A Soundness and Precision Benchmark for Java Debloating Tools

Jonas Klauke
Paderborn University
Paderborn, Germany
jonas.klauke@upb.de

Tom Ohlmer
Paderborn University
Paderborn, Germany
tohlmer@mail.upb.de

Stefan Schott
Paderborn University
Paderborn, Germany
stefan.schott@upb.de

Serena Elisa Ponta
SAP Labs
Mougins, France
serena.ponta@sap.com

Wolfram Fischer
SAP SE
Stuttgart, Germany
wolfram.fischer@sap.com

Eric Bodden
Paderborn University &
Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

Abstract

Modern software development reuses code by importing libraries as dependencies. Software projects typically include an average of 36 dependencies, with 80% being transitive, meaning they are dependencies of dependencies. Recent research indicates that only 24.9% of these dependencies are required at runtime, and even within those, many program constructs remain unused, adding unnecessary code to the project. This has led to the development of debloating tools that remove unnecessary dependencies and program constructs while balancing precision by eliminating unused constructs and soundness by preserving all required constructs. To systematically evaluate this trade-off, we developed Deblometer, a micro-benchmark consisting of 59 test cases designed to assess support for various Java language features in debloating tools. Each test case includes a manually curated ground truth specifying necessary and bloated classes, methods, and fields, enabling precise measurement of soundness and precision. Using Deblometer, we evaluated three popular Java debloating tools: Deptrim, JShrink, and ProGuard. Our evaluation reveals that all tools remove required program constructs, which results in changed semantics or execution crashes. In particular, the dynamic class loading feature introduces unsoundness in all evaluated tools. Our comparison shows that Deptrim retains more bloated constructs, while ProGuard removes more required constructs. JShrink's soundness is significantly affected by limited support for annotations, which leads to corrupted debloated artifacts. These soundness issues highlight the need to improve debloating tools to ensure stable and reliable debloated software.

CCS Concepts

• **Software and its engineering** → **Functionality**; **Completeness**; **Software reliability**; • **General and reference** → **Validation**; **Reliability**.

Keywords

Debloating, Micro-benchmark, Soundness, Precision

1 Introduction

The software supply chain consists of all processes and components involved in the development, building, and delivery of software. One part of it is the management of third-party libraries. This is typically handled by package managers like Maven, npm, or PIP, which automatically download and integrate the dependencies of a modern software project. These dependencies are either directly added by the developers, inherited from a parent project, or transitively added by other dependencies. There are, on average, 36 dependencies in an industry software project, with 80% being transitive [7]. However, Soto et al. [26] showed that 75.1% (2.7% direct, 15.4% inherited, 57% transitive) are bloated and therefore, they are not executed in the project. As a result, a large portion of dependencies in the software supply chain are maintained and updated without impacting the project itself. Many researchers [2, 9, 14, 22, 32, 33] demonstrated that debloating successfully decreases the attack surface by removing vulnerabilities in the project dependencies. However, if code required at runtime is incorrectly debloated (i.e., removed), it will cause execution failures [4, 18] or it may even lead to new vulnerabilities introduced by changed behavior during execution [5]. This raises the demand for debloating tools that are *sound*, ensuring all code required at runtime is preserved, and *precise*, effectively removing all bloated code.

Debloating can be performed at different granularity levels. One broader granularity is the complete removal of entire dependencies from the project; however, a more fine-grained form of debloating is the disposal of program constructs inside the dependencies. Typical program constructs of Java dependencies are classes, methods, or fields. These are also considered bloated if they are not accessed during the execution of a given software project. For example, this could be caused by a dependency that provides capabilities to manage the file system, while the project only utilizes it to read files. In this case, all classes, methods, and fields related to writing files become bloated program constructs from the perspective of the project. Ponta et al. [23] have mapped vulnerabilities to fix commits of open-source dependencies to pinpoint them to the actual program constructs, revealing that the vulnerability does not affect the entire dependency. By removing these program constructs, the attack surface of the project is further reduced. Bloated program constructs also pose a threat even if they are not executed, since they can be exploited by deserialization attacks [21] that misuse the mechanism to execute them.

In this paper, we investigate the soundness and precision of debloating tools that are capable of removing bloated program constructs in Java. We have created a manually curated micro-benchmark called Deblometer where we exactly know which program constructs are required and which are bloated. Similar to the micro-benchmark for call graphs developed by Reif et al [25], we evaluate the soundness and precision in Deblometer on encapsulated Java language features to directly spot the shortcomings of the debloating tools. To find Java language features of interest, we have investigated current literature about debloating, established micro-benchmarks in similar problem areas and the Java language feature documentation resulting in 59 test cases highlighting the usage of 13 Java language features. Each test case is evaluated on the correct removal of the following program constructs: classes, interfaces, methods, and fields. The set of needed and unneeded program constructs is defined by a manually created ground truth for each evaluated language feature. We evaluated Deptrim [27], JShrink [6], and ProGuard [10] on Deblometer. All tools support the removal of the evaluated program constructs.

Our evaluation suggests that Deptrim tends to remove fewer program constructs, which results in a higher soundness score but a lower precision score. In contrast, ProGuard favors more aggressive debloating, resulting in a lower soundness score but a higher precision score. The investigation of JShrink also reveals soundness and precision issues, especially due to its lack of support for Java annotations, which causes JShrink to produce corrupted JAR files that can no longer be executed. All tools struggle with sound resolution of the dynamic class loading language feature, while JShrink and ProGuard also show no support for reflection language features. The execution of unsound debloated test cases revealed crashes and altered behavior during execution. This highlights the need for improvement in debloating tools to produce stable debloated dependencies without causing crashes or modifying program semantics.

