UCLA
UCLA Electronic Theses and Dissertations

Title
JShrink: Debloating Modern Java Applications

Permalink
https://escholarship.org/uc/item/7vm6q2nm

Author
Arora, Jaspreet Singh

Publication Date
2020

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA

Los Angeles

JShrink: Debloating Modern Java Applications

A thesis submitted in partial satisfaction
of the requirements for the degree Master of Science
in Computer Science

by

Jaspreet Singh Arora

2020

© Copyright by

Jaspreet Singh Arora

2020

ABSTRACT OF THE THESIS

JShrink: Debloating Modern Java Applications

by

Jaspreet Singh Arora
Master of Science in Computer Science
University of California, Los Angeles, 2020
Professor Miryung Kim, Chair

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 of software bloat results in software consuming more resources than it otherwise needs to. Automation of effective debloating 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. JShrink augments traditional static reachability analysis with dynamic profiling and type dependency analysis, and renovates existing byte-code transformations to perform effective debloating. 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 reflection and dynamic language features is 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. We enhance JShrink with the checkpointing feature to ensure 100% behaviour preservation with minimal loss in code size reduction(0.9% om average), to make it a practical solution for balancing semantic preservation and code size reduction benefits.

The thesis of Jaspreet Singh Arora is approved.

Jens Palsberg

Harry Guoqing Xu

Miryung Kim, Committee Chair

University of California, Los Angeles

2020

TABLE OF CONTENTS

List of Figures . . . . **viii**

List of Tables . . . . **x**

Acknowledgments . . . . **xi**

1 Introduction **1**

2 Related Work **6**

2.1 Code Bloat 6

2.2 Static Bytecode Debloating 7

2.3 Dynamic Feature Analysis for Java 7

2.4 Delta Debugging 8

2.5 Runtime Bloat 9

3 Background **10**

3.1 Scope - Java Bytecode 10

3.2 Motivation for Modernizing Software Debloating 11

3.3 Dynamic Profiling for Java 11

3.3.1 Customized JVM 12

3.3.2 Heap/Thread dumps 13

3.3.3 Java Virtual Machine Tool Interface (JVM TI) 13

4 Approach **19**

4.1 Static Reachability Analysis 19

4.2 Dynamic Reachability Analysis . . . 20
4.2.1 Definition of Scope . . . 21
4.2.2 JMtrace . . . 25
4.2.3 Integration of JMtrace with JShrink . . . 29
4.3 Type Dependency Analysis . . . 32
4.4 Bytecode Debloating Transformations . . . 33
4.4.1 Unused Method Removal . . . 33
4.4.2 Unused Field Removal . . . 34
4.4.3 Method Inlining . . . 34
4.4.4 Class Hierarchy Collapsing . . . 34
4.5 Checkpointing . . . 35
4.5.1 Checkpoint . . . 36
4.5.2 Backup Service . . . 37
4.5.3 Workflow . . . 37
4.6 Implementation and Nuanced Extensions . . . 38
5 Evaluation & Results . . . 42
5.1 JShrink Evaluation . . . 42
5.1.1 Benchmark . . . 42
5.1.2 Experiment Setup and Baselines . . . 44
5.1.3 Experiments . . . 45
5.1.4 RQ1: Code Size Reduction . . . 45
5.1.5 RQ2: Semantic Preservation . . . 47
5.1.6 RQ3: Trade-offs . . . 50
5.1.7 RQ4: Software Debloating Robustness . . . 53

5.2 JMtrace Evaluation 53
5.2.1 Benchmark 55
5.2.2 Results 63
6 Conclusion **64**
A List of Benchmark Projects **65**
B JVM TI Bytecode Instrumentation vs Callback Tests **66**
C Source Code for Dynamic Features Test Project **67**
References **73**

LIST OF FIGURES

3.1 Running a JVM TI agent with a JVM [40] 14
3.2 Complete list of JVMTI subscribable events [40] 15
3.3 Example of JVMTI agent subscribed to the Class file load event 17
3.4 Example of JVMTI agent instrumenting a class in the Class file load callback . 18

4.1 An example of Java reflection API 21
4.2 An example of Java Custom Classloading 22
4.3 An example of Java Lambda Expression 22
4.4 An example of Java Dynamic Proxy 23
4.5 An example of invocation for a deserialized class 23
4.6 An example of invocation for a JNI method 24
4.7 An example of the use of `sun.misc.Unsafe` to change reference at runtime . . . 24
4.8 An Example Java class 26
4.9 The global data structures in the native agent 27
4.10 Native agent code snippet that logs the profiling information from global data structures 28
4.11 Sample log file generated by JMtrace 28
4.12 Profiling log file format 28
4.13 The entry and exit methods in the Mtrace class 29
4.14 Example call graph augmentation through dynamic analysis. 30
4.15 Class Diagram for (a)`Checkpoint` (b)`BackupService` 36
4.16 Code snippet illutsrating checkpointing workflow 38
4.17 An example of co-variant return types in a class hierarchy 39

4.18 An example of an unsafe inheritance hierarchy for class collapsing 40
4.19 A code snippet demonstrating unsafe inheritance hierarchy for class collapsing . 41

5.1 A failure induced by class reference updating. 48
5.2 Structure for example anonymous class implementing a `generic` abstract class . 50
5.3 Dynamically compiled class in test cases. 51

C.1 DynamicProxy.java 67
C.2 ProxiedClass.java 67
C.3 ConcreteA.java 67
C.4 ConcreteB.java 68
C.5 OtherClass.java 68
C.6 InterfaceClass.java 69
C.7 InterfaceWithInnerClass.java 69
C.8 InterfaceWithInnerClass.java 69
C.9 ConcreteWithInnerClass.java 69
C.10 CustomClassLoaderTarget.java 70
C.11 CustomClassLoader.java 70
C.12 Main.java 71

LIST OF TABLES

4.1 Capability of Handling Different Dynamic Features 31

5.1 Project statistics 43

5.2 Results of debloating the benchmark projects. 46

5.3 Entry Point Analysis. 52

5.4 Results of held-out tests for the benchmark projects. 54

5.5 Comparison of dynamic feature coverage for Li Sui et al. benchmark [53] 62

5.6 JShrink execution time breakdown. 63

A.1 List of Benchmark Projects 65

B.1 Performance comparison of JVM TI callback agent vs instrumentation agent . . 66

