# JShrink: In-depth Investigation into Debloating Modern Java Applications

Bobby R. Bruce
University of California, Los Angeles
Computer Science Department
b.bruce@cs.ucla.edu

Tianyi Zhang
University of California, Los Angeles
Computer Science Department
tianyi.zhang@cs.ucla.edu

Jaspreet Arora
University of California, Los Angeles
Computer Science Department
jasa92@g.ucla.edu

Guoqing Harry Xu
University of California, Los Angeles
Computer Science Department
harryxu@cs.ucla.edu

Miryung Kim
University of California, Los Angeles
Computer Science Department
miryung@cs.ucla.edu

## ABSTRACT

Modern software is bloated. Demand for new functionality has led developers to include more and more features, many of which become unneeded or unused as software evolves. This phenomenon, known as software bloat, results in software consuming more resources than it otherwise needs to. How to effectively and automatically debloat software is a long-standing problem in software engineering. Various software debloating techniques have been proposed since the late 1990s. However, many of these techniques are built upon pure static analysis and have yet to be extended and evaluated in the context of modern Java applications where dynamic language features are prevalent.

To this end, we develop an end-to-end bytecode debloating framework called JShrink and conduct an in-depth analysis of bytecode transformations for debloating modern Java applications. JShrink augments traditional static reachability analysis with dynamic profiling and type dependency analysis and renovates existing bytecode transformations to account for new language features in modern Java. We highlight several nuanced technical challenges that must be handled properly to debloat modern Java applications and further examine behavior preservation of debloated software via regression testing. Our study finds that (1) JShrink is able to debloat our real-world Java benchmark suite by up to 47% (14% on average); (2) accounting for dynamic language features is indeed crucial to ensure behavior preservation for debloated software—reducing 98% of test failures incurred by a purely static equivalent, Jax, and 84% for ProGuard; and (3) compared with purely dynamic approaches, integrating static analysis with dynamic profiling makes the debloated software more robust to unseen test executions—in 22 out of 26 projects, the debloated software ran successfully under new tests.


*ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States*

ACM ISBN 000-0-0000-0000-0/00/00…$15.00
https://doi.org/00.0000/0000000.0000000

## KEYWORDS

Java bytecode, size reduction, reachability analysis, debloating

**ACM Reference Format:**
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 12 pages. https://doi.org/00.0000/0000000.0000000

## 1 INTRODUCTION

The size and complexity of software has grown tremendously in recent decades. Though largely beneficial, this has led to unchecked bloat issues that are especially severe for modern object-oriented applications due to their excessive use of indirection, abstraction, and ease of extensibility. This problem of customizing and tailoring modern applications to only used components, in an automated fashion, is a long standing problem [25, 28, 32, 42, 43, 45, 49, 56, 58, 60, 64, 65].

Prior work on code size reduction focuses primarily on C/C++ binaries [25, 32, 42, 43, 49, 65], motivated by the long-held belief that C/C++ programs are easier to attack and are often choices for software development for embedded systems. However, with the rise of cloud computing, Android-based smart-phones, and smart-home internet-of-the-things, a managed, object-oriented language such as Java is making its way into all important domains and machines of all sizes. Although reducing the size of Java bytecode, which is the main goal of our effort, may not ultimately lead to a significant improvement in a traditional stand-alone machine setting, its benefit becomes orders of magnitude more significant in many modern small and large-scale computing scenarios—smaller bytecode size directly translates to reduced download size and loading time in smartphones and reduced closure serialization time in big data systems such as Apache Spark; these are all important performance metrics for which companies are willing to spend significant resources in optimizing.

However, past work has not given much attention to Java, especially debloating modern Java applications. Of particular interest to us is Tip et al.'s work [58] in the late 1990s that proposes various bytecode transformations for software debloating, which have since been utilized by other researchers [4, 13, 28]. In surveying the literature, we find that their effectiveness has yet to be systematically

evaluated on a real-world benchmark of modern Java applications. All previous implementations of those bytecode transformations relied on pure static analysis to identify reachable code, hereby ignoring code reachable through reflection, dynamic proxy, callbacks from native code, etc. Recent studies find that *dynamic language features* are prevalent in modern Java applications and they pose direct challenges in the soundness of static analysis [31, 37]. This unsoundness subsequently makes debloating *unsafe*—removing dynamically invoked code and inducing subsequent test failures. Furthermore, evaluations in prior work focus mostly on size reduction rather than behavior preservation, which raises a big safety concern for adopting debloating techniques in practice.

Therefore, we undertake the ambitious effort of modernizing and evaluating Java bytecode debloating transformations to account for new Java language features, e.g., dynamic proxy, pluggable annotation, lambda expression, etc., and quantify the tradeoff between size reduction and debloating safety. We augment static reachability analysis with dynamic profiling to handle dynamic language features. We incorporate a new type dependency analysis to account for a variety of ways to reference types in modern Java, like annotations and class literals to ensure type safety after debloating. We replicate and extend four kinds of debloating transformations—*method removal, field removal, method inlining*, and *class hierarchy collapsing* into a fully automated, end-to-end debloating framework called JShrink. JShrink allows for the utilization of these transformations either individually or en-masse.

To effectively evaluate those bytecode transformations, we built an automated infrastructure to construct a benchmark of real-world, popular Java applications. We applied a rigorous set of filtering criteria: (1) reputation score based on the GitHub Star rating system, (2) executable tests, (3) a Maven build script [39], which provides a standardized interface for obtaining library dependencies and regression testing, and (4) compatibility with the underlying bytecode analysis framework, Soot [61]. The availability of runnable test cases enables us to examine to what extent the behavior of original software is preserved after debloating via regression testing. Currently, the resulting benchmark includes 22 projects with SLOC ranging from 328 to 99,779 and with up to 69 library dependencies. We then apply JShrink to this benchmark to quantify size reduction, the degree to which test behavior could be preserved, and the impact of Java dynamic language features by answering the following research questions:

- RQ1 How much Java byte code reduction is achievable when applying different kinds of transformations?
- RQ2 To what extent, does JShrink preserve program correctness when debloating software?
- RQ3 What are the trade-offs in terms of debloating potential and semantic preservation?
- RQ4 How robust is the debloated software to unseen test executions such as new test cases?

JShrink reduces a project's size (application and included library dependencies) by up to 46.8% (14.2% on average). The *method removal* component reduces the application by the most (11.0% on average) followed by *method inliner* (2.1% on average), *field removal* (1.0% on average), and *class hierarchy collapser* (0.1% on average). A hybrid static and dynamic reachability analysis is necessary for improving behavior preservation of debloated software. JShrink does not break any existing tests for 22 out of 26 Java projects after debloating, while three existing techniques, Jax [58], JRed [28], and ProGuard [4] that rely on pure static analysis preserve behavior for only 9, 11, and 15 projects respectively. While this comparison in terms of the number of projects may look marginal, 98% of test failures encountered in Jax (83% for ProGuard) can be actually removed by JShrink's enhancements. This result implies the effort of handling new language features is absolutely necessary and worthwhile for improving behavior preservation, which justifies the need to address the long-standing debloating problem in the modern context. We find that size reduction potential is minimally impacted by this incorporation of dynamic reachability analysis. In other words, we only sacrifice size reduction by 2.7% on average, while providing much stronger behavior preservation guarantees. In order to achieve 100% behaviour transformation we enable *checkpointing*—a feature of JShrink where transformations are reverted if they are found to break the semantics of a target program. Though this stategy incurs marginal losses in size reduction (0.9% on average), we believe checkpointing to be a practical solution for balancing semantic preservation and code size reduction benefits.

Our work makes the following contributions:

- We present JShrink, an end-to-end Java bytecode debloating framework that replicates and modernizes four distinct bytecode transformations to handle new language features in the modern context.
- We find bytecode reduction of up to 46.8% is possible, where reachability-based method removal plays a dominant role in size reduction. JShrink ensures that debloated software still passes 98% of existing tests.
- We demonstrate the necessity of handling dynamic features and ensuring type safety. JShrink removes 98% and 83% of test case failures incurred by Jax [58] and ProGuard [4].
- We put forward an automated infrastructure of constructing real-world Java applications with test cases, a build script, and library dependencies for assessing debloating potential and checking behavior preservation using tests.

The main research contribution of this paper is *on systematization of the community's knowledge of Java debloating* in the modern era. As a reference point, several top conferences [12, 18, 47] have already started to have a "systematization of knowledge" track, with the goal to address the concern that "the community seems to lose memory of things that have been done in the past." With this paper, we hope to bring existing debloating techniques into a contemporary context where dynamic features are prevalent and where behavior preservation must be ensured and checked using tests. We make publically available the JShrink source code and additional resources necessary to replicate our results at https://doi.org/10.6084/m9.figshare.12435542.

