfig"> <constructor-arg type="io.jsonwebtoken.ClaimJwtException" index="0" ref="expiredJwtException"/> <constructor-arg index="1" value="130"/> </bean></pre> <p>(2) Parsing pattern</p> <table><thead><tr><th>xml Tag</th><th>Attribute Value</th></tr></thead><tbody><tr><td><bean></td><td>(1) CldConfig ; (2) io.minio.messages.CloudConfig</td></tr><tr><td><constructor-arg></td><td>(1) io.jsonwebtoken.ClaimJwtException; (2) expiredJwtException</td></tr></tbody></table> <p>(3) Extracted mappings</p> <p><i>CldConfig</i> → <i>io.minio.messages.CloudConfig</i> <i>expiredJwt</i> → <i>io.jsonwebtoken.ClaimJwtException</i></p>	xml Tag	Attribute Value	<bean>	(1) CldConfig ; (2) io.minio.messages.CloudConfig	<constructor-arg>	(1) io.jsonwebtoken.ClaimJwtException; (2) expiredJwtException		
xml Tag	Attribute Value									
<bean>	(1) CldConfig ; (2) io.minio.messages.CloudConfig									
<constructor-arg>	(1) io.jsonwebtoken.ClaimJwtException; (2) expiredJwtException									
*.yaml	<p>(1) Slimming extracts the attributes and their corresponding attribute values that are represented in hierarchical structures from a *.yaml file.</p> <p>(2) For the extracted attributes and their corresponding values, Slimming constructs the mappings in the following format: <i>attr₁</i> → <i>attr₂</i>, where <i>attr₁</i> is a string attribute, and <i>attr₂</i> is <i>attr₁</i>'s attribute value of the class name.</p>	<p>(1) *.yaml configuration file</p> <pre>spring: datasource: username: root password: * driver-class-name: com.mysql.cj.jdbc.Driver</pre> <p>(2) Parsing pattern</p> <table><thead><tr><th>yaml Attribute</th><th>Attribute Value</th></tr></thead><tbody><tr><td>spring.datasource.username</td><td>root</td></tr><tr><td>spring.datasource.password</td><td>*</td></tr><tr><td>spring.datasource.driver-class-name</td><td>com.mysql.cj.jdbc.Driver</td></tr></tbody></table> <p>(3) Extracted mappings</p> <p><i>spring.datasource.driver-class-name</i> → <i>com.mysql.cj.jdbc.Driver</i></p>	yaml Attribute	Attribute Value	spring.datasource.username	root	spring.datasource.password	*	spring.datasource.driver-class-name	com.mysql.cj.jdbc.Driver
yaml Attribute	Attribute Value									
spring.datasource.username	root									
spring.datasource.password	*									
spring.datasource.driver-class-name	com.mysql.cj.jdbc.Driver									
*.properties	<p>(1) Slimming extracts the attributes and their assigned attribute values directly from a *.properties file.</p> <p>(2) For the extracted attributes and their corresponding values, Slimming constructs the mappings in the following format: <i>attr₁</i> → <i>attr₂</i>, where <i>attr₁</i> is a string attribute, and <i>attr₂</i> is <i>attr₁</i>'s attribute value of the class name.</p>	<p>(1) *.properties configuration file</p> <pre>spring.jpa.hibernate.ddl-auto=update naming_strategy=org.cfg.NamingStrategy spring.jpa.database=H2</pre> <p>(2) Parsing pattern</p> <table><thead><tr><th>Attribute</th><th>Attribute Value</th></tr></thead><tbody><tr><td>spring.jpa.hibernate.ddl-auto</td><td>update</td></tr><tr><td>naming_strategy</td><td>org.cfg.NamingStrategy</td></tr><tr><td>spring.jpa.database</td><td>H2</td></tr></tbody></table> <p>(3) Extracted mappings</p> <p><i>naming_strategy</i> → <i>org.cfg.NamingStrategy</i></p>	Attribute	Attribute Value	spring.jpa.hibernate.ddl-auto	update	naming_strategy	org.cfg.NamingStrategy	spring.jpa.database	H2
Attribute	Attribute Value									
spring.jpa.hibernate.ddl-auto	update									
naming_strategy	org.cfg.NamingStrategy									
spring.jpa.database	H2									

Figure 4: Parsing patterns of SLIMMING for configuration files

Algorithm 1: Modelling string manipulations

Input: $start_location$; $end_location$
Output: $model_result$

```

1 Initiate  $model\_result$ ;
2  $start\_variable \leftarrow start\_location$ ;
3  $stringOperations \leftarrow \{String:String.substring(int),String:int.length(),...\}$ ;
4 while  $start\_variable \neq end\_location$  do
5    $tmp\_statement = get\_statement(start\_variable)$ ;
6   if  $tmp\_statement.Contains(stringOperations)$  then
7      $(instanceObject, APISignature, arg) \leftarrow method\_analysis(tmp\_statement)$ ;
8     if  $instanceObject's\ type \in \{Integer, String, Char\}$  then
9        $model\_result \leftarrow valueOfObject(instanceObject).APISignature(arg)$ ;
10    else
11       $model\_result \leftarrow instanceObject.APISignature(arg)$ ;
12       $start\_variable \leftarrow update\_variable(model\_result)$ ;
13    else
14       $start\_variable \leftarrow pointer\_analysis(tmp\_statement)$ ;
15 return  $model\_result$ ;
```

Figure 5: Modelling string manipulations

two recognized databases *GitHub Advisory DB* [8] and *Snyk Vulnerability DB* [17], as the basis for assessment.