This paper consists of the following contributions:

- A micro-benchmark [1] for Java debloating tools that remove bloated classes, methods, and fields. It consists of 59 test cases with manually curated ground truths categorized by 13 different language features.
- A systematic evaluation [2] of the soundness and precision of three popular Java debloating tools, Deptrim, JShrink, and ProGuard, focused on the resolving of encapsulated Java language features.

2 Approach

An overview of Deblometer is shown in Figure 1. It consists of the test cases and a validation harness. Each test case is defined by a bloated JAR and its corresponding ground truth. The debloating tool takes the bloated JARs as input, processes them, and produces the debloated Jars as output. Deblometer evaluates these debloated JARs against their corresponding ground truths in the test cases within the validation harness. The output of Deblometer includes the soundness and precision scores of the debloating tool for each test case.

[1] https://github.com/secure-software-engineering/Deblometer
[2] https://zenodo.org/records/16994277

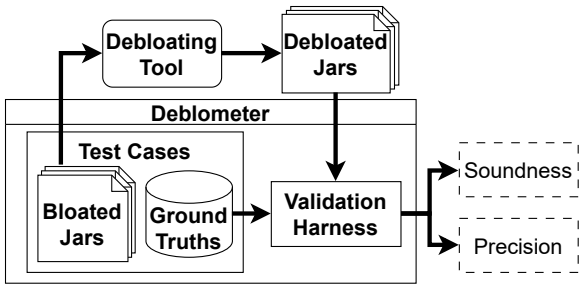

Figure 1: Overview of Deblometer

2.1 Test Case Creation

The test cases assess the soundness and precision of the evaluated debloating tools in removing bloated program constructs. Given the scope of our micro-benchmark, we categorized the test cases by language features to investigate the level of language feature support provided by the debloating tool. Each test case involves a usage of one specific language feature. It is specifically designed to use only that feature, allowing the soundness and precision results to be directly mapped to the corresponding language feature. However, because some feature usages depend on the presence of another feature, not all test cases exclusively use a single language feature. For example, Java supports functional interfaces, which combine lambdas and interfaces. To evaluate this usage, we designed a test case categorized under interfaces, but it is also influenced by the lambda feature. Since the test cases are used to evaluate debloating tools, they include additional classes, methods, and fields that are not involved in the usage of the targeted feature. To construct a representative list of relevant language features for categorizing our test cases, we reviewed existing literature on debloating [6, 22] and static analysis [15, 24, 25, 30], as many debloating tools rely on static analysis to process JAR files. We also examined the Java language feature documentation [13] to identify features not discussed in the current literature. A detailed overview of the selected language features is provided in Section 3. For each language feature, we created a separate JAR file containing all corresponding test cases, including both required and bloated classes, methods, and fields. These components are connected within the JAR by a main class that invokes all test cases. This main class serves as the entry point for the debloating process. It ensures that the debloating tools do not remove all program constructs in the bloated JAR simply because they are not reachable from the entry point.

2.2 Debloating Levels

Deblometer consists of three debloating levels: class, method, and field. The class level also includes interfaces and annotation definitions, as these are compiled into class files that are part of the bloated JAR. Every test case contains program constructs from these three debloating levels, which may or may not be used during execution. However, for certain language features, one or more types of program constructs may be excluded if they are not relevant to the specific feature being tested. For example, in the case of method overloading, where multiple methods in a class can share

the same name as long as their parameter lists differ, the field level is not considered because it is not related to the feature.

Listing 1 presents an example test case from Deblometer. This test case is categorized under the abstract class language feature. In this example, the feature is exercised through the use of an anonymous class. The program constructs for each debloating level are indicated by comments in the code. Examples of these constructs include the abstract class `Car` in Line 3 for the class level, the integer field `piston` in Line 4 for the field level, and the abstract method `engine()` in Line 6 for the method level. Within the `main` method, an anonymous class is instantiated with two overriding methods. Only the `engine` method is invoked in Line 21. This method also accesses the `piston` field from Line 4. As a result, the `material` method in Line 18 and the `material` field in Line 5 are considered bloated because they are never used during program execution and can be removed.

```
1  package Abstract;
2
3  abstract class Car { //class level
4    int piston = 4; //field level
5    String material = "aluminium"; //field level
6    public abstract void engine(); //method level
7    public abstract void material(); //method level
8  }
9
10 public class Main { //class level
11   public static void main(String[] args) {
12     Car obj = new Car() { //class level
13       @Override
14       public void engine() { //method level
15         System.out.println("No. of pistons in engine " + piston);
16       }
17       @Override
18       public void material() { //method level
19         System.out.println("Engine's material " + material);
20     };
21     obj.engine();
22   }
23 }
```

Listing 1: Anonymous class implementation within the abstract module.

2.3 Ground Truth

For the ground truth specification, a JSON format is used to ensure that the information is structured and accessible during the validation process. The JSON structure is organized according to the three debloating levels defined in the previous section: class, method, and field. Each level is further divided into "required" and "bloated" sections.

In the class section, each class is identified by its package and class name. In the method section, each entry includes the identifier of the declaring class, the method name, the return type, and the parameter list. In the field section, each entry specifies the declaring class and the field name.