ACKNOWLEDGMENTS

I would like to thank my advisor, Miryung Kim, for endowing me with the opportunity to contribute to this research effort and for her support and guidance. I am grateful to my research mentors, Tianyi Zhang and Bobby R. Bruce, for leading the project and for their patience, help, and counsel. This thesis is part of the larger research effort of the JShrink tool and shares text with a research paper submission co-authored by Bobby R. Bruce, Tianyi Zhang, Jaspreet Arora, Guoqing Harry Xu, and Miryung Kim, with much of the text written by co-authors. The choice to use the text directly from the paper is made to increase the coherence of the narrative with explicit acknowledgment that the text is reused in this MS thesis. My contributions to JShrink comprise of the implementation and performance analysis of JMTrace, the profiling tool used for dynamic reachability analysis in JShrink, the development of the checkpoint feature for complete semantic preservation, and other collaborative efforts in the implementation and running experiments as covered in Sections 2.5, 3.2, 4.2.2, 4.2.3, 4.5, 5.1.5 and 5.2.

CHAPTER 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 [42, 21, 55, 57, 41, 25, 59, 48, 29, 63, 44, 64].

Prior work on code size reduction focuses primarily on C/C++ binaries [42, 21, 48, 29, 41, 64], 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 [57] in the late 1990s that proposes various bytecode transformations for software debloating, which have since

been utilized by other researchers [20, 8, 25]. In surveying the available 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 [28, 35]. 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 [37], which provides a standardized interface for obtaining library dependencies and regression testing, and (4) compatibility with the underlying bytecode analysis framework, Soot [60]. 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 What Java bytecode size reductions are achievable when applying JShrink's 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 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 *field removal* (1.5% on average), *method inliner* (2.1% 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 [57], JRed [25], and ProGuard [20] 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 the behavior preservation of debloated software, 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 stronger behavior preservation guarantees. In order to achieve 100%

behavior preservation 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 strategy incurs a marginal loss in size reduction (0.9% on average), we believe checkpointing to be a practical solution for balancing sematic 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 develop JMTrace, a native profiling agent using JVM TI API [40], which captures the use of dynamic features in Java code and augments static reachability analysis in JShrink. We use a benchmark dataset by Sui et al. [53] to compare the effectiveness of JMTrace with Tamiflex [5] in capturing the use of dynamic features.

- We demonstrate the necessity of handling dynamic features and ensuring type safety. JShrink successfully removes 98% and 83% of test case failures incurred by Jax [57] and ProGuard [20].

- 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 passes 98.0% of existing tests without checkpointing, and 100% of the tests with checkpointing.

- 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.

- We extend JShrink with an optional checkpointing feature to ensures complete semantic preservation at the cost of code size increase to enable practical use of JShrink in the imperfect world of being unable to capture any call dependencies.

The main research contribution of this work is the development of an end to end framework for debloating of modern Java applications, as well as the systematization of the community's knowledge of Java debloating in the modern era. The seminal works of Java debloating [57, 25] are extended in a new context to account for dynamic Java language features and to manage the tradeoff between size reduction and debloating safety. We make publicly available the replication data set, an automated infrastructure for constructing a modern Java application benchmark with rigorous filtering criteria, replicated implementation of prior work, and JShrink[1]. The full list of Java applications used is listed in Appendix A.

[1] `https://github.com/tianyi-zhang/call-graph-analysis/`

CHAPTER 2

Related Work

2.1 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 [30, 27, 9, 12, 31] or code compaction [11, 61, 62] to reduce the size of binary code for efficient executions on embedded hardware with limited memory. We refer the interested reader to Beszédes et al. [4] for a detailed survey and comparison. Program slicing [56, 46, 52, 22, 51] 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.

The past few years have seen a proliferation of debloating techniques [43, 42, 63, 13, 48, 21] designed for various domains, including JavaScript programs [63], application containers (e.g., docker) [13], or native C programs [43, 42]. These range from static analysis [48, 13] to load/runtime techniques [42] and machine learning [21]. However, none targets modern Java, notoriously different from native programs in terms of memory management or dynamic method invocation.

Existing debloating techniques for Java only assess code reduction and performance improvement achieved by different kinds of bytecode transformations [58, 25, 20, 24], while ignoring the correctness of reduced programs by running existing test cases. Furthermore,

these techniques only perform static call graph analysis to approximate used code, which are incomplete in the presence of various dynamic language features discussed later. This makes test failures inevitable, as dynamically invoked code could be removed by debloating. In this thesis, we report the results of taking a *profile-augmented static debloating* approach—we augment static reachability analysis with dynamic reachability analysis using existing tests; we revisit and augment existing bytecode transformation techniques; and we check behavior preservation with real world tests.

2.2 Static Bytecode Debloating

In the late 1990s, Tip et al. developed Jax, which included, the most comprehensive set of transformations to reduce Java bytecode so far, including *method removal*, *field removal*, *method inlining*, *class hierarchy transformation*, and *name compression* [57]. They later introduced two more transformations, *class attribute removal* and *constant pool compression* in their 2002 journal paper [58]. Recent techniques are based on a subset of these transformations, partially, to debloat new types of applications, e.g., Android [24] and Maven libraries in continuous integration [8]. JRed [25] and RedDroid [24] only support the *method removal* and *class removal* transformations, while Molly [8] supports *field removal* as well. These above mentioned techniques are either *outdated* or *not publicly available*. Furthermore, their evaluations mainly focus on code size reduction without systematically quantifying the degree to which debloated software preserves semantics. However, behavior preservation is crucial for these techniques to be adopted in practice.