- *Incompatible license issues:* The open source licenses declared by bloated dependencies are incompatible with that of a client project. SLIMMING checks the incompatible license issues based on compatible relationships among 389 popular open source licenses provided on *Free Software Foundation* website [7].
- *Outdated dependency issues:* The bloated dependencies are the libraries that have not been maintained by developers for more than N years. Based on the outdated dependencies defined by approach [26], SLIMMING sets $N = 3$ and collects the last updated date of each project in *Maven Central Repository* when performing analysis.
- *Dependency conflict issues:* Multiple versions of a library that are directly or transitively introduced into a client project are all identified as bloated dependencies. SLIMMING captures the dependency conflict issues leveraging the warnings reported by *maven-dependency-tree* plugin [15].
- *Factors that induce costs on dependency management:* Let $BD = \{lib_1, lib_2, \dots, lib_n\}$ be a collection of identified bloated dependencies of a given project; $TD_i = \{lib_1, lib_2, \dots, lib_m\}$ be a

set of libraries transitively introduced by a bloated dependency $lib_i \in BD$; $RD_i \subseteq TD_i$ be a set of libraries in TD_i that are referenced by the client project. We refer to a library $lib_k \in RD_i$ as the *over dependency management cases*, which needs to be additionally declared as the client project's direct dependency after removing the bloated library $lib_i \in BD$ to avoid inducing runtime errors. At the cost of the above changes, developers would spend the effort to keep the additionally declared library versions up-to-date. However, their updated versions can easily induce incompatibility issues with the libraries depending on them.

A benefit-cost analysis for a debloating solution. For a bloated dependency $lib_i \in BD$, SLIMMING performs a benefit-cost analysis to generate a reliable debloating solution as follows:

- *Case 1:* If $TD_i = \emptyset$, SLIMMING removes the bloated dependency lib_i from configuration file.
- *Case 2:* If $TD_i \neq \emptyset$ and $RD_i = \emptyset$, SLIMMING removes the bloated dependency lib_i from configuration file.
- *Case 3:* Suppose that $TD_i \neq \emptyset$, $RD_i \neq \emptyset$, and the bloated dependency lib_i also induces the issues of vulnerabilities/dependency conflicts/incompatible licenses/outdated dependencies. Since removing lib_i can bring additional benefits to dependency management, SLIMMING suggests developers perform such a debloating solution at a cost of additionally declaring its dependencies $lib_k \in RD_i$ as direct dependencies of the client project.

According to the syntax rules of Maven dependency configurations, if the bloated dependencies are inherited from the parent module, SLIMMING removes them with the aid of `<provided>` tag of `POM.xml` file. Otherwise, SLIMMING performs debloating solutions using `<exclusion>` tag. SLIMMING triggers projects' bundled tests to validate whether the debloating solution will cause program errors. Our tool finally reports a collection of bloated dependencies and the corresponding debloating solution if all the tests are passed.

4 EVALUATION

We study two research questions in our evaluation section:

Table 1: Demographics of collected Java projects for RQ1 and RQ2

Metrics	Statistics	Max.	Min.	Avg.	Med.
#Tests		737	1	27	4
#Stars		34,200	5	4,527	55
#Modules		14	1	5	4
#Direct Dependencies		30	6	13	10

Table 2: Statistics of our constructed benchmark

	Type 1	Type 2	Type 3	Type 4	Others	Sum
DaCaPo	143	84	0	4	10	241
37 Java projects	1,888	84	1,082	213	12	3,279
Sum	2,031	168	1,082	217	22	3,520

- **RQ1 (Effectiveness of Reflection Analysis): How effective is SLIMMING in analyzing Java reflections?** To answer RQ1, we constructed a high-quality benchmark and compare SLIMMING with three state-of-the-art reflection analysis techniques to evaluate their effectiveness.
- **RQ2 (Reliability of Debloating Solutions): Can SLIMMING reliably remove the bloated dependencies without inducing new issues?** To answer RQ2, we collected 40 Java projects with 100% test coverage for their byte code instructions, to validate whether removing bloated dependencies will induce new issues.

4.1 Data Collection

Following the prior approaches [30, 39], we adopt 6 large DaCaPo datasets (2006-10-MR2) [5] and 3,520 reflective calls captured by dynamic techniques from 37 framework-based Java projects as benchmarks, to evaluate SLIMMING’s effectiveness in reflection analysis (RQ1). The benchmark construction process involves two steps:

- **Collecting Framework-based Java Projects:** Since the subjects in DaCaPo datasets (collected in 2006) do not rely on the frameworks such as *Spring Boot*, etc., they cannot comprehensively reflect the reflection resolving capability of SLIMMING. As such, we extend the datasets by collecting more representative framework-based Java projects. The project collection satisfies **four criteria**: (1) the projects have more than 5 Stars (i.e., popularity); (2) the projects have more than five direct dependencies (i.e., complexity); (3) the projects’ bundled tests achieve 100 percent of instruction coverage¹. Such high test coverage for Java byte code instructions enables us to completely capture the reflective targets, leveraging the dynamic program analysis tool TAMIFLEX [21]; (4) the projects depend on at least one popular framework. With the above process, we obtained 37 Java projects (37.8%, 21.6%, 13.5%, 8.1%, 13.5%, and 5.4% of them rely on *Spring Boot*, *Spring*, *Tomcat*, *Java EE Servlets*, *Java Beans* and *Apache Struts* frameworks, respectively). Table 1 describes the demographics of the collected Java projects. On average, each project has 27.3±81.5 tests, 4,579.5±11,023.4 stars, 5.4±4.6 modules, and 12.8±6.1 direct dependencies.
- **Identifying All the Reflective Targets Using TAMIFLEX:** TAMIFLEX is an effective dynamic analysis tool proposed by approach [21], which inserts runtime checks into the program to capture the reflective calls triggered by program execution. Leveraging TAMIFLEX, we execute the tests bundled with the 37 Java projects

¹We leverage a popular Java code coverage library JaCoCo [11] to measure the instruction coverage. Instruction coverage provides information about the amount of code that has been executed or missed.