## 2 BACKGROUND

*Scope: Java Bytecode.* The problem of software bloat has been a center of research studies for more than a decade in the area of performance tuning and optimization. Recently, there is a revived interest—partly due to the need of cyber defense (e.g., US Navy's Total Platform Cyber Protection (TPCP) program [2])—in extending

traditional debloating techniques to reduce code size, improve runtime performance, and remove attack surfaces for a wide spectrum of software applications, including JavaScript programs [64], native applications [43], and Docker containers [44].

In this paper, we focus on *code size reduction* as opposed to runtime memory bloat that was the target of a large body of prior work [40, 67, 69–71]. While code bloat exists commonly in a broad range of applications, we focus on object-oriented programs (specifically Java bytecode) as our scope for two reasons.

First, the culture of object orientation encourages developers to use frameworks, patterns, and libraries even for extremely simple tasks, resulting in a large number of classes and methods, which, though not used at all during execution, still need to be loaded by JVM due to type-induced dependencies. These classes and methods consume extra space and memory, thereby negatively impacting the performance of resource-constrained systems such as smart phones or IoT devices. Furthermore, they can potentially contain security vulnerabilities (e.g., gadgets in return-oriented programming [11]), which can be exploited by remote attackers to execute code segments that could not have been reached otherwise.

Second, many recent techniques [25, 42, 43, 49] on code bloat target native x86 programs, aiming to reduce the size of executable binaries. Native programs are significantly different from object-oriented programs in terms of compilation and execution. Native programs are statically compiled and linked, with most libraries statically loaded. In many cases, a compiler can already remove much of dead code. On the contrary, object-oriented programs are often *dynamically compiled and loaded*; the ubiquitous use of dynamic features such as dynamic class loading and reflection dictates that a compiler would not know which classes to load until the moment they are needed.

*History: Static Bytecode Debloating.* In the late 1990s, Tip et al. developed Jax, which included, so far, the most comprehensive set of transformations to reduce Java bytecode, including *method removal*, *field removal*, *method inlining*, *class hierarchy transformation*, and *name compression* [58]. They later introduced two more transformations, *class attribute removal* and *constant pool compression* in their 2002 journal paper [59]. Recent techniques are based on a subset of these transformations to debloat new types of applications, e.g., Android [27] and Maven libraries in continuous integration [13]. JRed [28] and RedDroid [27] only support the *method removal* and *class removal* transformations, while Molly [13] supports *field removal* as well. These above mentioned techniques are *outdated* or *not publicly available*. Furthermore, their evaluations did not quantify the degree to which debloated software preserves semantics by running existing tests. Behavior preservation is crucial for these techniques to be adopted in practice.

*Motivation for Modernizing Software Debloating and Assessing Behavior Preservation.* Java offers a number of dynamic features widely used in real-world programs [31]—reflection, dynamic class loading, dynamic proxy, etc., which are highly challenging to model through pure static analysis. Livshits et al. first investigated this problem in 2005 using points-to-analysis to statically resolve dynamic method invocation targets [38]. Other attempts focused on a specific scope of dynamic features such as reflection [36, 51], dynamic proxy [20], etc. Most static analysis tools tolerate and encourage some level of unsoundness to keep the analysis usable and scalable [37]. Landman et al. conduct a systematic literature review and an empirical study to assess the effectiveness of 24 different static analysis tools in the presence of real-world Java reflection usage [31]. They find that static analysis is inherently incomplete and reflection cannot be ignored for 78% of projects. This finding motivates our effort to evaluate the safety of debloating techniques in the context of dynamic language features. In Section 5.3, we quantify this benefit of handling dynamic features—debloated software based on pure static analysis would fail 3327 more tests in 26 projects.

*Profile-Augmented Static Debloating and Checking of Behavior Preservation.* Existing debloating techniques only assess the code reduction and performance improvement achieved by different kinds of bytecode transformations [4, 27, 28, 59]. None of them assess *the correctness of reduced programs by running existing test cases.* Furthermore, these techniques only perform static call graph analysis to approximate used code, and are incomplete in the presence of various dynamic language features discussed in Section 3. Ergo, test failures are inevitable, as dynamically invoked code could be removed by debloating. In this paper, we take a *profile-augmented static debloating* approach—we augment static reachability analysis with dynamic reachability analysis using existing tests; we remove code through static bytecode transformations; and we check behavior preservation by running existing tests after debloating.

## 3 JSHRINK

We build an end-to-end bytecode debloating framework called JShrink. Given the bytecode of a Java program and a set of test cases, JShrink takes three phases to debloat bytecode and verify its correctness. In Phase I, JShrink performs profile-augmented static analysis to determine used and unused code. In Phase II, JShrink applies four kinds of debloating transformations. Finally, JShrink reruns the given test cases to check behavior preservation between the original and the debloated version.

### 3.1 Profile Augmented Static Analysis

We apply three types of analyses—static reachability analysis, dynamic profiling, and type dependency analysis—to capture method invocation, field access, and class reference relationships between class entities. This is essential to determine unused code in the presence of dynamic language features and ensure type safety of debloated bytecode, especially in class hierarchy merging.

***Static Reachability Analysis.*** Static call graph analysis is a standard method used by previous bytecode debloating techniques [27, 28, 59] to decide unused methods. Given a set of methods (e.g., `main` methods, test cases, etc.) as entry points, it analyzes the body of each method and identifies call sites in the method body. Call graph analysis then constructs a directed graph for each entry method and adds edges from the entry method to its callee methods. Those callee methods are then treated as new entry points and the process continues until no additional methods are found, reaching a fix point.

Due to polymorphism in object-oriented languages, multiple call targets could be invoked from a call site via dynamic dispatching, depending on the runtime type of the receiver object. Various techniques have been proposed to approximate possible targets

of a dynamic dispatch, e.g., class hierarchy analysis (CHA) [15], 0-CFA [24, 50], rapid type analysis (RTA) [7], points-to analysis [35, 48], etc. Specifically, JShrink leverages CHA to construct call graphs, which identifies all corresponding method implementations of a callee in the subclasses of the declared receiver object type and considers them as potential call targets. We perform a whole-program analysis, including application code, imported third-party libraries, and JRE, to build call graphs. In addition, we use ASM [10] to analyze field accesses in each method and extend the call graphs with field access information.

***Dynamic Reachability Analysis.*** We initially considered using a lightweight dynamic analysis approach called TamiFlex [9], as it is a well known technique for addressing unsoundness caused by Java reflection. Tamiflex instruments Java reflection call sites to capture method calls and field accesses via reflection at runtime. However, TamiFlex is designed for reflection APIs only and thus lacks support for other dynamic features, which leads to many test failures, evidenced by our comparison results in Section 5.4. To systematically account for dynamic features, we define a comprehensive list of dynamic features based on Sui et al. [55].

(1) *Reflection* is a dynamic feature that enables users to dynamically instantiate classes, access fields, and invoke methods. It is widely used in modern Java context and is the foundation for many frameworks such as Spring and JUnit [31].
(2) *Reflection with ambiguous resolution* refers to a special case where multiple potential targets exist (e.g., overloading methods with different return types) for a dynamic invocation via reflection. Such bytecode is often generated by bytecode manipulation instead of by standard compilers.
(3) *Dynamic classloading* involves classes loaded through custom class loaders.
(4) *Dynamic proxy* refers to the proxy feature that dynamically creates invocation handlers for a class and its methods.
(5) *Invokedynamic* is a new bytecode instruction introduced in Java 7 that enables dynamic method invocation via method handles. It is often used to support *lambda expressions*.
(6) *Serialization* refers to dynamically loaded classes via class deserialization.
(7) *Java Native Interface (JNI)* is a framework that enables Java to call and be called by native code. This benchmark includes two programs that have callbacks from native code via JNI.
(8) *sun.misc.Unsafe* is a low-level Java API that can be used to directly manipulate JVM memory at runtime, e.g., dynamically loading classes, throwing exceptions, swapping instances, allocating new instances, etc.

We develop our own native profiling agent called *Jmtrace*, which instruments method invocations using JVM TI APIs[1] to inject logging statements at the entry and exit of each method in a class during class loading. Table 1 compares the capability of handling different kinds of dynamic features between static call graph analysis, TamiFlex, and *Jmtrace*. JShrink runs given test cases and identifies dynamic method calls that do not exist in static call graphs but are invoked during test execution. Then JShrink initiates another round of static reachability analysis using those dynamically invoked methods as entry points. Note that we only use dynamic profiling to augment static analysis, instead of replacing static analysis with dynamic analysis. In case of low test coverage from existing tests, this augmentation lets JShrink retain functionality statically reachable from user-specified entry points, such as public methods, `main` methods, and method entries from existing test cases.