2.3 Dynamic Feature Analysis for Java

Dynamic features are highly challenging to model through pure static analysis. Livshits et al. [36] pioneered the static analysis of dynamic features in 2005 using points to analysis to resolve dynamic method invocation targets statically. There have been other attempts mostly focused on specific dynamic features such as reflection [33][50], dynamic proxy [14],

etc. Liu et al. [34] proposed a novel technique to apply reflection-oriented slicing to resolve reflection call sites with more precision. The paper by Landman et al. [28] cites 24 different techniques and tools developed to deal with reflection. Most static analysis tools thus focus on generation of balanced "sound" call graphs, tolerating and encouraging some level of unsoundness, by under approximation of these features to keep the analysis usable and scalable [35].

Due to limitations of static analysis, a lot of research focuses on a hybrid analysis by adding dynamic analysis to augment static analysis tools. Tamiflex [5] uses java agent instrumentation to log reflective calls through a play out agent. Another approach is analysis of heap snapshots to capture method invocations and object initializations [17, 18]. Sui et al. [54] analyzed stack traces from Github issues and Stack Overflow forums to augment static call graphs. Features such as reflection resolution are still challenging for static analysis [28] and usually requires dynamic analysis [5]. Other challenges include tracking externally loaded classes through custom class loaders and serialization. Sui et al. [53] and Reif et al. [45] have categorized the dynamic features in Java and have provided micro-benchmarks of these features to compare the coverage of these features by various call graph construction tools. Reif et al's [45] evaluated the two most widely used Java analysis frameworks, Soot [60] and WALA [23], and found that neither provided complete coverage for all dynamic features in Java. In Chapter 3, we systematically define a list of dynamic features in Java uncovered in our literature survey, and report the coverage of these dynamic features through by JShrink.

2.4 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 [72, 44, 55, 26]. For example, JReduce [26] partitions the original program into transitive closures based on class-level dependencies and isolates a debloated program that still passes the test. Chisel [21] 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. In Chapter 5, we quantify the value of static reachability analysis for debloating. We exclude a subset of test cases during profiling and observe increase in test errors in only 4 out of 26 projects in our benchmark after debloating.

2.5 Runtime Bloat

Researchers have proposed a range of dynamic techniques that look for inefficiencies in data structure usage [39, 65, 66], object lifetime patterns [67], or reference copy chains [68, 71]. Such runtime bloat work is orthogonal to this work that removes code bloat via static bytecode transformations.

CHAPTER 3

Background

3.1 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 surface for a wide spectrum of software applications, including JavaScript programs [63], native applications [42], and Docker containers [43].

In this thesis, we focus on *code size reduction* as opposed to runtime memory bloat that was the target of a large body of prior work [68, 38, 70, 66, 69]. 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 [7]), which can be exploited by remote attackers to execute code segments that could not have been reached otherwise.

Second, many recent techniques [42, 21, 48, 41] 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.

3.2 Motivation for Modernizing Software Debloating

Java offers a number of dynamic features widely used in real-world Java programs [28]—reflection, dynamic class loading, dynamic proxy, etc., which are highly challenging to model through pure static analysis. 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 [28]. They find that static analysis is inherently incomplete and reflection cannot be ignored for 78% of projects. This finding motivates our effort to extend, replicate, and evaluate the safety of debloating techniques in the context of dynamic language features. In Section 5.1.5, we quantify this benefit of handling dynamic features—without explicit profiling, debloated software based on static analysis alone would fail 3327 more tests in 26 projects.

3.3 Dynamic Profiling for Java

Dynamic profiling for Java can be performed through a variety of methods. The main methodology is the extraction of runtime information like the executed methods, allocated objects and loaded classes during the execution of a Java program. We conducted proofs of concept for the following methods of dynamic analysis to decide the most effective profiling approach for JShrink.

3.3.1 Customized JVM

We used the `OpenJDK 8` JVM[1] and modified the internal code to create a customized JVM for our investigation. The customization focused on the alteration of the *hotspot* part of the JVM, which controls the flow and execution of a Java application inside the JVM. The altered code profiles the methods, classes and fields as they are created, invoked and destroyed in the JVM during the execution of a Java application. This involves modification of the flow of method invocation and object instantiation inside the JVM. The following methods must be altered to profile the dynamic dependencies of a Java application.

1. Method resolution function is the point of entry for method invocation that resolves a method invocation to actual method bytecode in memory.
2. Method execution function that executes the bytecode with parameters.
3. Invokedynamic method execution function that executes dynamic method bytecode with parameters.
4. Class loading function that loads a class file into memory on reference.
5. Object reference resolution function that resolves reference to a class type in memory.
6. Object memory allocation function that allocates memory for a class object.

The methods responsible for method resolution and method execution are modified to include instructions to log the class and signature of the resolved methods as well as the signature of the callee methods before execution. This information is then processed to extract the callee-caller relationships and thus a rudimentary call graph for a particular run of a Java program. The methods determined for method resolution is responsible for method resolution in all cases except static methods and reflection.

This approach has the least overhead with the most comprehensive coverage, but was deemed unfit for our use case, since it would require the execution of a Java program with a

[1]`http://hg.openjdk.java.net/jdk8/jdk8/hotspot/file/`

custom JVM, making it difficult to deploy and adopt. This requires the installation of this custom JVM along with JShrink, which is cumbersome for the users of the tool. In addition, the maintenance of a custom JVM is laborious, as the exact JVM methods modified might be changed or updated as Java evolves and the documentation for the `OpenJDK` code is not always complete.

3.3.2 Heap/Thread dumps

This method involves extracting heap dumps or thread dumps during program execution to log stack frames at regular intervals. This is a common method used for profiling, especially to profile and determine resource heavy methods in code or for recognizing bottlenecks in a Java program [17, 18]. There are built-in Java command line tools that can be leveraged for extracting thread dumps and heap dumps such as `jcmd, jfr, jstack, jmx, hprof`. A proof of concept tool extracts heap dumps from `hprof` and `jcmd` at intervals of 1ms to an external log file. These dumps are then processed to interpret the stack frames and to extract callee-caller relationships among the methods. The tool is highly contingent on the interval time used to extract information. Even with the shortest possible interval time, there is a chance that a method would be missing from the thread dumps if it takes less time for execution as compared to the interval time. In addition, the generated log files are large in size even for smaller programs and contain a lot of redundant information. It wastes a lot of resources and time to process these files to extract the required information.