Table 3: Experiment results of RQ1

Baselines	Metrics	TP	FP	FN	Precision	Recall	F-measure
Doop [6]		2,840	3,507	680	44.8%	80.7%	57.6%
JackEE [20]		3,020	2,709	500	52.7%	85.8%	65.3%
Soot [18]		2,693	5,042	827	34.8%	76.5%	47.9%
SLIMMING		3,477	107	43	97.0%	98.8%	97.9%

(100% test coverage for byte code instructions), to precisely and completely obtain the reflective targets.

Table 2 shows the statistics of our constructed benchmark. In total, TAMIFLEX captures 3,520 reflective targets. Among them, 2,032 (57.7%) are received from non-constant string arguments of class-receiving methods (**Type 1**), which are undergone a series of string manipulations; 168 (4.8%) are received from constant string arguments of class-receiving methods (**Type 2**); 1,082 (30.7%) are recorded in the framework configuration files, which need to communicate with framework-encapsulated reflection APIs (**Type 3**); 217 (6.2%) are embellished with annotations (**Type 4**); 22 (0.6%) are implemented in other manners (e.g., retrieving from the Internet).

4.2 RQ1: Effectiveness of Reflection Analysis

Study Methodology. We compare SLIMMING with three state-of-the-art reflection analysis techniques to evaluate their effectiveness:

- **Doop [6]** is a state-of-the-art framework for Java pointer and taint analysis. In addition to the basic options (e.g., ‘reflection-classic’) provided by Doop for reflection analysis, we also turned on the option ‘reflection-substring-analysis’ proposed by approach [39] (APLAS 2015) to resolve reflective targets based on string-flow analysis.
- **JackEE [20]** is a static analysis framework relying on the context of Doop (PLDI 2020), to resolve the reflections that are synthesized by *dependency injections*, *Java annotations* and *configurations* in Enterprise Frameworks including *Java EE Servlets*, *Java Beans*, *Spring*, and *Apache Struts*.
- **Soot [18]** is the best-known static analysis framework to construct dependency graphs for Java programs. We turn on the options ‘Safe forName’ and ‘Safe newInstance’ provided by Soot for reflection resolution.

All the baseline approaches extract the dependency relationships at method granularity, in our study, we only consider the resolved reflective class names to make the comparisons.

Evaluation Metrics. Leveraging our constructed benchmark, we consider six metrics in the evaluation: (1) **True Positive (TP)**: the reflective targets resolved by SLIMMING are recorded in the benchmark by TAMIFLEX; (2) **False Positive (FP)**: the reflective targets resolved by SLIMMING are not recorded in the benchmark by TAMIFLEX; (3) **False Negative (FN)**: the reflective targets are recorded in the benchmark by TAMIFLEX but not resolved by SLIMMING.

Based on the above three metrics, we can obtain the *Precision*, *Recall* and *F-measure* as follows: (a) **Precision** = $TP / (TP + FP)$; (b) **Recall**: $TP / (TP + FN)$; (c) **F-measure**: $2 \times Precision \times Recall / (Precision + Recall)$. *Precision* evaluates whether SLIMMING can resolve reflective targets precisely. *Recall* evaluates the capability of SLIMMING in resolving all the reflective targets. *F-measure* combines the *Precision* and *Recall* together [46].


Table 4: Effectiveness of analyzing different types of reflective targets

Baselines	Types					Type Inference
	Type 1	Type 2	Type 3	Type 4	Others	
DOOP [6]	728	168	0	165	0	1,779
JACKEE [20]	728	168	828	217	0	1,079
SOOT [18]	623	168	0	217	0	1,685
SLIMMING	1,982	168	1,075	217	0	35

Results. Table 3 shows the experiment results of RQ1. SLIMMING resolves 3,584 reflective targets with a *Precision* of 97.0%, a *Recall* of 98.8% and a *F-measure* of 97.9%. In contrast, DOOP, JACKEE and SOOT report a large number of false positive and negative cases, resulting in *Precision* = 44.8%, 52.7% and 34.8%, and *Recall* = 80.7%, 85.8%, and 76.5% respectively. By checking the analytic effectiveness for different types of reflective targets (as shown in Table 4), we found that: SLIMMING can resolve 1,982 out of 2,031 (97.6%) non-constant string arguments of class-receiving methods by modeling string manipulations (*Type 1*) and 1,075 out of 1,082 (99.4%) class names recorded in framework configuration files (*.xml, *.yaml and *.properties) by analyzing framework-encapsulated reflection APIs (*Type 3*). Based on the results, we can conclude that SLIMMING can effectively resolve different types of reflective targets, significantly outperforming all the baselines.

FP Analysis. For all the static reflection analysis techniques, if they cannot resolve the non-constant string arguments of class-receiving methods, they will mock the return types of class-receiving methods as reflective targets. In the cases that the class-receiving methods’ return types are abstract classes, interfaces, or super-classes, the techniques will infer all the potential types of instantiated class objects as the results. “Type Inference” column of Table 4 shows *#return types of class-receiving methods* that are mocked by these techniques. DOOP, JACKEE and SOOT report 3,507, 2,709 and 5,042 FPs, respectively, because they are less effective in dealing with *Types 1* and *3* cases.