[1]https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/

**Table 1: Capability of Handling Different Dynamic Features**

| | Static | Tamiflex | Jmtrace |
|---|---|---|---|
| Reflection | ○ | ● | ● |
| Reflection-ambiguous | ○ | ◐ | ● |
| Dynamic class loading | ○ | ● | ● |
| Dynamic proxy | ○ | ○ | ● |
| Invokedynamic | ◐ | ◐ | ● |
| JNI | ○ | ○ | ● |
| Serialization | ○ | ● | ● |
| Unsafe | ○ | ◐ | ● |

JVM TI APIs only permit instrumentation of method modies. As such *Jmtrace* is not capable of identifying fields dynamically accessed via reflection. Therefore, we customize TamiFlex to instrument only reflection calls related to field accesses and use it together with *Jmtrace*. Instrumentation to other reflection calls is disabled to avoid redundant profiling.

***Type Dependency Analysis.*** Traditional reachability analysis only keeps track of invoked methods and accessed fields, which is sufficient for method and field removal. Previous debloating techniques consider a class unused if none of its methods or fields are reachable from entry points [27, 28, 58]. However, we find this definition of unused classes is *problematic* in practice.Modern Java allows developers to reference classes in various waysi, not just limited to variable and method declaration or class inheritance, but through pluggable annotations, class literals, throws clauses, etc. A program can thereby reference a class without instantiating it, or directly access any of its methods of field members. In such a case, removing reference-only classes that do not have any method or field usage will cause a bytecode verification failure during class loading in the JVM or lead to a `ClassNotFoundException` at runtime. It is crucial to ensure type safety during class removal and class hierarchy collapsing. Therefore, JShrink builds type dependency graphs by scanning through Java bytecode using ASM. If a class `A` is referenced by a class `B`, we add an edge from `B` to `A` in the graph.

Based on static analysis, profiling of dynamic features, and type-dependency analysis, JShrink determines unused code at four granularities, listed below. We use "class" as a general term for concrete classes, abstract classes, and interfaces in Java.

- **Unused Method**: A method is unused if it is not reachable from any given entry point in the call graphs.
- **Unused Field**: A field is unused if it is not accessed by a used method in a call graph or dynamically accessed via reflection.
- **Unused Class**: A class is considered unused if none of the following three conditions are satisfied: (1) A method in the class is reachable from given entry points; (2) A field in the class is reachable from given entry points; (3) A descendant of this class in the class hierarchy is used.

- **Reference-only Class**: A class is not used but referenced by another used or reference-only class based on given type dependency graphs. This is a special category of classes not handled safely by existing debloating techniques [27, 28, 58]. In prior work, unused classes are completely removed if none of their class members are reachable. However, when replicating class-level bytecode transformations, we find that this is an unsafe choice, causing many `ClassNotFoundErrors` at runtime. Therefore, JShrink *partially* debloats reference-only classes to ensure type safety, as explained in class hierarchy collapsing.

## 3.2 Bytecode Debloating Transformations

Inspired by Tip et al. [58], JShrink provides the following bytecode debloating transformations.

***Unused Method Removal.*** JShrink provides three method removal options for a user to choose from—(1) completely remove the definition of an unused method, (2) only remove the body of an unused method but keep the method header, and (3) replace the method body with a warning statement indicating the method is removed. To safely wipe a method body, JShrink injects bytecode instructions to return dummy values if the return type is not `void`. The first option could achieve maximum code size reduction at the cost of safety, as it may lead to `NoSuchMethodError` if a removed method is triggered in future usages. With the second and the third options, unused methods are still defined in bytecode and thus programs will fail gracefully without catastrophic program crashes. The third option is the most informative, as it lets a user know which method is invoked at runtime but not captured by static analysis or given test cases. Our results in Section 5 uses the first option as default, but a user may choose the other two options in JShrink.

***Unused Field Removal.*** Given an unused field, JShrink completely removes its definition. Note that this transformation should be used in pair with method removal. If those unused methods accessing an unused field are not removed, JVM will report `FieldNotFoundError` that crashes the debloated software. Enabling this transformation alone requires fine-grained transformation within a method body, e.g., removing all field access instructions and subsequent instructions with data dependencies to the field.

***Method Inlining.*** JShrink inlines a method if the method has only one call site in the call graph and the method is the only call target the callsite. The former ensures that JShrink does not introduce code duplication during inlining, while the latter is crucial for semantic preservation in case of polymorphism.

Type safety of method inlining is widely discussed in the compiler literature [22, 23]. To ensure type safety, JShrink applies three constraints. First, JShrink does not inline class constructors. Second, JShrink does not inline native methods, abstract methods, and interface methods as they do not have method bodies. Third, JShrink does not inline a method if it accesses other class members that become invisible after inlining (detailed in Section 3.3). JShrink also does not inline `synchronized` methods.

***Class Hierarchy Collapsing.*** JShrink performs two basic transformations to collapse class hierarchy. The first, more sophisticated, transformation is to merge a base class $X$ and a subclass $Y$, if $Y$ is the only used subclass of $X$. JShrink checks if, for any overridden method $m'$ in $Y$, and the corresponding original method $m$, only one of either $m$ and $m'$ is used. If both are used, JShrink does not collapse the classes. If this rule was not enforced, JVM would not delegate an invocation on $m$ to its overridden method, $m'$, based on the real type of the receiver object at runtime. The second transformation is to remove unused classes. For a reference-only class, JShrink removes its class members and only retains the class header to avoid `ClassNotFoundError`. If a reference-only class is a concrete class, JShrink injects a default constructor as enforced by JVM. If a reference-only class is an interface, JShrink keeps those method declarations whose implementations in a subclass are used.

To implement the first transformation of merging a subclass $Y$ into a base class $X$, JShrink takes three steps. First, it moves all used method and field members of $X$ into $Y$ while removing unused class members in $Y$. Secondly, it updates all references to the merged subclasses, their method and field members, to their new locations after merging. During the merging and updating process, name conflicts may occur due to method overloading rules enforced by Java. For instance, class `B` may have overloaded methods `void m(A a)` and `void m(SubA a)`. After merging `SubA` to `A` and updating the parameter type of the second method in `B`, the signatures of the two methods become identical. Therefore, to handle name conflicts, JShrink renames methods and further updates references to those renamed methods as needed. Since class constructors cannot be renamed, in instances where naming conflicts with them, JShrink adds a new dummy integer parameter to "rename" a constructor and update all call sites of the renamed constructor by pushing an integer value, 0, on the stack.

***Checkpointing.*** While experimenting with real-world Java projects, we note that test failures may still occur due to rare but challenging corner cases caused by known limitations of JVMTI and Soot (Section 5.3). Therefore, JShrink implements an additional strategy of checkpointing to ensure safety. It checkpoints each type of debloating transformation, runs tests, and reverses failure-inducing transformations.

## 3.3 Implementation and Nuanced Extensions

We implement those bytecode debloating transformations using Soot [61]. We use the CHA implementation in Soot for static analysis, and ASM [10] to gather field accesses. We implement *Jmtrace* using JVM TI APIs. We highlight several nuanced extensions that we designed to ensure type safety and behavior preservation.

(1) *Co-variant return type.* From Java 5 onward, JVM supports co-variant return types, which allow an overridden method to have a return type different to the original. Therefore, instead of simply comparing whether two method signatures are the same, JShrink accounts for co-variant return types to determine overridden methods when merging two classes. Otherwise, JVM will throw a verification error.
(2) *Class member visibility.* When inlining a method or merging a class, it is important not to break access controls. If method $m$ from class $A$ is to be inlined into class $B$, JShrink enforces that $m$ does not call other private methods in $A$. Otherwise, JVM will raise `IllegalAccessError` since those

private methods are not visible to *B*. Similarly, if subclass *A* is in a different package compared with its superclass *B* and *A* contains a protected method that is called by another class *C* in the same package as *A*, merging *A* into *B* will cause `IllegalAccessError` since *A.m* becomes invisible to *C* after moving to a different package. Before merging a class to a different package, JShrink checks whether a protected method or field will become invisible after merging.

(3) *Lambda expression.* Lambda expressions are introduced in Java 8. They are anonymous functions that can be passed as parameters to method calls. For example, in `v.forEach(x -> A.foo())`, the lambda `x -> A.foo()` is passed to the `forEach` method and could be executed at runtime. Therefore, the method call `foo` must be captured by call graph analysis. This expression can be rewritten to `v.forEach(A::foo(x))` using the new method reference operator "::". JShrink checks for both cases and adds missing edges between the caller and method calls in a lambda expression to call graphs.