The example shown in Listing 2 represents the ground truth for the code sample in Listing 1. The class level begins at Line 2. All classes in the test case are executed, so they are listed in the "required" section from Lines 4 to 6, while the "bloated" section in Line 8 remains empty. The `Car` class is specified in Line 5 using the class name `Car` and the package name `Abstract`. The method level begins at Line 10. Only two methods are required in the test case, as only `main()` and `engine()` are invoked. The `engine()` method is described in Line 12 with the class name `Main$1`, the method name `engine`, the return type `void`, and an empty parameter list. The other `engine()` method and the `material()` method are not executed, and are therefore considered bloated. They are listed in the "bloated" section from Lines 16 to 18. The field level begins at Line 21. The test case includes one required field and one bloated field, since only one field is accessed during execution. The required field is defined in Line 23 with the class name `Car` and the field name `piston`. If any of the required program constructs are removed during debloating, the resulting application either crashes due to incomplete code or behaves incorrectly. For example, if a method that overrides a superclass method is removed, the execution changes because Java will call the method from the superclass instead. This happens because Java uses dynamic method dispatch, and without the overriding method, the method call is resolved to the one defined in the superclass.

```
1  {
2   "CLASS": {
3    "required": [
4      {"package": "Abstract", "name": "Main"},
5      {"package": "Abstract", "name": "Car"},
6      {"package": "Abstract", "name": "Main$1"}
7     ],
8    "bloated": []
9   },
10  "METHOD": {
11   "required": [
12     {"type": "Main", "name": "main", "return": "void", "param":
           "String[]"},
13     {"type": "Main$1", "name": "engine", "return": "void", "
          param": ""}
14    ],
15   "bloated": [
16     {"type": "Car", "name": "engine", "return": "void", "param"
          : ""},
17     {"type": "Car", "name": "material", "return": "void", "
          param": ""},
18     {"type": "Main$1", "name": "material", "return": "void", "
          param": ""}
19    ]
20  },
21  "FIELD": {
22   "required": [
23     {"class": "Car", "name": "piston"}
24    ],
25   "bloated": [
26     {"class": "Car", "name": "material"}
27    ]
28  }
29 }
```

Listing 2: JSON representation of required and bloated elements on class, method, and field level.

2.4 Validation Harness

The validation process analyzes the debloated JAR files produced by each debloating tool when applied to the bloated JARs from the Deblometer benchmark. The remaining classes, methods, and fields in the debloated JAR are compared to the ground truth of the corresponding test case. Deblometer uses SootUp [17], a static analysis framework that provides functionality to list all classes along with their declared fields and methods, to extract these program constructs. The extracted elements are then compared to the required and bloated entries specified in the ground truth.

Deblometer identifies true positives, false positives, and false negatives at each debloating level in order to compute the soundness and precision for the class, method, and field levels. A true positive refers to a program construct that is retained in the debloated JAR and correctly listed in the required section of the ground truth. A false positive is a bloated construct that remains in the debloated JAR, even though it should have been removed. A false negative occurs when a required construct is missing from the debloated JAR, which can lead to execution failures.

Soundness is calculated using the following formula:

$$\text{Soundness} = \frac{TP}{TP + FN}$$

Precision is calculated as:

$$\text{Precision} = \frac{TP}{TP + FP}$$

The evaluation results for each Java language feature are presented in tabular format, showing the soundness and precision at each of the three debloating levels, grouped by the language feature evaluated by the validation harness.

3 Tests

In the following, we present the Java language features evaluated in Deblometer. The process used to identify these features is described in Section 2.1. For each selected feature, we created multiple test cases to capture different usages of the evaluated language feature. These test cases include both required and intentionally bloated program constructs. The expected outcome of each test case is formally specified in the corresponding ground truth JSON files, as explained in Section 2.3. These ground truths serve as the basis for evaluating the soundness and precision of the debloating tools, as detailed in Section 2.4. Table 1 shows the number of test cases associated with each evaluated language feature. In total, Deblometer contains 59 test cases categorized across 13 different Java language features. All test cases categorized under a specific language feature are combined into a single bloated JAR representing that feature.

Table 1: Number of test cases per Java language feature. A language feature is only considered evaluated in related work if it was explicitly investigated in that work.

Language Feature	Evaluated in Related Work	Test Cases
Abstract		6
Annotation		7
Deserialization	[24, 25, 30]	2
Dynamic Class Loading	[24, 25, 30]	2
Exception		4
Externalization	[24, 25]	2
Generics	[15]	7
Interface	[24, 25]	4
Lambda	[15, 24, 25, 30]	4
Overloading	[24, 25]	6
Overriding	[15, 24, 25]	4
Reflection	[15, 24, 25, 30]	6
Serialization	[24, 25, 30]	5
		59

3.1 Abstract Classes

Abstract classes are a fundamental feature of object-oriented programming in Java, serving as templates for subclasses and enabling polymorphism. To evaluate this language feature, we designed test cases based on various implementation techniques, including the use of anonymous classes, non-abstract subclasses, explicit method implementations, and scenarios with no usage of the abstract class. Additional test cases include partial implementations that combine abstract and concrete logic, as well as unrelated method invocations not defined in the abstract class.

3.2 Annotations

Annotations are used to provide metadata that can influence program behavior at compile time, runtime, or during tooling processes. To evaluate this feature, we designed test cases based on various implementation techniques, including the conditional presence of annotations on classes, fields, and methods; the use of functional annotations that affect program execution; and the identification of redundant annotations that do not impact program semantics. Custom annotations were employed for this purpose, as standard Java annotations, while accessible, do not meet the specific requirements needed to support the analysis and removal of bloated code by debloating tools.

3.3 Deserialization

Deserialization enables the reconstruction of objects from a byte stream and plays a crucial role in data exchange and persistence. To evaluate this language feature, we designed test cases based on relevant implementation techniques, including standard deserialization using built-in mechanisms and method overriding during deserialization. We did not specifically test class-level debloating for this feature, as standard class reachability already applies, making such targeted testing redundant and unlikely to yield additional insights. However, classes involved in deserialization are still covered in our evaluation through general usage scenarios, even if not tested through dedicated implementations.

3.4 Dynamic Class Loading

Dynamic class loading allows classes to be loaded at runtime, enabling greater extensibility and modularity. To evaluate this language feature, we created test cases based on relevant implementation techniques, including dynamic instantiation using a class loader and reflective instantiation based on a specified class name.