FN Analysis. SLIMMING produces 43 FNs since it cannot resolve 14 non-constant string arguments of class-receiving methods (*Type 1*), 7 classes recorded in the framework configuration files (*Type 3*) and 22 classes received from Internet (*Others*). For the 14 *Type 1* cases, SLIMMING is unable to simulate the string manipulations that involve string arrays. For the 7 *Type 3* cases, these framework-based projects use customized types of files to configure reflective targets rather than *.xml, *.yaml or *.properties. Moreover, the three baselines can only deal with simple string operations including “+” and “append()” to resolve reflective targets, as such they produce many FNs for *Type 1* cases. JACKEE can handle few framework-encapsulated reflection APIs and analyze *.xml configurations in 2 popular frameworks including *Java Beans* and *Spring*, resulting in 254 FNs for *Type 3* cases.

 **Conclusion:** SLIMMING achieves a Precision of 97.0% and a Recall of 98.8%, which significantly outperforms all the baselines.

4.3 RQ2: Reliability of Debloating Solutions

Study Methodology. We adopt the 40 collected Java projects that satisfy the *Selection Criteria (1-3)* defined in Section 4.1 as a dataset for RQ2, since their 100% test coverage for Java byte code instructions can help validate whether removing the identified

bloated dependencies will cause program errors. Two state-of-the-art debloating tools are considered as baselines to compare their reliability of debloating solutions:

- **Maven Dependency Analyzer (MDA)** [14] is an official Maven plugin that is actively maintained by the Maven team (supported by the *Apache Software Foundation*) for identifying undeclared or unused libraries of Java projects.
- **DEPCLEAN** [42] is proposed to identify bloated dependencies of Maven projects (*ESE 2021*), including both direct/transitive dependencies declared in *POM.xml* and inherited dependencies declared in parent *POM.xml*.

Evaluation Metrics. We define four evaluation metrics:

- **TPR (Test Passing Ratio):** Proportion of the debloated projects that can pass bundled tests.
- **#FA (Number of False Alarms):** Number of the removed dependencies that are invoked by client project via Java reflections.
- **#IDC (Number of Induced Dependency Conflicts):** Number of dependency conflict issues induced by the debloating solutions.
- **#ODMC (Number of Over Dependency Management Cases):** Number of additionally declared direct dependencies that are derived by the debloating solutions.

In order to calculate *#IDC*, we leverage DECCA [45], a state-of-the-art detection tool to capture dependency conflicts. Specially, we are only concerned with the dependency conflict issues that can cause runtime errors (e.g., *NoSuchMethodError*).

Results. Table 5 shows the experiment results of RQ2. For the 40 collected Java projects, on average, the debloating solutions generated by SLIMMING reliably remove 7 bloated dependencies with a 100% test passing ratio and 0 false alarms. In contrast, on average, MDA and DEPCLEAN remove 7 and 13 bloated dependencies with the 57.5% and 73.6% test passing ratios, and 1 and 2 false alarms, respectively. SLIMMING outperforms the other two baselines on all the evaluation metrics.

#FA Analysis. Owing to SLIMMING’s effective static reflection analysis, it precisely identifies all the usage of third-party libraries without producing false alarms. DEPCLEAN is implemented as a Maven plugin that extends the MDA. Both of them rely on the ASM library [4] to perform the static analysis, which visits all the .class files of the compiled projects in order to register bytecode calls towards classes, methods, and fields among projects and their dependencies. Since ASM cannot deal with reflection calls well, these two baselines can produce false alarms for bloated dependencies.

#IDC and #ODMC Analyses. Due to its effective benefit-cost analysis, the solutions do not induce dependency conflict issues and on average only additionally declare 1 direct dependency, after removing the bloated dependencies. DEPCLEAN adopts an alternative debloating solution: it first adds all the used transitive dependencies as direct dependencies and then excludes all the bloated dependencies. As such, DEPCLEAN induces many over-dependency management cases, which potentially induce new issues (e.g., dependency conflicts) and bring much maintenance burden to project developers. Since MDA does not provide solutions for removing bloated dependencies, we cannot calculate its *#ODMC* metric.

Table 5: Experiment Results of RQ2

	#BD			TPR (%)			#FA			#IDC			#ODMC		
	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.	Max.	Min.	Avg.
MDA	25	0	7	100	0	57.5	6	0	1	5	0	2	-	-	-
DepClean	50	0	13	100	40.0	73.6	8	0	2	3	0	1	168	0	30
SLIMMING	32	0	7	100	100	100	0	0	0	0	0	0	3	0	1

† #BD denotes the number of bloated dependencies reported by the tools

Table 6: Statistics of the 4,035 projects in our large-scale study

Metrics	Statistics	Max.	Min.	Avg.	Med.
#Stars		25,530	6	425.1	22
#Modules		134	1	12.4	3
#Direct Dependencies		97	6	11.3	9
#Transitive Dependencies		467	0	59.1	45

Conclusion: On our collected Java projects, on average, the debloating solutions generated by SLIMMING reliably remove 7 bloated dependencies with a 100% TPR and 0 FAs. Due to its effective benefit-cost analysis, the solutions do not induce dependency conflict issues and only additionally declare a few direct dependencies after removing the bloated dependencies.