(4) *Class literals.* Class literals such as `X.class` are compiled to string constants in Java bytecode. It is critical to identify class references via class literals and add them to type dependency graphs to avoid `ClassNotFoundError`. JShrink identifies class literals by matching "*.class" against string constants used in a class. JShrink also updates the class literal of a merged class to its superclass to avoid `ClassNotFoundError`.

(5) *Method inheritance.* Merging classes in presence of both method overriding and inheritance could be problematic. Suppose a base class `A` inherits a method `m` from its super class `B` and its subclass `C` overrides `m`. If `A.m` is reachable from an entry point, it is hard to decide whether `A.m` is actually invoked on `A` objects or `B` objects due to polymorphism. If `A.m` is only invoked on `A` objects, we can safely merge class `C` into `B` even when the overridden method `C.m` is also used. However, if `A.m` is invoked on `B` objects, moving `C.m` into `B` will alter the dynamic patching behavior. In such a case, JShrink make a conservative choice of not merging a subclass to its base class, if (1) the base class inherits a used method from its superclass or an ancestor, (2) the subclass also overrides the same method, and (3) the overridden method is also used.

In summary, compared with prior work, JShrink makes the following major extensions to handle modern Java: (1) augmenting static reachability analysis with JVM TI based dynamic profiling, (2) incorporating type dependency analysis, (3) extending method inlining and class hierarchy collapsing transformations to ensure type safety, and (4) all nuanced extensions in Section 3.3 to handle new language features properly.

## 4 BENCHMARK

We build an automated infrastructure to construct a benchmark. It uses the Google BigQuery API to query GitHub projects and automatically applies a rigorous set of filtering criteria listed below to include real-world, popular Java projects on GitHub.[2]

- *Popular Java projects.* We are interested in high-quality Java projects, widely used by software developers. Therefore, our infrastructure chooses projects with at least 100 GitHub stars from other developers. The star rating ranges from 188 to 16205 on our benchmark, with an average of 3145.
- *Automated build system.* Our infrastructure requires a standardized API to automatically resolve library dependencies, compile target projects, and run test cases. The current implementation supports Maven [39], a popular build system used in Java, but could be easily extended to support other build systems such as Graddle.
- *Compilable.* After downloading those projects, we exclude those that induce build failures in our environment (an Amazon `r5.xlarge` instance with Ubuntu 18.04 and JDK 1.8.0), due to specific hardware or library configurations.
- *Executable tests.* We rely on test cases to evaluate to what extent debloated software preserves its original behavior. Therefore, after compiling a project, our infrastructure runs the Maven test command and parses generated test reports to identify the number of test cases and test failures. Projects with no test or any test failure are excluded.
- *No JVM verification errors.* Note that, when Soot writes code from its intermediate language, Jimple, back to bytecode, it automatically applies several optimizations such as constant pool compression. Therefore, we first pre-process all Java bytecode using Soot to fairly measure code size reduction achieved by JShrink. In this preprocessing step, fatal JVM verification errors could occur in some Java projects. We discard those projects due to JVM verification errors.
- *No Timeout.* Our infrastructure enforces a timeout constraint on the profile-augmented static analysis, since generating call graphs for some projects may take an excessively long time. We set this timeout to 10 hours.

**Table 2: Project statistics**

| | Stars | Tests | Libs | SLOC (App Only) | Size (KB: App+Libs)[3] |
|---|---|---|---|---|---|
| **Max** | 16,209 | 1,081 | 69 | 99,779 | 114,312 |
| **Min** | 188 | 1 | 0 | 328 | 30 |
| **Mean** | 3,135 | 237 | 15 | 14,729 | 15,734 |
| **Median** | 2,000 | 60 | 9 | 5,863 | 3,193 |
| **Total** | 69189 | 5213 | 332 | 324,035 | 346,160 |
| **SD** | 3,595 | 370 | 17 | 22,288 | 30,766 |

The final benchmark shown in Table 3 covers a wide spectrum of Java programs, including popular libraries, web applications, development and testing frameworks, and desktop applications. Table 2 summarizes the statistics for those 26 benchmark programs. All are popular GitHub projects with a median of 2,000 stars, where the average number of test cases and external library dependencies are 237 and 15 respectively. The total size of the projects (inclusive of library dependencies) range from 30KB to 112MB with a median of 3MB. Cobertura [1] reports 34.1% statement coverage by their existing tests, which we use for assessing behavior preservation after debloating.

[2] Google BigQuery API and GitHub Dataset, https://cloud.google.com/bigquery/public-data/.

[3] The total size reported is that of project and library dependencies in their compiled states.

# 5 EXPERIMENTS

We run JSHRINK on the benchmark of 26 popular Java projects and compare it with three existing bytecode debloating techniques to answer the questions outlined in this paper's introduction:

- RQ1 How much Java byte code reduction is achievable when applying different kinds of transformations?
- RQ2 To what extent program semantics is preserved when debloating software?
- RQ3 What are the trade-offs between debloating potential and preservation of software semantics?
- RQ4 How robust is the debloated software to unseen executions such as new test cases?

## 5.1 Experiment Setup and Baselines

Our experiments run on an Amazon `r5.xlarge` instance (3.1 GHz 4-core Intel Xeon Platinum processor, 32GB Memory) with Ubuntu 18.04 and JDK 1.8.0 installed. We choose this standard cloud-based setup to ease the replication effort for other researchers. We compare JSHRINK with Jax [58], JRed [28], and ProGuard [4]. Since both Jax and JRed are not available, we faithfully re-implement them based on their paper descriptions.

***Jax*** includes the most comprehensive set of bytecode transformations. To replicate Jax, we adapt JSHRINK to use static call graph analysis only and disable Section 3.3's extensions. Jax imposes an additional constraint that requires unused, to-be-removed classes not to have any derived classes. So we modify the class collapsing transformation accordingly.

***JRed*** only supports method removal and class removal. Like Jax, JRed relies on static call graph analysis only. To replicate JRed, we adapt JSHRINK to use static call graph analysis exclusively, only enable unused method removal and unused class removal, and disable all extensions from Section 3.3.

***ProGuard*** shrinks and obfuscates Java bytecode. Unlike Jax and JRed, it is publicly available. It has been integrated into Android SDK and is widely used to optimize Android applications. Similar to Jax and JRed, ProGuard performs static analysis only. It does not construct call graphs but instead traverses bytecode instructions in a given method to calculate a transitive closure of all referenced classes, methods, and fields. ProGuard has some static analysis support for Java reflection but is not accurate, since it only analyzes hardcoded strings passed into a pre-defined set of reflection APIs. As ProGuard is publically available, we evaluate it directly. We use version 6.3.

## 5.2 RQ1: Code Size Reduction

To answer RQ1, we apply the four transformations implemented in JSHRINK on each project individually and en-masse. The evaluations of Jax [58] and JRed [28] in their original papers only use `main` methods as the entry points. However, we find that many projects (such as `gson` and `java-apns` in our benchmark) are library projects whose public classes and methods are potentially invoked by downstream client projects. Therefore, in our experiments, we make a conservative choice of setting all public methods, `main` methods, and test methods as entry points to maximally approximate possible usage.

We report the size reduction of bytecode only, excluding resource files. Column Transformations in Table 3 shows the size reduction ratio achieved by each transformation. Compared with the other three transformations, *method removal* (Column MR) is the most effective in size reduction, achieving an average of 11.0% reduction (up to 42.2%). *Method inlining* (Column MI) and *field removal* (Column FR) reduce bytecode by 2.1% and 1.0% respectively on average. *Class hierarchy collapsing* (Column CC) only achieves a minimal reduction of 0.1% on average (up to 0.6%).

Column Code Size Reduction in Table 3 shows the size reduction achieved by all transformations, compared with Jax, JRed, and ProGuard. Specifically, Column JShrink-C shows the size reduction when enabling the checkpoint feature to automatically reverse failure-inducing transformations. When applying all transformations together, JSHRINK can reduce a project by up to 46.8% (14.2% on average). Checkpointing only has a minimal impact on size reduction (0.9% less reduction) while achieving 100% semantic preservation. JRed has the smallest size reduction (13% on average). This is because JRed only supports two kinds of transformations, method removal and class removal. Though both Jax and JSHRINK support the same set of transformations, Jax achieves a larger size reduction, 17.0% in comparison to 14.2% in JSHRINK for two reasons. First, JSHRINK retains dynamically called methods and loaded methods. Second, JSHRINK partially debloats reference-only classes, while Jax completely removes them. ProGuard crashes on two projects due to a known bug while performing partial evaluation on strings [3]. Compared with JSHRINK, ProGuard reduces code more aggressively (33.8% on average) because it performs static reference-based analysis, producing a smaller set of reachable methods. However, ProGuard causes 6X more test failures than JShrink, as elaborated in the next section.