3.5 Exception

Exception handling is used to manage errors and unexpected conditions during program execution. To evaluate this language feature, we developed test cases based on different implementation techniques, including a basic custom exception, an extended custom

exception with additional fields, a hierarchical structure of custom exceptions, and an implementation of an unchecked exception.

3.6 Externalization

Externalization provides developers with full control over the serialization process by allowing them to explicitly define how objects are written to and read from a stream. This requires implementing the `readExternal` and `writeExternal` methods from the Externalizable interface. To evaluate this language feature, we created test cases based on specific implementation techniques: one focused solely on reading and another solely on writing.

3.7 Generics

Generics enable type-safe programming by allowing classes, interfaces, and methods to operate on typed parameters. They enhance code reusability and reduce the need for explicit type casting. To evaluate this language feature, we developed test cases based on various implementation techniques, including basic generic type definitions, generic inheritance, generic interfaces, interface composition involving generics, method overloading with generic parameters, the use of parameterized arrays, and wildcard types for flexible type bounds.

3.8 Interface

Interfaces are a fundamental construct used to define expected behaviors that classes can implement, promoting abstraction and multiple inheritance of type. To evaluate this language feature, we created test cases based on various implementation techniques, including standard interface implementation, the use of anonymous classes, interface composition to combine multiple behaviors, and lambda expressions representing functional interface implementations.

3.9 Lambda Functions

Lambdas are a language feature that enable a concise representation of functional behavior, allowing functions to be treated as first-class citizens. To evaluate this feature, we designed test cases based on relevant implementation techniques, including lambda expressions that interact with classes, modify fields within their scope, and implement logic directly within methods.

3.10 Overloading

Overloading allows multiple methods with the same name to coexist within a class, distinguished by their parameter lists. To evaluate this language feature, we designed test cases covering various implementation techniques, including methods with different numbers of arguments, compile-time binding resolution, variations in parameter types, swapped parameter order, method selection priority, and type promotion. Class-level debloating was not specifically tested for this feature, as standard class reachability applies, making targeted testing redundant. However, classes are implicitly covered through general program behavior, even though specific class implementations were not individually evaluated. Additionally, since overloading applies only to methods, testing fields was not necessary.

3.11 Overriding

Overriding allows a subclass to provide a specific implementation of a method defined in its superclass, enabling dynamic dispatch and runtime polymorphism. To evaluate this language feature, we designed test cases based on relevant implementation techniques, including standard method overriding, polymorphic behavior through superclass references, and method hiding where static methods in a subclass obscure rather than override those in the superclass. Field-level testing was not performed because overriding applies strictly to methods and does not affect fields.

3.12 Reflection

Reflection enables inspection and manipulation of classes, methods, and fields at runtime, allowing dynamic behavior and introspection. To evaluate this language feature, we created test cases based on various implementation techniques, including access control checks using name equality, retrieval of enum-defined fields and methods, inheritance-aware introspection, member retrieval, method inspection via enums, and standard reflection involving iteration over fields and methods.

3.13 Serialization

Serialization enables objects to be converted into a byte stream for storage or transmission, preserving their state for later reconstruction through deserialization. To evaluate this language feature, we designed test cases based on relevant implementation techniques, including deep serialization of classes containing other serializable objects, serialization through inheritance hierarchies, and standard serialization scenarios where methods or logic without state are considered obsolete, as they cannot be meaningfully serialized.

4 Evaluation

In this section, we evaluate the behavior and limitations of debloating tools, measuring their soundness and precision using Deblometer. Because our focus is on Java and the removal of program constructs, we selected tools that can debloat Java applications by removing classes, methods, and fields. To identify suitable tools, we surveyed available Java debloating solutions that meet these criteria and are publicly accessible. Deblometer evaluates the following tools[3]:

- Deptrim (v0.1.2)
- JShrink[4] (retrieved on June 2, 2025)
- ProGuard (v7.7)

The selected tools use either static analysis, dynamic analysis, or a combination of both to identify all required program constructs. ProGuard relies exclusively on static analysis during the debloating process, whereas Deptrim and JShrink combine static and dynamic analysis to address the known limitations of static analysis reported in related work [24, 25, 30]. However, the default configuration of JShrink, as suggested by its authors, relies solely on static analysis for debloating.

[3]We initially included J-Reduce [16] in our evaluation. However, it consistently produced empty JAR files for our test cases, so we excluded it from the final analysis.
[4]https://github.com/UCLA-SEAL/JShrink

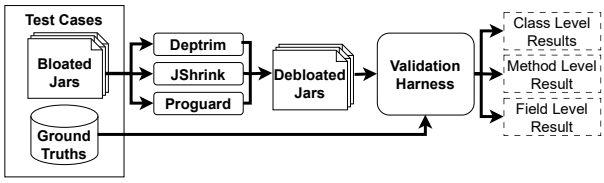

Figure 2: The evaluation process using Deblometer

The evaluation process for the selected tools is illustrated in Figure 2. It begins by running the debloating tools on the bloated JARs provided by the test cases in Deblometer. Each tool generates debloated JARs from the input JARs, which are then validated against the corresponding ground truths using Deblometer 's validation harness to calculate soundness and precision scores for each language feature and debloating level. To ensure accurate validation, all obfuscation features were disabled, preserving original class, method, and field names. Since Deptrim does not operate directly on JAR files but rather on project dependencies, we created a wrapper project that includes all Deblometer bloated JARs as dependencies, along with a main class that invokes the main methods of each dependency. If Deptrim does not remove any program constructs and therefore produces no debloated JAR, we treat the original bloated JAR as its result. When Deblometer identifies a language feature as unsoundly supported, we execute the corresponding debloated JAR to verify whether it remains functional.