5 LARGE-SCALE ANALYSIS

Leveraging SLIMMING, we conducted a large-scale study on Java projects to reveal the landscapes of bloated dependencies in the Maven ecosystem. Besides, we further study the research question:

- **RQ3 (Usefulness of SLIMMING):** *Can SLIMMING provide useful solutions to assist developers remove bloated dependencies? Do the debloating solutions conform to developers' viewpoints on software maintenance?* To answer RQ3, we report the bloated dependencies with detailed diagnosis info and feasible solutions to projects' issue trackers. We assess the usefulness of SLIMMING based on developers' feedback.

5.1 Landscapes of Bloated Dependencies

Study Methodology. For a large-scale empirical study, we queried the GitHub API on March 1st of, 2023 to obtain a list of GitHub URLs, including all projects that use Java as the primary programming language. From this list, we kept the projects as subjects if they: (1) achieve more than 5 Stars (*popularity*); (2) have more than five direct dependencies (*complexity*). The initial dataset contains a total of 21,200 Java projects. In the second step, we compiled the newest release version of each collected Java project using Maven. This compilation process is a challenging task involving resolving and downloading dependencies. To increase the ratio of successful compilations, we tried to compile the projects using Java 8 if they did not compile with Java 11. Unfortunately, 17,165 Java projects still failed to compile due to: (1) lacking dependencies (parts of dependencies were only provided in private repositories); (2) resolving the incompatible dependency versions (developers did not explicitly declare their version constraints). Eventually, we obtained 4,035 Java projects (including 8,337 modules) that passed the compilations as the definitive dataset for our empirical study could be further analyzed by SLIMMING.

Table 6 shows the descriptive statistics of the 4,035 Java projects. 991 of them are multi-module projects, and the number of modules ranges from 1 to 134 (Avg. = 12.4, Med. = 3). The last two lines in the table give the number of direct dependencies (Avg. = 11.3, Med. = 9) and transitive dependencies (Avg. = 59.1, Med. = 45).

Results. Table 7 summarizes the statistics of our investigation results. With a longitudinal analysis of 4,035 Java projects, SLIMMING identified 152,711 bloated dependencies from 2,503 projects (62.0%). On average, each project contains 30.5 ± 22.7 bloated dependencies (0.4 ± 0.7 bloated direct dependencies and 27.9 ± 21.3 bloated transitive dependencies), accounting for 39.7% of the total number of their third-party libraries. For the 775 multi-module projects, nearly 8.1% of bloated dependencies are inherited from parent POM.xml.

We identified 10,800 dependencies only invoked by client projects via Java reflections. Specifically, the 16,200 reflective targets resolved by SLIMMING involve complicated implementations, including non-constant string arguments (58.3%) and constant string arguments (4.8%) of class-receiving methods, classes recorded in the framework configuration files (30.7%), and classes embellished with annotations (0.6%). These libraries can easily be mistaken for bloated dependencies by state-of-the-art debloating tools. SLIMMING precisely resolves the reflective calls to avoid false alarms due to its effective static analysis capability.

By further inspecting the 152,711 bloated dependencies, we found that 37,027 (24.3%) were outdated libraries, 3,157 (2.1%) were vulnerable libraries, 901 (0.6%) induced license conflicts, and 13,823 (9.1%) induced dependency conflict issues. However, we observed that many project developers invested non-trivial effort to upgrade these libraries and resolve the warnings of vulnerabilities/license conflicts/dependency conflicts caused by bloated dependencies, which was not required. For example, a popular project Logisland [12] (106 Stars on GitHub) introduced 152 bloated dependencies. However, during the maintenance process, developers submitted four commits to the GitHub repository with the purpose of keeping the bloated library versions up-to-date. Since 14 bloated dependencies were vulnerable libraries, project Logisland [12] continually received high-severity vulnerability warnings against them from build tools. Reliably removing the bloated dependencies can help developers save unnecessary maintenance costs and reduce the frequent false alarms warned by build tools for vulnerability issues and license/version conflicts between dependencies.

5.2 RQ3: Usefulness of SLIMMING

Study Methodology. From the 2,503 Java projects with bloated dependencies detected by SLIMMING in Section 5.1, we selected the ones that satisfy two criteria to report to their corresponding issue trackers: (1) their code repositories have commit records in the past year (actively maintained); (2) the number of bloated dependencies is greater than 5, or the bloated dependencies induce dependency management issues (e.g., vulnerabilities). Following the criteria, we finally chose 66 subjects. Our issue reports describe the bloated dependencies with detailed diagnosis info and file the corresponding feasible debloating solutions as *Pull Requests* (PRs) to code repositories. To allow developers better understand the benefit-cost analysis of debloating solutions, we also provide a figure to visualize the library dependency graph with labeled additionally

Table 7: Statistics of bloated dependencies in our large-scale study

Statistics in Each Project	Max.	Min.	Avg.	Med.
#Bloated dependencies	198	1	30.1	25
Proportion of bloated dependencies to all the introduced dependencies (%)	75.9	3.3	41.9	38.3
#Bloated direct dependencies	6	0	0.38	0
#Bloated transitive dependencies	193	0	27.9	23
#Bloated dependencies inherited from parent POM	156	0	2.2	0
#Dependencies only invoked by client project via Java reflections	26	0	2.2	2
#Bloated Dependencies that are outdated libraries	153	0	7.4	4
#Bloated Dependencies that are vulnerable libraries	15	0	0.6	0
#Bloated Dependencies that induce license conflicts	19	0	0.2	0
#Bloated Dependencies that induce dependency conflicts	94	0	2.8	1

declared dependencies and bloated dependencies (marking out their dependency management issues: vulnerabilities/dependency conflicts/incompatible licenses/outdated dependencies). We evaluate the usefulness of SLIMMING based on developers' feedback.