## 5.3 RQ2: Semantic Preservation

Code size reduction, however, is only meaningful if the semantics of the target program is preserved. To assess how closely JSHRINK preserves program semantics, we run existing test cases before and after debloating. A program is considered to have broken semantics if there exist any test failures after debloating. Column Test Failures shows the semantic preservation capability of JSHRINK, Jax, JRed, and ProGuard. "✓" denotes a project with no test failure after debloating, while "×" denotes that test failures exist after debloating. The numbers in brackets show the numbers of failing tests.

When checkpointing is enabled, JSHRINK achieves 100% behavior preservation as expected. Disabling checkpointing leads to test failures in 4 projects only. Checkpointing does not cause significant loss in size reduction, because a single kind of transformation, *class hierarchy collapsing*, leads to most test failures (75 of 81) while contributing the least in size reduction (0.1% on average). The root cause is due to existing bugs in Soot. Soot throws runtime exceptions when rewriting some classfiles, which interferes our ability to update all classfiles that reference a merged class when collapsing class hierarchies. By simply reverting failure-inducing class collapsing transformations, JSHRINK avoids most test failures.

By contrast, JRed, Jax, and ProGuard cause test failures in 15, 17, and 11 projects respectively. Without checkpointing, only 81 of 5432 test cases fail after debloating using JSHRINK. This gives

**Table 3: Results of debloating the benchmark projects.**

| Application | Tests | Transformations | | | | Code Size Reduction | | | | | | Test Failures | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | MR | FR | CC | MI | JRed | Jax | ProGuard | TamiFlex | JShrink | JShrink-C | JRed | Jax | ProGuard | TamiFlex | JShrink | JShrink-C |
| jvm-tools | 102 | 1.7% | 0.6% | 0.0% | 2.0% | 2.2% | 5.2% | 12.2% | 4.2% | 4.2% | 4.2% | ✓ (0) | × (102) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| bukkit | 906 | 15.4% | 1.2% | 0.2% | 1.9% | 19.8% | 24.0% | 72.7% | 18.5% | 18.5% | 18.5% | × (906) | × (906) | × (3) | × (39) | ✓ (0) | ✓ (0) |
| qart4j | 1 | 42.2% | 3.7% | 0.2% | 0.9% | 58.0% | 64.2% | 84.8% | 46.8% | 46.8% | 46.8% | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| dubbokeeper | 1 | 13.8% | 1.5% | 0.2% | 1.9% | 17.2% | 20.9% | 73.1% | 17.3% | 17.3% | 17.3% | × (1) | × (1) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| frontend-maven-plugin | 6 | 18.7% | 1.6% | 0.2% | 2.0% | 24.3% | 28.2% | 65.8% | 22.4% | 22.4% | 22.4% | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| gson | 1050 | 0.3% | 0.8% | 0.0% | 4.4% | 0.4% | 5.8% | 2.3% | 5.5% | 5.5% | 5.5% | × (1) | × (1) | × (58) | ✓ (0) | ✓ (0) | ✓ (0) |
| disklrucache | 61 | 0.1% | 1.3% | 0.0% | 0.2% | 0.1% | 1.9% | 0% | 1.7% | 1.7% | 1.7% | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| retrofit1-okhttp3-client | 9 | 8.4% | 0.9% | 0.0% | 2.2% | 11.0% | 14.5% | 22.7% | 12.3% | 11.5% | 11.5% | × (9) | × (9) | × (3) | × (3) | ✓ (0) | ✓ (0) |
| rxrelay | 58 | 15.7% | 1.1% | 0.0% | 0.7% | 17.5% | 19.3% | 63.5% | 17.5% | 17.5% | 17.5% | × (28) | × (58) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| rxreplayingshare | 20 | 20.1% | 0.9% | 0.2% | 0.9% | 24.1% | 27.5% | 91.9% | 22.1% | 22.1% | 22.1% | × (20) | × (20) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| junit4 | 1081 | 1.7% | 0.5% | 0.1% | 4.8% | 2.3% | 8.0% | 9.0% | 6.5% | 6.8% | 1.4% | × (1081) | × (1081) | × (43) | × (17) | × (13) | ✓ (0) |
| http-request | 163 | 0.2% | 2.6% | 0.0% | 3.8% | 0.3% | 6.7% | 0.1% | 6.6% | 6.6% | 6.6% | ✓ (0) | ✓ (0) | × (15) | ✓ (0) | ✓ (0) | ✓ (0) |
| lanterna | 34 | 0.2% | 0.8% | 0.6% | 1.9% | 0.2% | 2.4% | 0% | 1.9% | 2.0% | 2.0% | ✓ (0) | × (34) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| java-apns | 111 | 13.8% | 1.3% | 0.3% | 3.4% | 16.0% | 21.9% | 34.4% | 18.9% | 18.9% | 18.9% | × (9) | × (107) | × (18) | ✓ (0) | ✓ (0) | ✓ (0) |
| mybatis-pagehelper | 106 | 20.1% | 1.4% | 0.1% | 2.3% | 25.5% | 28.6% | 65.0% | 24.7% | 23.9% | 21.6% | × (106) | × (106) | × (83) | × (100) | × (55) | ✓ (0) |
| algorithms | 493 | 0.0% | 0.3% | 0.0% | 5.1% | 0.0% | 5.6% | 3.8% | 5.5% | 5.5% | 5.5% | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| fragmentargs | 15 | 8.9% | 2.7% | 0.0% | 0.1% | 11.0% | 14.7% | 16.8% | 11.6% | 11.6% | 0.0% | × (4) | × (4) | × (4) | × (4) | × (4) | ✓ (0) |
| moshi | 835 | 0.2% | 0.0% | 0.0% | 0.0% | 0.2% | 0.3% | 58.2% | 0.2% | 0.2% | 0.2% | × (835) | × (835) | × (52) | ✓ (0) | ✓ (0) | ✓ (0) |
| tomighty | 26 | 16.5% | 1.5% | 0.1% | 2.2% | 20.7% | 24.7% | 56.4% | 20.2% | 20.1% | 20.1% | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| zt-zip | 121 | 5.4% | 2.4% | 0.6% | 2.9% | 6.4% | 13.3% | 16.4% | 11.3% | 11.3% | 11.3% | × (110) | × (110) | × (115) | ✓ (0) | ✓ (0) | ✓ (0) |
| gwt-cal | 92 | 16.5% | 0.7% | 0.1% | 0.3% | 19.4% | 20.8% | 31.6% | 17.5% | 17.5% | 17.5% | × (3) | × (3) | ✓ (0) | ✓ (0) | ✓ (0) | ✓ (0) |
| Java-Chronicle | 8 | 0.0% | 1.1% | 1.0% | 1.4% | 0.0% | 3.5% | 0.0% | 3.5% | 3.5% | 3.5% | ✓ (0) | ✓ (0) | ✓ (0) | × (8) | ✓ (0) | ✓ (0) |
| maven-config-processor-plugin | 77 | 25.4% | 3.2% | 0.3% | 1.0% | 31.5% | 35.3% | 82.0% | 29.8% | 29.8% | 29.8% | × (21) | × (21) | × (20) | ✓ (0) | ✓ (0) | ✓ (0) |
| jboss-logmanager | 42 | 11.1% | 0.5% | 0.04% | 1.9% | 11.7% | 14.3% | 17.0% | 26.2% | 13.6% | 13.6% | ✓ (0) | ✓ (0) | × (24) | ✓ (0) | ✓ (0) | ✓ (0) |
| autoLoadCache | 11 | 16.5% | 1.5% | 0.3% | 1.9% | 18.2% | 21.9% | Crash | 20.2% | 20.2% | 16.5% | × (10) | × (10) | Crash | × (7) | × (9) | ✓ (0) |
| tprofiler | 3 | 4.7% | 4.1% | 0.0% | 1.4% | 6.5% | 13.5% | Crash | 10.2% | 10.2% | 10.2% | ✓ (0) | ✓ (0) | Crash | ✓ (0) | ✓ (0) | ✓ (0) |
| **Total** | 5432 | – | – | – | – | – | – | – | – | – | – | 3174 | 3408 | 496 | 170 | 81 | 0 |
| **Mean** | 209 | 11.0% | 1.0% | 0.1% | 2.1% | 13.0% | 17.0% | 33.8% | 15.0% | 14.2% | 13.3% | – | – | – | – | – | – |
| **Median** | 60 | 9.9% | 1.23% | 0.1% | 1.9% | 11.4% | 14.6% | 19.8% | 14.8% | 12.6% | 12.5% | – | – | – | – | – | – |

JShrink a test pass rate of 98.5%, in comparison to 41.6%, 37.3%, and 91% by JRed, Jax, and ProGuard respectively. This indicates that incorporating dynamic profiling, type-dependency analysis, and those nuanced extensions are crucial to semantics preservation. The majority of test failures caused by JRed and Jax are due to fatal JVM `NoClassDefFoundError` and `ClassNotFoundException` verification errors that crash the entire test execution — for JRed, 10 of 26 projects fail with these fatal exceptions, while using Jax results in 13 projects failing fatally. For ProGuard, most test failures are caused by imprecise static analysis. Though ProGuard strives to handle Java reflection by statically analyzing string arguments passed into a predefined set of reflection APIs, such static analysis is neither accurate nor complete, which justifies our choice of augmenting static analysis with profiling for dynamic language features.