Table 2 presents the results of our experiments. It shows the soundness and precision results for each language feature, divided by the debloating levels of class, method, and field for the three debloating tools: Deptrim, JShrink, and ProGuard. The table shows the grouped results of all test cases categorized under each specific language feature. The column R lists the number of required classes, methods, or fields contained in the debloated JAR. For example, 4/6 means that 4 of the required program constructs are present in the debloated JAR, while 2 are missing. The column B lists the number of removed bloated classes, methods, or fields in the debloated JAR. For example, 7/20 means that 7 of the bloated constructs were removed, while 13 remain in the debloated JAR. The column S shows the soundness score as a percentage, and the column P shows the precision score as a percentage.

Regarding Table 2, all three tools (Deptrim, JShrink, and ProGuard) demonstrate generally high soundness and precision at the class level. Among them, Deptrim consistently yields the best overall results and performs especially well in cases involving annotations and generics, where JShrink and ProGuard show minor deficiencies. In addition, JShrink produces corrupted JARs for the annotation, lambda, and interface language features, which causes Deblometer to crash because the class files cannot be parsed by SootUp. A manual inspection revealed that the corruption was caused by an incomplete removal of annotations. JShrink removed the annotation from the constant pool of the class file, but references to deleted constant pool values remained. As a result, the corrupted JARs could not be executed or parsed by Deblometer. To enable evaluation, we removed the corrupted class files from the affected debloated JARs. The program constructs in these files were manually reviewed and their results were incorporated into the evaluation by Deblometer. The missing support for annotations also affected the lambda and interface test cases. This is because lambda classes include annotations in the class file, and one of the interface test cases makes use of lambdas as well. It is important to note that not all language feature tests were specifically designed to evaluate class-level debloating. In some cases, the tests rely only on basic class detection rather than class-specific usage of the language feature. This can lead to inflated scores that do not fully reflect the actual debloating capabilities of the tools. Despite these limitations, Deptrim proves to be the most robust and reliable tool for class-level analysis. ProGuard performs well and ranks just behind, while JShrink falls short due to its incomplete annotation handling in certain test cases.

Regarding Table 2, all three tools achieve generally high soundness scores at the method level. This indicates that relevant methods are consistently preserved during the debloating process. However, the precision scores across most language features reveal a significant shortcoming: the tools often retain a substantial amount of bloated code, leading to high false positive rates. This shows that while the tools are effective at preserving required methods, they struggle to remove unnecessary ones with precision. For instance, although Deptrim maintains perfect soundness across nearly all features, its precision remains relatively low in cases such as abstract classes (28 percent) and annotations (71 percent on bloated JARs), indicating excessive code retention. In contrast, ProGuard applies a more precise debloating strategy and often outperforms Deptrim in terms of precision. However, this increased precision comes at the cost of soundness in several cases. This issue is especially evident in annotations, deserialization, and overloading, where required methods are incorrectly removed. Such behavior reflects a more aggressive and risk-prone approach, prioritizing leaner outputs over complete correctness. JShrink, by comparison, exhibits low or unstable precision across multiple features and occasionally crashes due to corrupted JARs. Overall, the results highlight that while method-level soundness is generally satisfactory, achieving high precision without compromising correctness remains a key challenge. Among the evaluated tools, ProGuard demonstrates the most aggressive trade-off, focusing on reducing code size even if it risks removing critical program elements.

Regarding Table 2, the three debloating tools show consistently high soundness scores across most language features at the field level. This indicates that they are generally effective at retaining required fields. However, precision remains a persistent issue for all tools, with relatively high false positive rates suggesting difficulty in eliminating unused or irrelevant fields. For example, although Deptrim achieves 100 percent soundness across nearly all features, its precision varies significantly. It drops to 36 percent for generics and 50 percent for features such as annotations, interfaces, and lambdas. In the interface feature, Deptrim is the only tool that maintains full soundness, while both JShrink and ProGuard fail entirely, with soundness and precision scores of 0 percent. This result underscores Deptrim's strength in preserving essential field-level declarations. Despite this, all tools fail in the dynamic class loading cases, with soundness and precision both at 0 percent. Overall, these results show that identifying and retaining necessary fields is mostly successful, especially for Deptrim. However, the

Table 2: Evaluation results at the class, method, and field levels for each language feature. The table shows the number of required program constructs (R), the soundness score (S), the number of removed bloated program constructs (B), and the precision score (P). S and P are shown as percentages and highlighted in bold if they are below 100. Method-level evaluation of serialization is omitted because there are no true positives in the ground truth. Similarly, overriding and overloading are not assessed at the field level, as the test cases do not include relevant fields. Entries marked with '–' indicate a 0/0 result, except in explicitly excluded cases.