3.3.3 Java Virtual Machine Tool Interface (JVM TI)

The Java Virtual Machine Tool Interface is a native programming interface provided as part of the Java Virtual Machine(JVM) and provides an interface for creating monitoring and development tools to track and modify the state and control flow of programs executing in the JVM[40]. A JVM TI client(or agent) is a C/C++ dynamic library which can be provided as an argument to the JVM while executing a Java program. Agents are executed concurrently with the JVM and have a direct line of communication with the JVM instance. An example

of binding a JVM instance to a native agent is show in Figure 3.1. The interface also supports

```
java -agentpath:/home/foo/jvmti/<jvmti-agent-library-name> |
    -agentlib:<jvmti-agent-library-name> MyClass
```

Figure 3.1: Running a JVM TI agent with a JVM [40]

bytecode instrumentation, which allows developers to edit JVM byetcode instructions on the fly. The API provides two methods of operation:

- **Query Interface** - An agent is hooked to the JVM instance on initialization and receives an instance of the environment as a native object. Throughout the lifecycle of the instance, the agent can query the JVM for information about the state of the program and monitor its state.

- **Event Subscription** - An agent can subscribe to various control flow and monitoring events and register method callbacks. This transfers control of execution to the method inside the agent along with relevant event information (Java Native Interface objects) when that particular event occurs inside the JVM instance. Some events that are useful for instrumentation and profiling include – *Class File Load, Class Load, Method Entry, Method Exit, Native Method Bind[40].* Figure 3.2 contains the complete list of events available through JVM TI.

JMV TI agents are easy to develop, maintain and distribute, since they use built-in well documented API, and are more comprehensive in terms of data extraction than heap/thread dumps. Therefore, even though they add more overhead to the Java application execution time during profiling as compared to the other approaches, we implement the profiling for dynamic analysis in JShrink as a native JVM TI agent, *JMtrace.* The overhead is a concern in profiling, as the program is likely to be executed multiple times to extract different instances of execution, so we compare the performance of the following two approaches used to implement profiling through the JVM TI API. Appendix B discusses the experiments con-

```
1  typedef struct {
2      jvmtiEventVMInit VMInit;
3      jvmtiEventVMDeath VMDeath;
4      jvmtiEventThreadStart ThreadStart;
5      jvmtiEventThreadEnd ThreadEnd;
6      jvmtiEventClassFileLoadHook ClassFileLoadHook;
7      jvmtiEventClassLoad ClassLoad;
8      jvmtiEventClassPrepare ClassPrepare;
9      jvmtiEventVMStart VMStart;
10     jvmtiEventException Exception;
11     jvmtiEventExceptionCatch ExceptionCatch;
12     jvmtiEventSingleStep SingleStep;
13     jvmtiEventFramePop FramePop;
14     jvmtiEventBreakpoint Breakpoint;
15     jvmtiEventFieldAccess FieldAccess;
16     jvmtiEventFieldModification FieldModification;
17     jvmtiEventMethodEntry MethodEntry;
18     jvmtiEventMethodExit MethodExit;
19     jvmtiEventNativeMethodBind NativeMethodBind;
20     jvmtiEventCompiledMethodLoad CompiledMethodLoad;
21     jvmtiEventCompiledMethodUnload CompiledMethodUnload;
22     jvmtiEventDynamicCodeGenerated DynamicCodeGenerated;
23     jvmtiEventDataDumpRequest DataDumpRequest;
24     jvmtiEventReserved reserved72;
25     jvmtiEventMonitorWait MonitorWait;
26     jvmtiEventMonitorWaited MonitorWaited;
27     jvmtiEventMonitorContendedEnter MonitorContendedEnter;
28     jvmtiEventMonitorContendedEntered MonitorContendedEntered;
29     jvmtiEventReserved reserved77;
30     jvmtiEventReserved reserved78;
31     jvmtiEventReserved reserved79;
32     jvmtiEventResourceExhausted ResourceExhausted;
33     jvmtiEventGarbageCollectionStart GarbageCollectionStart;
34     jvmtiEventGarbageCollectionFinish GarbageCollectionFinish;
35     jvmtiEventObjectFree ObjectFree;
36     jvmtiEventVMObjectAlloc VMObjectAlloc;
37 } jvmtiEventCallbacks;
```

Figure 3.2: Complete list of JVMTI subscribable events [40]

ducted to compare the overhead introduced by both agents while profiling Java applications in our benchmark.

3.3.3.1 Callback Agent

– This involves the creation of a native JVM TI agent which subscribed to relevant events in the JVM. The registered callback receives an environment object which can be used to query the JVM for the required information, such as signatures for the invoked methods,

callee information, class names, etc. An example of a JVMTI agent which is subscribed to the `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` event can be seen in Figure 3.3. In this agent, the method `cbClassFileLoadHook` will be invoked with the class information when a class file is loaded into the JVM. Similarly we can subscribe to the method entry and method exit events through the JVM TI API. This information is then logged to an external file which can be processed offline to extract information for dynamic analysis.

3.3.3.2 Bytecode Instrumentation Agent

– This method involves the instrumentation of the Java program byte code at runtime to log the profiling information to an external destination, such as a log file. The instrumentation is implemented through a native JVM TI agent inside a callback. An example of a JVMTI agent which is subscribed to the `JVMTI_EVENT_CLASS_FILE_LOAD_HOOK` event can be seen in Figure 3.3. In this agent, the method `cbClassFileLoadHook` will be invoked with the class information when a class file is loaded into the JVM. An example of the callback method can be seen in Figure 3.4. This method instruments the loaded class on a class load event in the JVM. The instrumentation depicted in the figure inserts a callback to the `MTRACE_class` methods `MTRACE_entry` and `MTRACE_exit` at the entry and exit of each method in the loaded class, which receives the method signature, callee signature, and class names as parameters. This information is then logged to an external file in the `MTRACE_entry` method. Bytecode instrumentation through a JVMTI agent is favoured as compared to other approaches of bytecode instrumententation such as Java Agents[2], as instrumentation through a native agent has less latency and memory overhead.