## 5.4 RQ3: Trade-offs

To further understand the trade-offs between debloating potential and semantic preservation, we vary entry points for JShrink's reachability analysis and compare with an alternative profiler called TamiFlex [9].

*Entry point analysis.* As discussed in Section 3, JShrink functions by running call-graph analysis on entry points. These entry points are a union of two sets: the set of dynamically accessed methods determined via runtime profiling, and the set of all public, `main`, and test methods, determined via static analysis. While the former is dependent on the test suite of each project, the latter can be set manually. E.g., a user of JShrink may determine that only the `main` entry point needsi to be processed as it is the only known entry point to the application. Such decisions may result in a smaller call-graph and thus increase the deboating potential of a target project. On the other hand, selecting fewer entry points can make the deloated software less robust without complete knowledge of used methods. For example, a method may be removed despite being used by the project via some unexplored entry point.

To understand this trade-off, we run JShrink on all our projects using the `main` method as an entry point, the public methods, and just the test methods alone as entry points. Table 4 shows the experiment results with the baseline where all such methods are considered as entry points. The size reduction is consistently larger when we select a subset of entry points to the reachability analysis. When targeting the test entry points, projects can be debloated by 36.6% more than our conservative baseline. Though, in every case where a subset of entry points are chosen, the number of test failures increases. While only 1.5% of all tests fail when targeting all entry points, this figure jumps to 3.4%, a 70% increase in test case failures, when selecting a subset.

We therefore conclude that the size reduction and robustness depend on what we choose as entry points. If preserving program semantics is a hard constraint, we suggest the conservative choice of setting all possible entry points.

**Table 4: Entry Point Analysis.**

| Entry Point | Size Reduction | Test Failures |
|---|---|---|
| Main, Test, & Public | 14.2% | 81 (1.5%) |
| Main Only | 18.6% | 186 (3.4%) |
| App Public Only | 18.3% | 157 (2.9%) |
| Test Only | 19.4% | 187 (3.4%) |

*Jmtrace vs. TamiFlex.* As discussed in Section 3.1, our native profiler, *Jmtrace*, uses JVMTI to instrument method bodies in any classes loaded in a JVM. Therefore, it can capture all dynamically invoked methods. By contrast, TamiFlex [9] only instruments a predefined set of reflection APIs and thus is considered more light-weight. The two TamiF. columns in Table 3 show the size reduction and test failures caused by the TamiFlex variant of JShrink. TamiFlex only identifies a subset of dynamic method calls captured by *Jmtrace* and thus should trim more unreachable methods. However, the size reduction improvement achieved by TamiFlex is trivial, only 0.06% on average. On the other hand, JShrink with TamiFlex breaks 52 more test cases in comparison to JShrink with *Jmtrace*.

*An in-depth inspection.* To investigate which extension aid in improving behavior preservation, we choose one project, `java-apns` for a thorough investigation into each failure. This is because there are a total of 3174 and 3408 test failures for JRed and Jax respectively; thus, it would be prohibitively time consuming to examine all test failures individually for all projects. `Java-apns` produces 107 test failures after Jax but was error-free when processed by JShrink. We manually examine and determine what extension was responsible for rectifying the failure. Incorporation of *Jmtrace* reduced test failures by 59%. The rest of the enhancements such as type dependency analysis all contribute to improving a test pass rate, but none was the dominant contributor. *This result indicates that handling dynamic language features is absolutely necessary, while each of the remaining enhancements contributes to behavior preservation.*

## 5.5 RQ4: Robustness

Finally, we assess the robustness of debloated software by running new tests not seen during dynamic profiling. We use 80% of the original test suite in each project for profiling and debloating. Then we use the remaining 20% as a *hold-out test set* for examining the robustness of each debloated project. In particular, for the three projects with one test case, we only use their tests for robustness assessment. The hold-out test set contains 42 test cases on average. JShrink does not cause any test failures in 22 out of 26 projects when running the debloated project on its hold-out test set. JShrink causes 3, 5, 45, and 1 test failures in the remaining four projects respectively—`retrofit1-okhttps3-client`, `junit`, `java-apns`, and `autoLoadCache`. This implies that, though there is a chance that unseen executions may cause runtime exceptions in debloated software, the chance is relatively low — only 4 out of 26 projects (15%) in our benchmark. This should be attributed to the design choice of using both static reachability analysis and dynamic profiling in JShrink. While dynamic profiling precisely captures all invoked methods in previous executions and handles dynamic features, static reachability analysis overapproximates other potential reachable code from given entry points, improving the robustness of debloated software compared to purely dynamic profiling alone.

# 6 DISCUSSION

This work presents the first systematic evaluation of bytecode debloating transformations using modern Java applications. We find that there is a lack of effort in ensuring and assessing behavior preservation by previous bytecode debloating techniques, which poses a big safety concern of adopting them in practice. Thus, we make significant engineering effort to improve behavior preservation, including augmenting static analysis with dynamic profiling, incorporating type dependency analysis to ensure type safety, and implementing nuanced extensions to account for new language features. By doing so, we improve the test pass rate by around 61%.

***Attack Surface Removal.*** Software debloating could also potentially remove security vulnerabilities in a program. To demonstrate the benefit of attack surface removal, we conduct a case study of a gadget-chain deserialization vulnerability in Java [5]. This vulnerability allows remote attackers to execute arbitrary commands by carefully crafting a payload of serialized Java classes (i.e., *gadget chains*). A gadget chain includes a "kick-off" gadget that is executed during or after deserialization, a "sink" gadget that executes arbitrary commands during instantiation, and other auxiliary gadgets that create a chain from the start gadget execution to end gadget. Frohoff et al. discovered a collection of 31 distinct gadget chains in JDK and popular Java libraries and present a proof-of-concept tool called `ysoserial` that automatically generates payloads of these gadget chains [6]. Based on these known gadget chains, we automatically scan libraries and classes and detect the presence of gadget chains that can be potentially exploited by remote attackers.

If a method or a class in a gadget chain is successfully removed, the gadget chain will be effectively removed and the attack surface shall cease to be a threat. Running our gadget-chain analysis, we detect two gadget chains in one project, `dubbokeeper` in our benchmark. Both gadget chains involve unsafe classes and methods in imported libraries from Spring Framework, a widely used web application framework in Java. These gadget chains have also been reported multiple times as security vulnerabilities, e.g., CVE-2017-8045, CVE-2017-3203, CVE-2016-2173. After applying JShrink, both gadgets in `dubbokeeper` are removed. Hence exploiting the same gadget chain in `dubbokeeper` will only lead to `ClassNotFoundException`, rather than arbitrary code execution after debloating. We therefore demonstrate JShrink is effective at reducing these forms of attack surfaces, thereby improving application security.

***Threats to validity.*** While replicating bytecode transformations described in Jax [58], our implementation is different from [58] is two aspects. First, when merging two classes, the original Jax algorithm requires that the number of reachable field members in the new class after merging does not increase, in order to ensure that the new class did not consume more memory when instantiated. We ignore this constraint in JShrink, because our main goal is to reduce code size not memory consumption. Second, method inlining in Jax is originally implemented by simply adding a `final` modifier to an inlinable method so that the just-in-time (JIT) compiler in JVM can inline the method. However, this is tricky as JIT compilers also apply their own heuristics to decide whether a method, though declared as `final`, could be inlined. So we implement our own inlining transformation using Soot's APIs. This potentially produces better results than the original implementation in [58].