Language Feature		Deptrim				JShrink				ProGuard			
		R	S	B	P	R	S	B	P	R	S	B	P
abstract	Class	15/15	100	5/5	100	3/15	**20**	5/5	100	15/15	100	5/5	100
	Method	5/5	100	7/20	**48**	1/5	**20**	19/20	**50**	5/5	100	19/20	**83**
	Field	2/2	100	2/3	**80**	1/2	**50**	2/3	**50**	2/2	100	2/3	**67**
annotation	Class	$24/24_b$	100_b	$0/4_b$	$\mathbf{86}_b$	$4/24_c$	$\mathbf{17}_c$	$4/4_c$	100_c	24/24	100	4/4	100
	Method	$5/5_b$	100_b	$0/2_b$	$\mathbf{71}_b$	$1/5_c$	$\mathbf{20}_c$	$2/2_c$	100_c	2/5	**40**	2/2	100
	Field	$3/3_b$	100_b	$0/3_b$	$\mathbf{50}_b$	$1/3_c$	$\mathbf{33}_c$	$2/3_c$	$\mathbf{50}_c$	3/3	100	0/3	**50**
deserialization	Class	$4/4_b$	100_b	-_b	100_b	4/4	100	-	100	4/4	100	-	100
	Method	$3/3_b$	100_b	$0/2_b$	$\mathbf{60}_b$	3/3	100	0/2	**60**	1/3	**33**	2/2	100
	Field	$4/4_b$	100_b	$0/1_b$	$\mathbf{80}_b$	4/4	100	0/1	**80**	4/4	100	0/1	**80**
dynamic class loading	Class	4/6	**67**	4/4	100	4/6	**67**	4/4	100	4/6	**67**	4/4	100
	Method	2/4	**50**	4/4	100	2/4	**50**	4/4	100	2/4	**50**	4/4	100
	Field	0/2	**0**	6/6	**0**	0/2	**0**	6/6	**0**	0/2	**0**	6/6	**0**
exception	Class	9/9	100	3/3	100	9/9	100	3/3	100	9/9	100	3/3	100
	Method	1/1	100	0/1	**50**	1/1	100	1/1	100	1/1	100	1/1	100
	Field	1/1	100	0/1	**50**	1/1	100	1/1	100	1/1	100	1/1	100
externalization	Class	4/4	100	2/2	100	4/4	100	2/2	100	4/4	100	2/2	100
	Method	2/2	100	4/6	**50**	2/2	100	4/6	**50**	2/2	100	4/6	**50**
	Field	4/4	100	4/6	**67**	4/4	100	6/6	100	4/4	100	6/6	100
generics	Class	20/20	100	3/4	**95**	20/20	100	4/4	100	18/20	**90**	4/4	100
	Method	10/10	100	1/9	**56**	10/10	100	7/9	**83**	10/10	100	7/9	**83**
	Field	4/4	100	0/7	**36**	4/4	100	7/7	100	4/4	100	7/7	100
interface	Class	14/14	100	1/1	100	$13/14_c$	$\mathbf{93}_c$	$1/1_c$	100_c	11/14	**79**	1/1	100
	Method	5/5	100	1/7	**45**	$5/5_c$	100_c	$6/7_c$	$\mathbf{83}_c$	5/5	100	6/7	83
	Field	1/1	100	0/1	**50**	$0/1_c$	$\mathbf{0}_c$	$1/1_c$	$\mathbf{0}_c$	0/1	**0**	1/1	**0**
lambda	Class	10/10	100	1/1	100	$10/10_c$	100_c	$1/1_c$	100_c	10/10	100	1/1	100
	Method	4/4	100	1/2	**80**	$4/4_c$	100_c	$2/2_c$	100_c	4/4	100	2/2	100
	Field	2/2	100	0/2	**50**	$2/2_c$	100_c	$0/2_c$	$\mathbf{50}_c$	2/2	100	0/2	**50**
overloading	Class	$14/14_b$	100_b	-_b	100_b	14/14	100	-	100	14/14	100	-	100
	Method	$8/8_b$	100_b	$0/7_b$	$\mathbf{53}_b$	8/8	100	7/7	100	7/8	**88**	7/7	100
	Field	-_b	-_b	-_b	-_b	-	-	-	-	-	-	-	-
overriding	Class	11/11	100	1/1	100	11/11	100	1/1	100	11/11	100	1/1	100
	Method	4/4	100	1/4	**57**	4/4	100	2/4	**67**	4/4	100	2/4	**67**
	Field	-	-	-	-	-	-	-	-	-	-	-	-
reflection	Class	$13/13_b$	100_b	-_b	100_b	13/13	100	-	100	13/13	100	-	100
	Method	$5/5_b$	100_b	$0/4_b$	$\mathbf{56}_b$	0/5	**0**	4/4	**0**	0/5	**0**	4/4	**0**
	Field	$6/6_b$	100_b	$0/6_b$	$\mathbf{50}_b$	3/6	**50**	3/6	**50**	3/6	**50**	3/6	**50**
serialization	Class	13/13	100	1/2	**93**	13/13	100	1/2	**93**	13/13	100	2/2	100
	Method	-	-	-	-	-	-	-	-	-	-	-	-
	Field	14/14	100	2/4	**88**	14/14	100	2/4	**88**	14/14	100	4/4	100

b: bloated JAR was evaluated

c: DEBLOMETER crashed due to corrupted bytecode; manual intervention

accurate removal of bloated field-level code remains a common and unresolved challenge for all evaluated tools.

Finally, we executed all debloated JARs that Deblometer identified as unsound. This included 1 JAR for Deptrim, 5 for JShrink, and 7 for ProGuard, each representing an unsoundly supported language feature. The JARs produced by Deptrim and JShrink resulted in exceptions that caused the programs to crash. Similarly, 3 of the 7 unsound JARs from ProGuard also crashed during execution. The remaining 4 ProGuard JARs exhibited altered runtime behavior due to Java's dynamic resolution of virtual method calls. Such behavioral changes are especially concerning, as they are difficult to detect and may introduce new vulnerabilities that do not exist in the original JAR.

5 Discussion

The main challenges that lead to unsoundness and imprecision in debloating arise from the limitations of static analysis when dealing with dynamic or implicit language features, as highlighted in related work on static analysis techniques [24, 25, 30]. Across all evaluation levels shown in the tables, one major source of unsoundness is the inability to correctly analyze reflective and dynamic behavior. At the method level, reflection presents a particularly difficult challenge. Only Deptrim achieves 100 percent soundness in these cases, thanks to its use of dynamic analysis that enhances static analysis by resolving dynamic features. This approach demonstrates that achieving high soundness often requires sacrificing precision, and achieving high precision may come at the cost of soundness. JShrink performs especially poorly when handling annotations. It produces corrupted debloated JARs that crash during execution in several test cases. In addition, it lacks support for Java annotations, which significantly limits its applicability for modern Java applications. The challenges of static analysis are further demonstrated in the dynamic class loading test cases. Here, required classes that are loaded at runtime by the classloader are incorrectly removed. This failure not only breaks class-level soundness but also obscures actual field usage, making it harder to preserve essential fields. As a result, tools such as JShrink and ProGuard fail to retain required fields, leading to broken runtime behavior and reduced soundness at both the method and field levels. These examples show that current debloating approaches tend to either retain too much code, resulting in low precision, or remove too aggressively, leading to unsoundness. This trade-off becomes particularly problematic when handling dynamic language constructs, reflection, or incomplete modeling of dependencies.