As the results in Appendix B show, the JVM TI instrumentation agent is 1.5x to 5x faster as compared to the JVM TI callback agent. Based on these factors, we choose the bytecode instrumentation method to implement the profiling tool *JMtrace* for JShrink. As we show in Chapter 5, this approach is proficient in profiling usage of all types of dynamic features in Java.

[2] `https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/Instrumentation.html`

```
static void JNICALL
cbClassFileLoadHook(jvmtiEnv *jvmti, JNIEnv* env, jclass class_loaded, jobject loader,
               const char* name, jobject protection_domain,
               jint class_data_len, const unsigned char* class_data,
               jint* new_class_data_len, unsigned char** new_class_data);

/* Agent_OnLoad: This is called immediately after the agent is
 *   loaded. This is the first code executed.
 */
JNIEXPORT jint JNICALL
Agent_OnLoad(JavaVM *vm, char *options, void *reserved)
{
  jvmtiEnv            *jvmti;
  jvmtiError           error;
  jvmtiCapabilities    capabilities;
  jvmtiEventCallbacks  callbacks;

  /*Get JVM TI env*/
  res = (*vm)->GetEnv(vm, (void **)&jvmti, JVMTI_VERSION_1);

  (void)memset(&capabilities,0, sizeof(capabilities));
  capabilities.can_generate_all_class_hook_events = 1;
  error = (*jvmti)->AddCapabilities(jvmti, &capabilities);

  /* Providing the pointers to the callback functions to this jvmtiEnv*/
  (void)memset(&callbacks,0, sizeof(callbacks));

  /* JVMTI_EVENT_VM_START */
  callbacks.VMStart        = &cbVMStart;
  /* JVMTI_EVENT_VM_INIT */
  callbacks.VMInit         = &cbVMInit;
  /* JVMTI_EVENT_VM_DEATH */
  callbacks.VMDeath        = &cbVMDeath;
  /* JVMTI_EVENT_CLASS_FILE_LOAD_HOOK */
  callbacks.ClassFileLoadHook = &cbClassFileLoadHook;

  error = (*jvmti)->SetEventCallbacks(jvmti, &callbacks, (jint)sizeof(callbacks));

  error = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE, JVMTI_EVENT_VM_START, (jthread)NULL);

  error = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE, JVMTI_EVENT_VM_INIT, (jthread)NULL);

  error = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE, JVMTI_EVENT_VM_DEATH, (jthread)NULL);

  error = (*jvmti)->SetEventNotificationMode(jvmti, JVMTI_ENABLE, JVMTI_EVENT_CLASS_FILE_LOAD_HOOK, (jthread)NULL);

  return JNI_OK;

}
```

Figure 3.3: Example of JVMTI agent subscribed to the Class file load event

```
1  /* Callback for JVMTI_EVENT_CLASS_FILE_LOAD_HOOK */
2  static void JNICALL
3  cbClassFileLoadHook(jvmtiEnv *jvmti, JNIEnv* env, jclass class_being_redefined, jobject loader,
4                  const char* name, jobject protection_domain,
5                  jint class_data_len, const unsigned char* class_data,
6                  jint* new_class_data_len, unsigned char** new_class_data)
7  {
8      enter_critical_section(jvmti); {
9          if ( !gdata->vm_is_dead ) {
10             const char *classname;
11             classname = strdup(name);
12
13             /* The tracker class itself? */
14             if ((strcmp(classname, STRING(MTRACE_class)) != 0)) {
15                 int           system_class;
16                 unsigned char *new_image;
17                 long          new_length;
18                 new_image = NULL;
19                 new_length = 0;
20
21                 /* Call the class file write code to instrument class methods*/
22                 java_crw_demo(cnum,
23                     classname,
24                     class_data,
25                     class_data_len,
26                     system_class,
27                     STRING(MTRACE_class), "L" STRING(MTRACE_class) ";",
28                     STRING(MTRACE_entry), "(II)V",
29                     STRING(MTRACE_exit), "(II)V",
30                     NULL, NULL,
31                     NULL, NULL,
32                     &new_image,
33                     &new_length,
34                     NULL,
35                     &mnum_callbacks);
36
37                 if ( new_length > 0 ) {
38                     unsigned char *jvmti_space;
39                     jvmti_space = (unsigned char *)allocate(jvmti, (jint)new_length);
40                     /*send new class body to JVM*/
41                     (void)memcpy((void*)jvmti_space, (void*)new_image, (int)new_length);
42                     *new_class_data_len = (jint)new_length;
43                     *new_class_data   = jvmti_space;
44                      }
45                  else{
46                           log_message(gdata->log, "Could not edit class %s\n",classname);
47                      }
48                 (void)free((void*)new_image);
49               }
50             } (void)free((void*)classname);
51         }
52     } exit_critical_section(jvmti);
53 }
```

Figure 3.4: Example of JVMTI agent instrumenting a class in the Class file load callback

CHAPTER 4

Approach

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 to verify its correctness. In Phase I, JShrink performs profile-augmented static analysis to determine used and unused code. In Phase II, JShrink applies four bytecode transformations: remove unused classes and class members, merge a class hierarchy to delayer unnecessary abstractions, and inline methods to avoid indirection. Finally, JShrink reruns the given test cases to check the behavior consistency between the original program and its debloated version.

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 to ensure type safety of debloated bytecode, especially in class hierarchy merging.

4.1 Static Reachability Analysis

Static call graph analysis is a standard method used by previous bytecode debloating techniques [58, 25, 24] 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, indicating that a callee is reachable from the caller. 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 dynamic dispatching, e.g., class hierarchy analysis (CHA) [10], 0-CFA [49, 19], rapid type analysis (RTA) [3], points-to analysis [47, 32], 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 also experimented with alternative approaches such as RTA and points-to analysis. Though these approaches can more precisely identify call targets than CHA, these approaches induce more computation overhead and thus do not scale to large Java projects with many library dependencies. Therefore, we choose CHA as a tradeoff, to include more large, representative Java projects. We perform a whole-program analysis, including application code, imported third-party libraries, and JRE, to build call graphs. We further extend call graphs with field access instructions in each method using ASM [6] and associate the method with referenced fields.