In Section 5, we reimplement Jax and JRed by adapting JShrink based on their papers. The size reduction numbers reported from their original Jax and JRed papers are higher, at 48.7% and 44.5% respectively. This is because we conservatively choose all public, `main`,

and test methods as entry points, while their original experiment used `main` methods as the entry points only.

***Limitation.*** Our goal is to assess size reduction potential based on both static analysis and dynamic profiling using *developer-written tests*. We acknowledge that developer-written tests may suffer from low-coverage issues and thus cannot represent possible execution scenarios that may be encountered in the future. To investigate this issue, we examine how robust a debloated project is to unseen executions in a hold-out test set in Section 5.5. The result shows that though unseen executions indeed cause runtime exceptions in some projects, the chance of running into such issues is relatively low (15%). This is because static reachability analysis and dynamic profiling synergistically work toghether to handle unseen executions–*dynamic profiling already provides good hints of dynamically invoked method even with an incomplete set of test cases, and static reachability analysis overapproximates reachable code from all possible entry points.* The solution to reducing this rate of failure is to increase test case coverage by either by manually adding tests, or via automated regression test generation tools, such as EvoSuite [21].

## 7 RELATED WORK

***Code Bloat.*** Code size reduction is an important development activity in areas such as networking and embedded systems. A large body of work exists on code compression [14, 17, 30, 33, 34] and code compaction [16, 62, 63] to reduce the size of binary code for efficient executions on embedded hardware with limited memory. We refer interested readers to Beszédes et al. [8] for a detailed survey. Program slicing [26, 46, 53, 54, 57] is a dataflow-based static technique that computes, from a given *seed*, a subset of statements that can still form a valid and executable program. Slicing reduces code size by computing a dependence graph and preserving only the statements that are directly or transitively reachable from the seed on the graph. Fine-grained static slicing is known to have limitations due to imprecision of heap modeling and pointer handling and thus does not work well for large-scale applications with pointers, reflection, and dynamic class loading. Soto-Valero et al. investigated library dependency debloat in maven projects. Compared with JShrink, their analysis is coarse-grained at the library dependency level [52]. Therefore, their debloating technique can only remove unused libraries, rather than unused code within a library. Furthermore, their analysis is purely static and does not account for dynamic features, which GitHub developers reported as an important concern in their qualitative study.

The past two years have seen a proliferation of debloating techniques [19, 25, 43, 44, 49, 64] designed for various domains, including JavaScript programs [64], application containers (e.g., docker) [19], or native C programs [43, 44]. These range from static analysis [19, 49] to load/runtime techniques [43] and machine learning [25]. However, none target modern Java, notoriously different from native programs in terms of memory management or dynamic method invocation. This paper revisits and extends existing bytecode transformation techniques, quantifies debloating potential, and checks behavior preservation with real world tests.

***Delta Debugging.*** Given a test oracle, delta-debugging based techniques can repeatedly split the original program into different sub-programs and re-check the test oracle to produce a debloated program [29, 45, 56, 73]. For example, JReduce [29] partitions the original program into transitive closures based on class-level dependencies and isolates a debloated program that still passes the test. Chisel [25] uses reinforcement learning to reduce the number of search iterations during delta debugging. While these approaches ensure behavior preservation of debloated software by repeatedly running existing tests on each intermediate program, they suffer from two limitations—(1) the resulting debloated software may not retain any functionality beyond test-exercised code, simply reflecting test coverage, and (2) the debloated software cannot be easily configured to retain code statically reachable from public APIs or main method entries, since designing such oracle would be exactly the same task we undertook in JShrink.

***Runtime Bloat.*** Researchers have proposed a range of dynamic techniques that look for inefficiencies in data structure usage [41, 66, 67], object lifetime patterns [68], or reference copy chains [69, 72]. Such runtime bloat work is orthogonal to this work that removes code bloat via static bytecode transformations.

## 8 CONCLUSION

Software debloating is a long standing problem. Some even consider this problem to have been solved 20 years ago through static reachability-analysis based code transformation. We therefore set out to extend and rigorously evaluate software debloating transformations in the context of modern Java. Unlike previous research, we handled dynamic language features, ensured type safety, and took measures to pass the JVMs bytecode verification checks. We found that prior work falls short of *behavior preservation*, meaning debloated software no longer passes the same tests, with a test failure rate of up to 62.7%. Such lack of behavior preservation would make it impossible to adopt debloating techniques in practice, as no one would like to remove unused code at the cost of breaking a majority of existing tests.

The technical contributions that we made are significant, and our study shows that these extensions, which we embody in a tool called JShrink, are worthwhile and necessary to improve the test passing rate of prior work. With checkpointing, JShrink is able to provide 100% behavior preservation guarantees with marginal size reduction loss (0.9%). ProGuard, a popular software debloating tool, reduced software size by almost double but also resulted in 6X more test failures compared to JShrink. To the best of our knowledge, we are the first that systematically quantify size reduction, behavior preservation, and the benefit of dynamic profiling in software debloating. To support the open-science policy, we present our source code and additional resources necessary to replicate our results at https://doi.org/10.6084/m9.figshare.12435542.

## REFERENCES

[1] [n.d.]. Cobertura: A code coverage utility for Java. https://cobertura.github.io/cobertura. Accessed: 2020-02-16.
[2] [n.d.]. ONR BAA Announcement # N00014-17-S-B010. https://www.onr.navy.mil/-/media/Files/Funding-Announcements/BAA/2017/N00014-17-S-B010.ashx. Accessed: 2019-05-13.
[3] [n.d.]. ProGuard Bug #767: A misjudgement exception occurs while preverifying. https://sourceforge.net/p/proguard/bugs/767. Accessed: 2020-04-04.
[4] [n.d.]. ProGuard: Java and Android Apps Optimizer. https://www.guardsquare.com/en/products/proguard. Accessed: 2019-12-13.

[5] [n.d.]. Why The Java Deserialization Bug Is A Big Deal. Available from www.darkreading.com. https://www.darkreading.com/informationweek-home/why-the-java-deserialization-bug-is-a-big-deal/d/d-id/1323237

[6] [n.d.]. ysoserial: a proof-of-concept tool for generating payloads that exploit unsafe Java object deserialization. https://github.com/frohoff/ysoserial. Accessed: 2019-05-10.

[7] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. *ACM Sigplan Notices* 31, 10 (1996), 324–341.

[8] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. 2003. Survey of Code-size Reduction Methods. *ACM Computer Survey* 35, 3 (2003), 223–267.

[9] 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 2011 International Conference on Software Engineering — ICSE '11*. ACM, 241–250.

[10] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*.

[11] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 2008. When good instructions go bad: Generalizing return-oriented programming to RISC. In *Proceedings of the 2008 ACM conference on Computer and Communications Security — CCS '08*. ACM, 27–38.

[12] Jeffrey C. Carver, Natalia Juristo, Maria Teresa Baldassarre, and Sira Vegas. 2014. Replications of Software Engineering Experiments. *Empirical Softw. Engg.* 19, 2 (April 2014), 267–276. https://doi.org/10.1007/s10664-013-9290-8

[13] Ahmet Celik, Alex Knaust, Aleksandar Milicevic, and Milos Gligoric. 2016. Build system with lazy retrieval for Java projects. In *Proceedings of the 2016 SIGSOFT International Symposium on Foundations of Software Engineering — FSE '16*. ACM, 643–654.

[14] Keith D. Cooper and Nathaniel McIntosh. 1999. Enhanced Code Compression for Embedded RISC Processors. In *Proceedings of the 1999 Conference on Programming Language Design and Implementation — PLDI '99*. ACM, 139–149.

[15] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 1995 European Conference on Object-Oriented Programming — ECOOP '95*.

[16] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. 2000. Compiler Techniques for Code Compaction. *Transactions on Programming Languages and Systems* 22, 2 (2000), 378–415.

[17] Jens Ernst, William Evans, Christopher W. Fraser, Todd A. Proebsting, and Steven Lucco. 1997. Code Compression. In *Proceedings of the 1997 Conference on Programming Language Design and Implementation — PLDI '97*. ACM, 358–365.

[18] Robert Feldt, Tim Menzies, and Thomas Zimmermann. 2018. The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018), ROSE Festival 2018 Recognizing and Rewarding Open Science in Software Engineering. https://2018.fseconference.org/track/rosefest-2018.

[19] Kostas Ferles, Valentin Wüstholz, Maria Christakis, and Isil Dillig. 2017. Failure-directed Program Trimming. In *Proceedings of the 2017 Symposium on the Foundations of Software Engineering — FSE '17*. ACM, 174–185.