Results. Table 8 summarizes the statuses of our reported issues. Among our submitted 66 issue reports, The developers quickly confirmed 38 of them (57.6%), and 36 confirmed reports (54.5%) were later fixed or under-fixing using our debloating solutions. In 2 confirmed reports (3.0%), the developer did not merge our PRs since their projects were planning to use the bloated dependencies in their subsequent versions as the extension of functionalities. In our 4 reported issues (6.1%), developers said they did not care about bloated dependencies. The other 24 reports (36.4%) were still pending, likely due to the less active project maintenance. In our 59 issue reports, removing bloated dependencies can also resolves: 20 vulnerabilities, 8 license incompatibilities, 11 dependency conflicts, and 243 outdated dependencies.

Feedback on debloating solutions derived by SLIMMING. We discuss the representative cases that reveal the usefulness and rationality of our debloating solutions in three aspects:

Table 8: Experiment results of RQ3

66 Issue Reports Submitted by SLIMMING
Jax-rs2#203; TarsJava#170; Small-spring#56; Immutables#1382; Tomee#852; Azure-iot#1543; Quickstart#388; JavaNotes#1; Lemon#210; Karate-grpc#23; Camel-kafka#1352; Linkis#1716; Ysomap#18; XChange#4200; Swagger-parser#1590; Vertx#429; Mycat#694; Yuzhouwan#343; Openapi-gen#9768; Jax#36; Nerdronix#3; Spring-boot#32; CombatLogX#191; Tutorials#13568; Jetcache#565; Wso2-syn#1771; Tesseract4java#61; Openimaj#317; Azure#19070; QuickBooks#158; Dubbo#278; Deeplearning4j#9190; Springboot#34; Alldata#50; Tez#127; Amazon#135; OSM2World#174; Kafka#83; Skywalking#172; Jmeter#523; Java-sec#75; Jeecg#4819; Ice4j#258; DataSphereStudio#584; Notebook#24; Shardingsphere#153; Scouter#925; Myblog#8; Yshopmall#23; X-SpringBoot#31; Istack#57; Pay-java#92; Kontraktor#119; Jprotobuf#84; Spring-amqp#65; Milton2#176; Ninja#746; Protege#1051; Aws#3192; JacORB#299; Mall4j#18; Asterisk#383; Msf4j#582; DataX#973; Karamel#206; Ruoyi#36


- ▲: Confirmed issues were fixed with our debloating solutions;
- : Confirmed issues were under fixing with our debloating solutions;
- : Confirmed issues but developers plan to use bloated dependencies in future;
- ★: Rejected issues in which developers did not care about bloated dependencies;
- ▲: Pending issues; **Removing the bloated dependencies can also resolve:** vulnerabilities, license incompatibilities, dependency conflicts, outdated dependencies. Detailed info is on <https://slimming-fat.github.io/>

- The confirmed reports cover many popular projects. The confirmed reports involve many popular projects, such as Apache/Linkis #1716 [1] (3.1k Stars), and Apache/Tomee#852 [3] (419 Stars).
- Many projects were willing to remove bloated libraries to streamline software dependencies. For example, SLIMMING identified 32 bloated dependencies in Linkis#1716 [1], accounting for 78% of the project's third-party libraries. Especially, two dependencies are invoked by client code through Java reflections. Developers adopted our debloating suggestions since our PR passed all the tests. In report Tez#127 [2], the bloated dependency `org.codehaus.jettison:jettison:1.3.4` contains vulnerability #CVE-2022-45693. Although removing this bloated library at the cost of additionally declaring its transitive dependency `stax:stax-api:1.0.1` as project Tez's direct dependency, developers quickly merged the PR submitted by SLIMMING.
- Developers showed great interest in our SLIMMING tool. For example, one developer left a comment in report quickstart#388 [16]: "Thanks very much for the contribution, extremely useful. What tool did you use to do the analysis?" A developer appreciated our PR for mapstruct#134 [13] and commented that "Can you please tell us how you did your analysis? It seems to me like this PR has been created in an automated way."

Impacts of debloating solutions on dependency management.

By inspecting the code repositories of 38 projects that confirmed our submitted issues, we found 60 commit records submitted by developers for upgrading the bloated dependencies and 20 vulnerability warnings from *GitHub Advisory* against the bloated dependencies. For example, in report Linkis#1716 [1], project developers updated ten versions for bloated dependencies `commons-io:commons-io`, `com.google.code.gson:gson` and `commons-net:commons-net` in the past three years and continually received high-severity vulnerability warnings (#CVE-2021-29425, #CVE-2022-25647 and #CVE-2021-37533) against this library from build tools. The results indicate that debloating solutions derived by SLIMMING help

developers remove unused dependencies and reduce unnecessary maintenance costs.

 **Conclusion:** SLIMMING reported 484 bloated dependencies to 66 open-source projects. 38 reports (57.6%) were confirmed by developers. The feedback indicates the usefulness and rationality of our debloating solutions.

6 DISCUSSIONS

Limitations. (1) For static analysis, SLIMMING resolves the dynamic behaviors of reflective calls leveraged by six popular Java frameworks (*Java EE Servlets*, *Java Beans*, *Apache Struts*, *Spring*, *Spring Boot* and *Tomcat*), rather than all the existing frameworks, which may lead to few false alarms for bloated dependencies; (2) Before reporting the identified bloated dependencies, SLIMMING triggers projects' bundled tests to validate their debloating solutions. In practice, such a validation heavily depends on the test coverage ratio of given projects, which cannot guarantee the reliability of debloating solutions.