6 Threats to Validity

A potential threat to validity arises from possible inaccuracies in the manually created ground truth. To minimize this risk, three authors independently verified the ground truth to ensure its accuracy. Another threat is that the test cases are artificially constructed and may not reflect real-world scenarios. However, the test cases in Deblometer are specifically designed to clearly expose shortcomings related to individual language features. A further limitation is that the test cases might be too narrow to allow for generalization of the results. To address this, we designed multiple test cases for each language feature and applied them in different contexts. Another threat is that the set of language features evaluated in Deblometer does not cover every possible feature in Java. To mitigate this, we focused on language features known to introduce unsoundness and imprecision, as reported in related work. Another potential threat to validity lies in how the debloating tools were configured. To reduce the impact of suboptimal configurations, we contacted the developers of each tool to ensure the most appropriate settings were used for our micro-benchmark scenarios.

7 Related Work

Most debloating tools are evaluated using popular benchmarks like GNU Coreutils [8], DaCapo [3], and SPEC CPU 2006/2017 [28, 29], however, these benchmarks are not specially tailored for debloating approaches. Chisel [11] and OCCAM [19, 20] have published benchmarks [12] for their tools in which real-world applications are debloated as test cases. Compared to Deblometer, these benchmarks spot either failures or investigate the code reduction, however, they do not inform the developers about specific language features that cause issues in the debloating process. Brown et al. [4] extended the chisel benchmark with more complex real-world applications and evaluated a set of debloating tools on it. Their test cases are written in the C programming language, making them not applicable to Java debloating tools. The industrial readiness of Java debloating tools is analyzed by Ponta et al. [22], however, their scope is on the attack surface reduction of vulnerable Java dependencies, lacking insights into the soundness and precision of these tools. Alhanahnah et al. [1] surveyed the landscape of debloating literature and their performed evaluations, however, they did not compare the tools or evaluate their soundness and precision. Java language feature support within call graph construction is analyzed by Reif et al. [24, 25] and Sui et al. [30, 31] using micro-benchmarks and dynamic analysis. Their approaches however are not fully applicable for evaluating debloating tools due to their scope being limited to method level.

8 Conclusion

In this paper, we presented Deblometer, a novel micro-benchmark designed to evaluate the debloating of program constructs in bloated JARs. It consists of 59 test cases, each focused on a specific Java language feature. We used Deblometer to assess the soundness and precision of three debloating tools capable of removing classes, methods, and fields in Java: Deptrim, JShrink, and ProGuard. Our evaluation reveals clear trade-offs between soundness and precision, driven by fundamentally different tool strategies. Deptrim prioritizes soundness through conservative analysis and performs soundly on nearly all evaluated language features except for dynamic class loading. However, this approach results in imprecision across all features. ProGuard employs a more aggressive debloating strategy, achieving higher precision at the cost of soundness, particularly for complex features such as annotations and deserialization. It is also the only tool that is unsound on test cases involving generics and method overloading. JShrink exhibits limited support for abstract classes and Java annotations, attaining the lowest scores on both features. It can also produce corrupted JARs that fail to execute when the bloated JAR contains annotations. Results from Deblometer highlight that modern debloating tools

struggle significantly with dynamic language features such as reflection and dynamic class loading. Method-level reflection and field usage under dynamic class loading expose the current limitations of these tools in balancing soundness and precision. Deblometer can guide the development and improvement of debloating tools by precisely identifying soundness and precision issues related to specific Java language features. These issues can then be addressed systematically by developers

Acknowledgments

We thank Subramanya Padmaraja Setty for his help with the design of the benchmark and for assisting in the gathering of language features and test cases. This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (GZ: SFB 901/3) under the project number 160364472.

References

[1] Mohannad Alhanahnah, Yazan Boshmaf, and Ashish Gehani. 2024. SoK: Software Debloating Landscape and Future Directions. In *Proceedings of the 2024 Workshop on Forming an Ecosystem Around Software Transformation* (Salt Lake City, UT, USA) *(FEAST '24)*. Association for Computing Machinery, New York, NY, USA, 11–18. doi:10.1145/3689937.3695792

[2] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is More: Quantifying the Security Benefits of Debloating Web Applications. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1697–1714. https://www.usenix.org/conference/usenixsecurity19/presentation/azad

[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Oregon, USA) *(OOPSLA '06)*. Association for Computing Machinery, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

[4] Michael D. Brown, Adam Meily, Brian Fairservice, Akshay Sood, Jonathan Dorn, Eric Kilmer, and Ronald Eytchison. 2024. A Broad Comparative Evaluation of Software Debloating Tools. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, Philadelphia, PA, 3927–3943. https://www.usenix.org/conference/usenixsecurity24/presentation/brown

[5] Michael D. Brown and Santosh Pande. 2019. Is Less Really More? Towards Better Metrics for Measuring Security Improvements Realized Through Software Debloating. In *12th USENIX Workshop on Cyber Security Experimentation and Test (CSET 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/cset19/presentation/brown

[6] 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 135–146. doi:10.1145/3368089.3409738

[7] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. 2022. Identifying Challenges for OSS Vulnerability Scanners - A Study & Test Suite. *IEEE Transactions on Software Engineering* 48, 9 (2022), 3613–3625. doi:10.1109/TSE.2021.3101739

[8] Free Software Foundation, Inc. 2016. Coreutils - GNU core utilities. https://www.gnu.org/software/coreutils/ (Accessed:2025-06-28).