4.2 Dynamic Reachability Analysis

To handle dynamic language features in Java, we initially considered using a lightweight dynamic analysis approach called TamiFlex [5], 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 5.1.6. To systematically account for dynamic features, a literature survey was conducted to define the scope of dynamic features in Java.

4.2.1 Definition of Scope

Landman et al. [28] analyzed real world projects to categorize the possible use cases of Java Reflection and group the 181 public methods in the Java Reflection API into 17 functional categories. We also referred to research by Sui et al. [53] and Reif et al. [45] which include benchmarks for capturing the use cases of dynamic features in Java. Based on this analysis, we created a comprehensive list of dynamic features in modern Java.

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 [28]. Figure 4.1 shows an example of Java's reflective call API. The method `foo` is not directly invoked from the `main` method, but is invoked through reflection, which affects the generation of the call graph.

```
1  import java.lang.reflect.*;
2
3  class Reflection{
4    public static void main(String[] args)
5      throws IllegalAccessException, InvocationTargetException, InstantiationException, NoSuchMethodException {
6
7      Class appClass = Reflection.class;
8      Object appObj = appClass.newInstance();
9
10     Method method1 = appClass.getMethod("foo");
11     method1.invoke(appObj);
12
13     Method method2 = appClass.getMethod(args[0]);
14     method2.invoke(appObj);
15   }
16   public List foo(){
17     System.out.println("Hello List " + fooCalled());
18     return null;
19   }
20   public Set foo(){
21     System.out.println("Hello Set " + fooCalled());
22     return null;
23   }
24   public void fooCalled(){
25     System.out.println("world!");
26   }
27 }
```

Figure 4.1: An example of Java reflection API

2. **Ambiguous Reflection** 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. Figure 4.1 shows an example with overloaded `foo` methods. The invocation is ambiguous which affects the generation of the call graph.

3. **Dynamic Classloading** involves classes loaded through custom class loaders. Figure 4.2 shows an example where a method is invoked on a class loaded through a custom classloader. The class and method name could be parameters, which affects the generation of the call graph.

```
1 class ClassLoading{
2   public static void main(String[] args) {
3     CustomClassLoader customClassLoader = new CustomClassLoader();
4     try {
5         Class<?> c = classLoader.findClass("CustomClassLoaderTarget");
6         c.getMethod("accessMethod").invoke(c.newInstance());
7     } catch (Exception e) {}
8   }
9 }
```

Figure 4.2: An example of Java Custom Classloading

4. **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*. Figure 4.3 shows an example of a Java `lambda expression`.

```
1 public class LambdaFunction {
2   public static void main(String[] args){
3     java.util.function.Function<Integer, String> c = (i) -> target();
4     c.apply(3);
5   }
6   public String target(Integer input) {
7     return null;
8   }
9   public String target() {
10    return null;
11  };
12 }
```

Figure 4.3: An example of Java Lambda Expression

5. **Dynamic Proxy** refers to the proxy feature provided after Java 7 to dynamically create invocation handlers for a class and its methods. Figure 4.4 shows an example where the `invoke` method of the `DynamicProxy` class run as proxy for the methods in the class `ProxiedClass` at runtime. The `invoke` method is not explicitly invoked and the actually invoked method `proxiedMethod` might never be invoked inside the proxy. This affects the generation of the call graph.

```
1 import java.lang.reflect.*;
2 public class DynamicProxy implements InvocationHandler {
3   @Override
4   public Object invoke(Object o, Method method, Object[] objects) throws Throwable {
5     System.out.println("Dynamic Proxy: "+method.getName());
6     return null;
7   }
8 }
9 public class DynamicProxyClient {
10   private ProxiedClass proxy;
11   public static void main(String[] args){
12     proxy = (ProxiedClass) Proxy.newProxyInstance(ProxiedClass.class.getClassLoader(), new Class[] { ProxiedClass.class }, new DynamicProxy());
13     proxy.proxiedMethod();
14   }
15 }
```

Figure 4.4: An example of Java Dynamic Proxy

6. **Serialization** refers to dynamically loaded classes via class deserialization. Figure 4.5 shows an example of a class deserialized and loaded at runtime.

```
1 import java.io.*;
2 public class DeserializationExample {
3   public static void main(String[] args) throws Exception{
4     ObjectInputStream ois = new ObjectInputStream(new FileInputStream(args[0]));
5     ((TargetInterface) ois.readObject()).target();
6     ois.close();
7   }
8 }
9 //Example of a Serializable class implementing TargetInterface
10 class Target implements TargetInterface, Serializable{
11   public void target(){
12      System.out.println("Hello world");
13   }
14 }
```

Figure 4.5: An example of invocation for a deserialized class

7. **Java Native Interface (JNI)** is a framework that enables Java to call and be called by native code. Figure 4.6 shows an example of a class with a JNI method.

```
1  //JNIExample.java
2  public class JNIExample {
3    static {
4      System.loadLibrary("native");
5    }
6    public static void main(String[] args) {
7     new JNIExample().target();
8    }
9   private native void target();
10 }
11 //JNIExample.h
12 JNIEXPORT void JNICALL JNIExample_target(JNIEnv *, jobject);
```

Figure 4.6: An example of invocation for a JNI method

8. **sun.misc.Unsafe** is a low level API which can be used to directly manipulate memory in the JVM at runtime, including dynamically loading classes, throwing exceptions, swapping instances, allocating new instances, etc. Figure 4.6 shows an example of the use of `sun.misc.Unsafe` API where the reference of a variable is changed from one class to another at runtime.

```
1  public class UnsafeExample {
2    public Target target;
3    public static void main(String[] args) throws Exception {
4      target = new Target();
5      //dynamically changing reference from Target to class Target2
6      sun.misc.Unsafe unsafe = null;
7      try {
8      Field f = sun.misc.Unsafe.class.getDeclaredField("theUnsafe");
9      f.setAccessible(true);
10           unsafe = (sun.misc.Unsafe) f.get(null);
11           unsafe.putObject(this,
12          Utility.getUnsafe().objectFieldOffset(UnsafeExample.class.getDeclaredField("target")),
13          new Target2());
14   } catch (Exception e) {}
15     target.target();
16   }
17 }
```

Figure 4.7: An example of the use of `sun.misc.Unsafe` to change reference at runtime

A test project[1] created to include all possible use cases of dynamic features is used to systematically build the profiling tool and measure the soundness of the generated call graph analysis at each step. The test project is included in Appendix C.