Threats To Validity. One possible threat is the representativeness of our study Java projects. In our experiments, we only consider the open-source Java projects, not cover the industrial projects. To minimize the threat, we conducted a large-scale study in the Maven ecosystem to collect the projects in various scales and complexity. Another threat comes from evaluating SLIMMING's usefulness. We rely on developers' feedback to validate our submitted bug reports. In general, there may be disagreement between the developers on the validity of the bug reports. However, we did not encounter such disagreement for all the evaluated subjects. Therefore, the positive feedback is a strong indication of the usefulness of our approach.

7 RELATED WORK

7.1 Java Reflection Analysis

There are two directions of research on Java reflection analysis: *Static Reflection Analysis* [19, 22, 28–31, 33, 39, 47] and *Dynamic Reflection Analysis* [21, 25]. Livshits et al. [31] introduce the first static reflection analysis for Java. By interleaving with a pointer analysis, this reflection analysis discovers constant string values by performing regular string inference and infers the types of reflectively created objects. Many modern pointer analysis frameworks such as Doop [22], Wala [19], and Chord [33], adopt a similar approach to analyze Java reflection statically, and many subsequent reflection analyses [28–30, 39, 47], are also inspired by the same work. Hirzel et al. [25] propose an online pointer analysis for handling some dynamic features in Java at runtime. To tackle reflection, their analysis instruments a program to generate constraints dynamically when the injected code is triggered at run time. Sridharan et al. [43] and Antoniadis et al. [20] proposed approaches that exploit domain knowledge to automatically generate a specification of framework-related behaviors (e.g., reflection usage) by processing both application code and configuration files.

In addition to a pointer analysis for resolving reflection targets, our SLIMMING leverages a scalable approach to identify the specifications of framework-related reflection usage patterns (including *Spring Boot*, *Spring*, *Tomcat*, *Java EE Servlets*, *Java Beans* and *Apache Struts*) in a systematic manner. It outperforms the existing reflection analysis approaches in the metrics of both *Precision* and *Recall*.

7.2 Software Debloating

In the past few years, several fine-grained debloating techniques have been proposed for reducing binary code size for efficient installations and executions on embedded hardware with limited memory. These fine-grained debloating techniques are designed for various domains, including JavaScript programs [35, 44], Java applications [23, 27, 32, 41], application containers (e.g., docker) [37], or native C programs [24, 34, 36, 38]. These range from static analysis [23, 27, 32, 38, 44] to load/runtime techniques [34, 36, 37, 41] and machine learning [24]. For promoting dependency management, Soto-Valero proposed two coarse-grained debloating approaches [40, 42] at library-granularity, which are the most relevant research to SLIMMING. They first developed a tool DEPCLEAN [42], which constructs the complete dependency tree of a project and statically analyzes the bytecode to determine the presence of bloated dependencies. Leveraging DEPCLEAN, they conducted an empirical study [40] to learn the evolution and impact of bloated dependencies in the Maven software ecosystem.

However, all the existing debloating techniques that rely on static analysis do not consider dynamic language features. The effectiveness of prior debloating techniques that use dynamic analysis approaches heavily depends on the test coverage ratio of the given projects. *This unsoundness makes debloating unreliable: removing dynamically invoked code and inducing subsequent runtime errors.* In addition, the coarse-grained debloating tool DEPCLEAN removes the unused libraries without considering its impacts on a project's dependency graph structure. *Such solutions make debloating unreasonable: removing libraries may induce new issues that are not conducive to dependency management.* In this paper, we address these limitations, propose an effective technique to resolve Java reflections, and systematically analyze the library dependency graph to remove the bloated dependencies reliably.

8 CONCLUSION AND FUTURE WORK

In this paper, we developed an effective technique, SLIMMING, to help developers reliably remove bloated dependencies from Java projects. Practical static reflection analyses empower our tool and drive debloating solutions by weighing the benefits against the costs of dependency management. The evaluation results indicate the effectiveness of SLIMMING in deriving debloating solutions. In the future, we plan to improve the static analysis capability and generalize SLIMMING to other programming ecosystems to help more developers streamline software dependencies.

ACKNOWLEDGMENTS

The authors express thanks to the anonymous reviewers for their constructive comments. We thank Professor Tian Tan for his guidance in Java reflection analysis technique. The work is supported by the National Natural Science Foundation of China (Grant No. 62141210), the Fundamental Research Funds for the Central Universities (Grant No. N2217005), Open Fund of State Key Lab. for Novel Software Technology, Nanjing University (KFKT2021B01), and 111 Project (B16009).