[9] Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. 2022. C2C: Fine-grained Configuration-driven System Call Filtering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 1243–1257. doi:10.1145/3548606.3559366

[10] Guardsquare nv. 2025. Java Obfuscator and Android App Optimizer | ProGuard. https://www.guardsquare.com/proguard (Accessed:2025-06-25).

[11] 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* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 380–394. doi:10.1145/3243734.3243838

[12] Kihong Heo and Pardis Pashakhanloo. 2020. ChiselBench. https://github.com/aspire-project/chisel-bench (Accessed:2025-06-28).

[13] Marc R. Hoffmann and Cay S. Horstmann. 2025. The Java Version Almanac. https://javaalmanac.io/ (Accessed:2025-06-28).

[14] Zhenghao Hu, Sangho Lee, and Marcus Peinado. 2023. Hacksaw: Hardware-Centric Kernel Debloating via Device Inventory and Dependency Analysis. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1994–2008. doi:10.1145/3576915.3623208

[15] Judit Jász, István Siket, Edit Peng?, Zoltán Ságodi, and Rudolf Ferenc. 2019. Systematic Comparison of Six Open-source Java Call Graph Construction Tools. In *Proceedings of the 14th International Conference on Software Technologies* (Prague, Czech Republic) *(ICSOFT 2019)*. SCITEPRESS - Science and Technology Publications, Lda, Setubal, PRT, 117–128. doi:10.5220/0007929201170128

[16] Christian Gram Kalhauge and Jens Palsberg. 2019. Binary reduction of dependency graphs. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Tallinn, Estonia) *(ESEC/FSE 2019)*. Association for Computing Machinery, New York, NY, USA, 556–566. doi:10.1145/3338906.3338956

[17] Kadiray Karakaya, Stefan Schott, Jonas Klauke, Eric Bodden, Markus Schmidt, Linghui Luo, and Dongjie He. 2024. SootUp: A Redesign of the Soot Static Analysis Framework. In *Tools and Algorithms for the Construction and Analysis of Systems*, Bernd Finkbeiner and Laura Kovács (Eds.). Springer Nature Switzerland, Cham, 229–247.

[18] Jiakun Liu, Zicheng Zhang, Xing Hu, Ferdian Thung, Shahar Maoz, Debin Gao, Eran Toch, Zhipeng Zhao, and David Lo. 2024. MiniMon: Minimizing Android Applications with Intelligent Monitoring-Based Debloating. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 206, 13 pages. doi:10.1145/3597503.3639113

[19] Gregory Malecha, Ashish Gehani, and Natarajan Shankar. 2015. Automated software winnowing. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing* (Salamanca, Spain) *(SAC '15)*. Association for Computing Machinery, New York, NY, USA, 1504–1511. doi:10.1145/2695664.2695751

[20] Jorge A. Navas and Ashish Gehani. 2022. OCCAM-v2: Combining Static and Dynamic Analysis for Effective and Efficient Whole-program Specialization: Leveraging scalable pointer analysis, value analysis, and dynamic analysis. *Queue* 20, 5 (Nov. 2022), 58–85. doi:10.1145/3570922

[21] OWASP Foundation, Inc. 2025. Deserialization of untrusted data. https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data (Accessed:2025-06-25).

[22] Serena Elisa Ponta, Wolfram Fischer, Henrik Plate, and Antonino Sabetta. 2021. The Used, the Bloated, and the Vulnerable: Reducing the Attack Surface of an Industrial Application. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 555–558. doi:10.1109/ICSME52107.2021.00056

[23] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. 2019. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories* (Montreal, Quebec, Canada) *(MSR '19)*. IEEE Press, 383–387. doi:10.1109/MSR.2019.00064

[24] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 251–261. doi:10.1145/3293882.3330555

[25] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 107–112. doi:10.1145/3236454.3236503

[26] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. 26, 3 (2021), 45. doi:10.1007/s10664-020-09914-8

[27] César Soto-Valero, Deepika Tiwari, Tim Toady, and Benoit Baudry. 2023. Automatic Specialization of Third-Party Java Dependencies. *IEEE Transactions on Software Engineering* 49, 11 (2023), 5027–5045. doi:10.1109/TSE.2023.3324950

[28] Standard Performance Evaluation Corporation. 2006. SPEC CPU® 2006 benchmark (Retired: January 2018). https://www.spec.org/cpu2006/ (Accessed:2025-06-28).

[29] Standard Performance Evaluation Corporation. 2017. SPEC CPU® 2017 benchmark. https://www.spec.org/cpu2017/ (Accessed:2025-06-28).

[30] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 2018. On the Soundness of Call Graph Construction in the Presence of Dynamic Language Features - A Benchmark and Tool Evaluation. In *Programming Languages and Systems*, Sukyoung Ryu (Ed.). Springer International Publishing, Cham, 69–88.

[31] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the recall of static call graph construction in practice. In *Proceedings of the ACM/IEEE*

42nd International Conference on Software Engineering (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 1049–1060. doi:10.1145/3377811.3380441

[32] Xiaoke Wang, Tao Hui, Lei Zhao, and Yueqiang Cheng. 2023. Input-Driven Dynamic Program Debloating for Code-Reuse Attack Mitigation. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 934–946. doi:10.1145/3611643.3616274

[33] Ryan Williams, Tongwei Ren, Lorenzo De Carli, Long Lu, and Gillian Smith. 2021. Guided Feature Identification and Removal for Resource-constrained Firmware. *ACM Trans. Softw. Eng. Methodol.* 31, 2, Article 28 (Dec. 2021), 25 pages. doi:10.1145/3487568