4.2.2 JMtrace

To handle various dynamic language features in Java, we develop our own native profiling agent called JMtrace, which instruments method invocations using JVM TI APIs[2] to inject logging statements at the entry and exit of each method in a class during class loading. JMtrace is based on the mtrace utility included as an example JVM TI agent with `OpenJDK 8`[3]. It is implemented as a native profiling agent which leverages the bytecode instrumentation approach. In this section, we discuss the different components of JMtrace.

4.2.2.1 Native Agent

A native JVM TI agent(`mtrace.c`) is responsible for subscription to JVM events, as well as management and logging of information to an external log file. It subscribes to the following events of the JVM TI API –

1. JVMTI_EVENT_VM_INIT - Event at the initialization of a Java Virual Machine
2. JVMTI_EVENT_VM_DEATH - Event at the death of a Java Virtual Machine
3. JVMTI_EVENT_CLASS_FILE_LOAD_HOOK - Event at the load of a class file
4. JVMTI_EVENT_THREAD_START - Event at the start of a new thread
5. JVMTI_EVENT_THREAD_END - Event at the end/death of a thread

---

[1] `https://github.com/jay-ucla/jvmmethodinvocations`

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

[3] `http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/demo/jvmti/mtrace`

The JVM initialization event is used to set up the necessary data structures inside the native agent. The instrumentation for profiling is performed in the handler for the class file load event. Every time a required class is loaded into JVM during execution, it triggers a class file load event and invokes the handler method in the native agent. The agent stores information about the loaded class in a global data structure (Figure 4.9) and iterates through all methods in the class file. It injects each method with two callbacks – an invocation of `method_entry` in the JMtrace Callback Class at the beginning of every method, an invocation of `method_exit` in the JMtrace Callback Class before the return from the method. This is illustrated through an example in Figure 4.8 which shows an `Example` class before and after instrumentation. The JMtrace Callback Class is shown in Figure 4.13. For native Java methods(`JNI`), similar callback injections are instrumented directly to the entry and exit methods in the native agent.

```
1 class Example{
2   private int field1;
3   private void target1(){
4     System.out.println("target1");
5   }
6   public void target2(){
7     System.out.println("target2");
8   }
9 }
```

(a) Before instrumentation

```
1  class Example{
2    private int field1;
3    private void target1(){
4      Mtrace.method_entry(1,1);
5      System.out.println("target1");
6      Mtrace.method_exit(1,1);
7    }
8    public void target2(){
9      Mtrace.method_entry(1,2);
10     System.out.println("target2");
11     Mtrace.method_exit(1,2);
12   }
13 }
```

(b) After instrumentation

Figure 4.8: An Example Java class

The entry and exit methods in the native agent extract the invoked method signature, callee information and class information from the thread instance. The information is stored into a global object in the native agent as shown in Figure 4.9. In case this method had been invoked by another method, the callee is appended in the list of callers for this method. Otherwise, this method is added to the list of invoked methods with the callee as the only member of the caller list as shown in the sample log in Figure 4.11 at line 3. The agent also

```c
/* Data structure to hold method and class information in agent */
typedef struct MethodInfo {
  const char *name; /* Method name */
  const char *signature; /* Method signature */
  int calls; /* Method call count */
  int returns; /* Method return count */
  char *callers;
} MethodInfo;
typedef struct ClassInfo {
  const char *name; /* Class name */
  int mcount; /* Method count */
  MethodInfo *methods; /* Method information */
  int calls; /* Method call count for this class */
} ClassInfo;
/* Global agent data structure */
typedef struct {
  /* JVMTI Environment */
  jvmtiEnv *jvmti;
  jboolean vm_is_dead;
  jboolean vm_is_started;
  /* Data access Lock */
  jrawMonitorID lock;
  /* Options */
  char *include;
  char *exclude;
  int max_count;
  /* ClassInfo Table */
  ClassInfo *classes;
  jint  ccount;

  /*Logfile*/
 FILE *log;
} GlobalAgentData;
```

Figure 4.9: The global data structures in the native agent

maintains a count for the number of times any particular method has been invoked.

At the event of the death of the JVM, the captured information is logged to an external file. Figure 4.10 shows a code snippet from the handler function for the JVM death event that logs the profiling information into the log file. This file is later parsed to extract the caller-callee relationships and referenced classes to obtain information for dynamic analysis in JShrink. Figure 4.11 shows a snippet from the profiling log generated by JMtrace for a sample Java application. The format for the log file is shown in Figure 4.12.

```
1  log_message(gdata->log, "Class,%s,%d,calls\n", cp->name, cp->calls);
2  /* Sort method table (in place) by number of method calls. */
3  qsort(cp->methods, cp->mcount, sizeof(MethodInfo), &method_compar);
4  for ( mnum=cp->mcount-1 ; mnum >= 0 ; mnum-- ) {
5    MethodInfo *mp;
6    mp = cp->methods + mnum;
7    log_message(gdata->log, "Method,%s,%s,%d,calls,%d,returns,%s\n",
8                    mp->name, mp->signature, mp->calls, mp->returns, mp->callers);
```

Figure 4.10: Native agent code snippet that logs the profiling information from global data structures

```
1  Class,com/test/Main,22,calls
2  Method,staticHello,(Ljava/lang/String;)V,13,calls,13,returns,com.test.Main: main
3  Method,instanceHello,()V,2,calls,2,returns,com.test.Main: main;com.test.Main: innerHello
4  Method,main,([Ljava/lang/String;)V,1,calls,1,returns,
5  Method,reflectiveHello,(Ljava/lang/String;)V,1,calls,1,returns,sun.reflect.NativeMethodAccessorImpl: invoke0
6  Method,innerHello,()V,1,calls,1,returns,com.test.Main: main
7  Method,instanceHello,(Ljava/lang/String;)V,1,calls,1,returns,com.test.Main: main
8  Method,lambdaPrint,(Ljava/lang/String;)Z,1,calls,1,returns,com.test.Main: main
9  Method,staticHello,()V,1,calls,1,returns,com.test.Main: main
10 Method,<init>,()V,1,calls,1,returns,com.test.Main: main
11 Method,sayHello,(Ljava/lang/String;)V,0,calls,0,returns,(null)
```