REFERENCES

- [1] 2023. Apache/Linkis#1716. <https://github.com/apache/linkis/pull/1716>. Accessed: 2023-03-26.
- [2] 2023. Apache/Tez#127. <https://github.com/apache/tez/pull/127>. Accessed: 2023-03-26.
- [3] 2023. Apache/Tomee#852. <https://github.com/apache/tomee/pull/852>. Accessed: 2023-03-26.
- [4] 2023. ASM. <https://asm.ow2.io/>. Accessed: 2023-03-26.
- [5] 2023. DaCaPo (2006-10-MR2). <https://sourceforge.net/projects/dacapobench/files/archive/2006-10-MR2/dacapo-2006-10-MR2.jar>. Accessed: 2023-03-26.
- [6] 2023. Doop. <https://github.com/plast-lab/doop-mirror>. Accessed: 2023-03-26.
- [7] 2023. Free Software Foundation website. <http://www.gnu.org/licenses/license-list.en.html>. Accessed: 2023-03-26.
- [8] 2023. GitHub Advisory DB. <https://github.com/advisories>. Accessed: 2023-03-26.
- [9] 2023. Issue jax-rs2-guide#203. <https://github.com/feuyeux/jax-rs2-guide-11/pull/203>. Accessed: 2023-03-26.
- [10] 2023. Issue spring-amqp#65. <https://github.com/spring-projects/spring-amqp-samples/pull/65>. Accessed: 2023-03-26.
- [11] 2023. JaCoCo. <https://www.eclemma.org/jacoco/trunk/doc/counters.html>. Accessed: 2023-03-26.
- [12] 2023. Logisland. <https://github.com/Hurence/logisland>. Accessed: 2023-01-24.
- [13] 2023. mapstruct#134. <https://github.com/mapstruct/mapstruct-examples/pull/134>. Accessed: 2023-03-26.
- [14] 2023. Maven Dependency Analyzer. <https://github.com/apache/maven-dependency-analyzer>. Accessed: 2023-03-26.
- [15] 2023. maven-dependency-tree. <https://maven.apache.org/plugins/maven-dependency-plugin/tree-mojo.html>. Accessed: 2023-03-26.
- [16] 2023. quickstart#388. <https://github.com/jbosstm/quickstart/pull/388>. Accessed: 2023-03-26.
- [17] 2023. Snyk Vulnerability DB. <https://security.snyk.io/vuln/maven>. Accessed: 2023-03-26.
- [18] 2023. Soot. <http://soot-oss.github.io/soot/>. Accessed: 2023-03-26.
- [19] 2023. WALA UserGuide: PointerAnalysis. https://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis#Contexts_for_Reflection. Accessed: 2023-03-26.
- [20] Anastasios Antoniadis, Nikos Filippakis, Paddy Krishnan, Raghavendra Ramesh, Nicholas Allen, and Yannis Smaragdakis. 2020. Static analysis of Java enterprise applications: frameworks and caches, the elephants in the room. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 794–807.
- [21] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*. 241–250.
- [22] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly declarative specification of sophisticated points-to analyses. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*. 243–262.
- [23] Bobby R Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim. 2020. Jshrink: In-depth investigation into debloating modern java applications. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 135–146.
- [24] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. 2018. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 380–394.
- [25] Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. 2007. Fast online pointer analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 29, 2 (2007), 11–es.
- [26] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022), 90.
- [27] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. Jred: Program customization and bloatware mitigation based on static analysis. In *2016 IEEE 40th annual computer software and applications conference (COMPSAC)*, Vol. 1. IEEE, 12–21.
- [28] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing reflection resolution for Java. In *ECOOP 2014—Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28–August 1, 2014. Proceedings* 28. Springer, 27–53.
- [29] Yue Li, Tian Tan, and Jingling Xue. 2015. Effective soundness-guided reflection analysis. In *Static Analysis: 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9–11, 2015, Proceedings* 22. Springer, 162–180.
- [30] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and analyzing java reflection. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 28, 2 (2019), 1–50.
- [31] Benjamin Livshits, John Whaley, and Monica S Lam. 2005. Reflection analysis for Java. In *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2–5, 2005. Proceedings* 3. Springer, 139–160.
- [32] Konner Macias, Mihir Mathur, Bobby R Bruce, Tianyi Zhang, and Miryung Kim. 2020. Webjsrink: a web service for debloating java bytecode. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1665–1669.
- [33] Mayur Naik, Alex Aiken, and John Whaley. 2006. Effective static race detection for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 308–319.
- [34] Chenxiong Qian, Hong Hu, Mansour Alharthi, Simon Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *USENIX Security Symposium*. 1733–1750.
- [35] Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. 2020. Slimium: debloating the chromium browser with feature subsetting. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 461–476.
- [36] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*. 869–886.
- [37] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: automatically debloating containers. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 476–486.
- [38] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 329–339.
- [39] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Programming Languages and Systems: 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30–December 2, 2015, Proceedings* 13. Springer, 485–503.
- [40] César Soto-Valero, Thomas Durieux, and Benoit Baudry. 2021. A longitudinal analysis of bloated java dependencies. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1021–1031.
- [41] César Soto-Valero, Thomas Durieux, Nicolas Harrand, and Benoit Baudry. 2023. Coverage-based debloating for java bytecode. *ACM Transactions on Software Engineering and Methodology* 32, 2 (2023), 1–34.
- [42] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45.
- [43] Manu Sridharan, Shay Artzi, Marco Pistoia, Salvatore Guarneri, Omer Tripp, and Ryan Berg. 2011. F4F: taint analysis of framework-based web applications. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*. 1053–1068.
- [44] Hernán Ceferino Vázquez, Alexandre Bergel, Santiago Vidal, JA Díaz Pace, and Claudia Marcos. 2019. Slimming javascript applications: An approach for removing unused functions from javascript libraries. *Information and software technology* 107 (2019), 18–29.
- [45] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 26th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 319–330.
- [46] Rongxin Wu, Hongyu Zhang, Sunghun Kim, and Shing-Chi Cheung. 2011. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 15–25.
- [47] Yifei Zhang, Tian Tan, Yue Li, and Jingling Xue. 2017. Ripple: Reflection analysis for android apps in incomplete information environments. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. 281–288.