[20] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. 2018. Static Analysis of Java Dynamic Proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 209–220. https://doi.org/10.1145/3213846.3213864

[21] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 2011 Conference on Foundations of Software Engineering — FSE '11*. ACM, 416–419.

[22] Neal Glew and Jens Palsberg. 2002. Type-safe method inlining. In *Proceedings of the European Conference on Object-Oriented Programming — ECOOP '02*. Springer, 525–544.

[23] Neal Glew and Jens Palsberg. 2005. Method Inlining, Dynamic Class Loading, and Type Soundness. *Journal of Object Technology* 4, 8 (2005), 33–53.

[24] David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. 1997. Call graph construction in object-oriented languages. *ACM SIGPLAN Notices* 32, 10 (1997), 108–124.

[25] 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)*. ACM, New York, NY, USA, 380–394. https://doi.org/10.1145/3243734.3243838

[26] S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the Conference on Programming Language Design and Implementation — PLDI '88*. ACM, 35–46.

[27] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android application redundancy customization based on static analysis. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 189–199.

[28] Yufei Jiang, Dinghao Wu, and Peng Liu. 2016. JRed: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 2016 Computer Software and Applications Conference — COMPSAC '16*. IEEE, 12–21.

[29] 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)*. ACM, New York, NY, USA, 556–566. https://doi.org/10.1145/3338906.3338956

[30] Darko Kirovski, Johnson Kin, and William H. Mangione-Smith. 1997. Procedure Based Program Compression. In *Proceedings of the 1997 International Symposium on Microarchitecture — Micro '97*. ACM, 204–213.

[31] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for Static Analysis of Java Reflection: Literature Review and Empirical Study. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 507–518. https://doi.org/10.1109/ICSE.2017.53

[32] Jason Landsborough, Stephen Harding, and Sunny Fugate. 2015. Removing the kitchen sink from software. In *Proceedings of the 2015 Genetic and Evolutionary Computation Conference Companion — GECCO Companion '15*. ACM, 833–838.

[33] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. 1999. Evaluation of a High Performance Code Compression Method. In *Proceedings of the 1999 International Symposium on Microarchitecture — Micro '99*. 93–102.

[34] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. 2000. Code Compression for Low Power Embedded System Design. In *Proceedings of the 2000 Annual Design Automation Conference — DAC '00*. 294–299.

[35] Ondrej Lhoták. 2002. Spark: A flexible points-to analysis framework for Java. (2002).

[36] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. 2014. Self-inferencing Reflection Resolution for Java. In *ECOOP 2014 – Object-Oriented Programming*, Richard Jones (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–53.

[37] Benjamin Livshits, Dimitrios Vardoulakis, Manu Sridharan, Yannis Smaragdakis, OndÅŽej LhotÃąk, JosÃĪ Amaral, Bor-Yuh Evan Chang, Samuel Guyer, Uday Khedker, and Anders MÃ¿ller. 2015. In Defense of Soundiness: A Manifesto. *Commun. ACM* 58 (01 2015), 44–46. https://doi.org/10.1145/2644805

[38] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (Tsukuba, Japan) *(APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 139–160. https://doi.org/10.1007/11575467_11

[39] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. 2010. *Apache Maven*. Alpha Press.

[40] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. [n.d.]. Four Trends Leading to Java Runtime Bloat. *IEEE Software* 27, 1 ([n. d.]), 56–63.

[41] Nick Mitchell and Gary Sevitsky. 2007. The Causes of Bloat, the Limits of Health. *Proceedings of the 2007 Conference on Object-Oriented Programming Systems, Languages, and Applications — OOPSLA '07* (2007), 245–260.

[42] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. {RAZOR}: A Framework for Post-deployment Software Debloating. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1733–1750.

[43] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating software through piece-wise compilation and loading. In *Proceedings of the 2018 USENIX Security Symposium — USENIX Security '18*. 869–886.

[44] Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick McDaniel. 2017. Cimplifier: Automatically Debloating Containers. In *Proceedings of the 2017 Symposium on the Foundations of Software Engineering — FSE '17*. ACM, 476–486.

[45] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. 2012. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Notices*, Vol. 47. ACM, 335–346.

[46] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. 1994. Speeding Up Slicing. In *Proceedings of the 1994 Symposium on Foundations of Software Engineering — FSE '94*. ACM, 11–20.

[47] IEEE Security and Privacy. 2019. A list of CS conferences with "SoK" tracks. https://oaklandsok.github.io/others/.

[48] Marc Shapiro and Susan Horwitz. 1997. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1–14.

[49] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. 2018. TRIMMER: Application Specialization for Code Debloating. In *Proceedings of the 2018 International Conference on Automated Software Engineering — ASE '18*. ACM, 329–339.

[50] Olin Shivers. 1991. *Control-flow analysis of higher-order languages*. Ph.D. Dissertation. Citeseer.

[51] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In *APLAS*.

[52] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2020. A Comprehensive Study of Bloated Dependencies in the Maven Ecosystem. *arXiv*

*preprint arXiv:2001.07808* (2020).

[53] Manu Sridharan, Stephen J. Fink, and Rastislav Bodik. 2007. Thin Slicing. In *Proceedings of the Conference on Programming Language Design and Implementation — PLDI '07*. ACM, 112–122.

[54] Venkatesh Srinivasan and Thomas Reps. 2016. An Improved Algorithm for Slicing Machine Code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications — OOPSLA '16*. ACM, 378–393.

[55] 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 *Asian Symposium on Programming Languages and Systems*. Springer, 69–88.

[56] Chengnian Sun, Yuanbo Li, Qirun Zhang, Tianxiao Gu, and Zhendong Su. 2018. Perses: syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 361–371.

[57] Frank Tip. 1994. *A Survey of Program Slicing Techniques*. Technical Report. Amsterdam, The Netherlands, The Netherlands.

[58] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. 1999. Practical Experience with an Application Extractor for Java. In *Proceedings of the 1999 Conference on Object-oriented Programming, Systems, Languages, and Applications — OOPSLA '99*. ACM, 292–305.

[59] Frank Tip, Peter F Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. 2002. Practical extraction techniques for Java. *ACM Transactions on Programming Languages and Systems — TOPLAS '02* 24, 6 (2002), 625–666.

[60] Mohsen Vakilian, Raluca Sauciuc, J David Morgenthaler, and Vahab Mirrokni. 2015. Automated decomposition of build targets. In *Proceedings of the 2015 International Conference on Software Engineering — ICSE '15*. IEEE Press, 123–133.

[61] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot — A Java Bytecode Optimization Framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research — CASCON '99*. IBM Press, 13–23.

[62] R. van de Wiel, L. Augusteijn, A. Bink, and P. Hoogendijk. 2001. Code compaction: Reducing memory cost of embedded software. Philips White Paper.

[63] R. van de Wiel and P. Hoogendijk. 2001. Belt-tightening in software. Philips Res. Passw. Mag.. , 16–19 pages.

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

[65] Dongpeng Xu, Jiang Ming, Yu Fu, and Dinghao Wu. 2018. VMHunt: A verifiable approach to partially-virtualized binary code simplification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 442–458.

[66] Guoqing Xu. 2012. Finding Reusable Data Structures. In *Proceedings of the 2012 Conference on Object-Oriented Programming Systems, Languages, and Applications — OOPSLA '12*. ACM, 1017–1034.

[67] Guoqing Xu. 2013. CoCo: Sound and Adaptive Replacement of Java Collections. In *Proceedings of the 2013 European Conference on Object-Oriented Programming — ECOOP '13*. Springer, 1–26.

[68] Guoqing Xu. 2013. Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-world Programs. In *Proceedings of the 2013 Conference on Object Oriented Programming Systems Languages and Applications — OOPSLA '13*. ACM, 111–130.

[69] Guoqing Xu, Matthew Arnold, Nick Mitchell, and Atanas Rountev añd Gary Sevitsky. 2009. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the 2009 Conference on Programming Language Design and Implementation — PLDI '09*. ACM, 419–430.

[70] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Edith Schonberg, and Gary S evitsky. 2010. Finding Low-Utility Data Structures. In *Proceedings of the 2010 Conference on Programming Language Design and Implementation — PLDI '10*. ACM, 174–186.

[71] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 2010. Software bloat analysis: finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *Proceedings of the 2010 workshop on Future of Software Engineering Research — FoSER '10*. ACM, 421–426.

[72] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Uncovering Performance Problems in Java Applications with Reference Propagation Profiling. In *Proceedings of the International Conference on Software Engineering — ICSE '12*. IEEE, 134–144.

[73] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.* 28, 2 (Feb. 2002), 183–200. https://doi.org/10.1109/32.988498