Figure 4.11: Sample log file generated by JMtrace

```
1  # For Classes in Java application
2  Class,<ClassName>,<No. of times class is referenced in application>,calls
3  # For Methods in application class <ClassName>
4  Method,<MethodName>,<Method return type>,<No. of times invoked>,calls,<No. of times finished execution>,returns,<List of callee methods>
```

Figure 4.12: Profiling log file format

4.2.2.2 JMtrace Callback Class

A Java class(`Mtrace.java`) includes the Java class and static methods which are invoked from each instrumented method in the Java application at entry and exit. As shown in Figure 4.13, these methods receive a thread instance and the relevant parameters to identify the class in the native agent. The thread instance is used to obtain the caller information which is passed on to the native `method_entry` and `method_exit` methods in the JMtrace Native Agent along with the method information.

```
1  public class Mtrace {
2    /* At the very beginning of every method, a call to method_entry() is injected */
3    private static int engaged = 1;
4    private static native void _method_entry(Object thr, int cnum, int mnum, String caller);
5    public static void method_entry(int cnum, int mnum)
6    {
7      if ( engaged != 0 ) {
8        String caller = "";
9        StackTraceElement ste[] = Thread.currentThread().getStackTrace();
10       if(ste.length>3){
11         caller += ste[3].getClassName()+": "+ste[3].getMethodName();
12       }
13       _method_entry(Thread.currentThread(), cnum, mnum, caller);
14     }
15   }
16   /* Before any of the return bytecodes, a call to method_exit() is injected */
17   private static native void _method_exit(Object thr, int cnum, int mnum);
18   public static void method_exit(int cnum, int mnum)
19   {
20     if ( engaged != 0 ) {
21       _method_exit(Thread.currentThread(), cnum, mnum);
22     }
23   }
24 }
```

Figure 4.13: The entry and exit methods in the Mtrace class

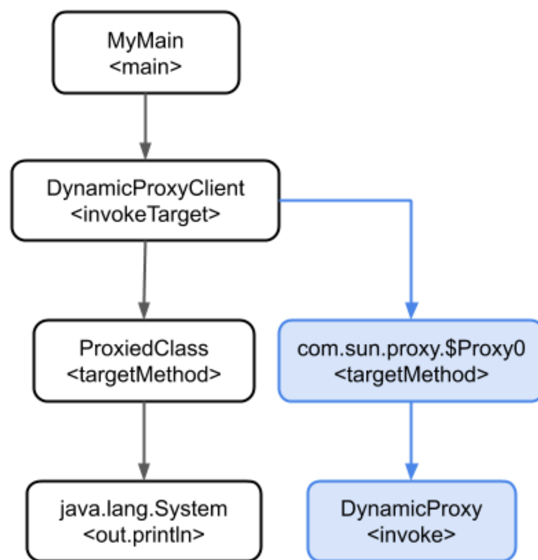
4.2.3 Integration of JMtrace with JShrink

JMtrace is integrated into JShrink as an optional tool. JMtrace is integrated as an alternative to Tamiflex [5] or as an additive component for dynamic analysis. The integration includes a Java class which is responsible for the interaction of JShrink and JMtrace. The Java class executes JMtrace with a Java project test suite provided through JShrink to

gather the run-time information from the project into a log file. The class also parses the log file to extract callee-caller relationships and class references. JSHRINK uses this dynamic information to augment the existing static call graph of the project with new entry points. Figure 4.14 shows an example of such call graph augmentation through dynamic analysis. Figure 4.14(a) contains a Java program which uses the dynamic proxy feature. Figure 4.14(b) shows the call graph for the invocation of the `target` method in the `ConcreteProxiedClass`. The white nodes show the part of the callgraph constructed through static analysis of the code in (a). The `DynamicProxy` class implements the proxy method `invoke`, which is not captured in the original call graph constructed through static analysis. JMTrace captures

```
1  import java.lang.reflect.*;
2  public interface ProxiedClass {
3    public void targetMethod();
4  }
5  public class ConcreteProxiedClass{
6    public void targetMethod(){ System.out.println("Target
         Invoked"); }
7  }
8  public class DynamicProxy implements InvocationHandler {
9    @Override
10   public Object invoke(Object o, Method method, Object[]
         objects) throws Throwable {
11     //Dynamic Proxy interrupts targetMethod
12     return null;
13   }
14 }
15 public class DynamicProxyClient {
16   private ProxiedClass proxy = (ProxiedClass) Proxy.
         newProxyInstance(ConcreteProxiedClass.class.
         getClassLoader(), new Class[] { ProxiedClass.class },
         new DynamicProxy());
17   public void invokeTarget(){
18     proxy.targetMethod();
19   }
20 }
21
22 public class MyMain {
23   public static void main(String[] args){
24      new DynamicProxyClient().invokeTarget();
25   }
26 }
```

(a) Dynamic proxy example



(b) Call graph for `invokeTarget` on line 24 in (a).

Figure 4.14: Example call graph augmentation through dynamic analysis.

the execution of the internal methods of the intermediate `Proxy` class created in the JVM and

Table 4.1: Capability of Handling Different Dynamic Features

Java Feature	Static	Tamiflex	JMtrace
Reflection	○	●	●
Reflection-ambiguous	○	◐	●
Dynamic class loading	○	●	●
Dynamic proxy	○	○	●
Invokedynamic	◐	◐	●
JNI	○	○	●
Serialization	○	●	●
Unsafe	○	◐	●

the `DynamicProxy` class method `invoke` at runtime, and augments the call graph with these additional nodes, shown in blue color in Figure 4.14(b). Thus, JMTrace helps capture the runtime information not accessible through static analysis through profiling actual execution. 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. Since JMtrace only instruments method bodies, it is not capable of identifying dynamically accessed fields via reflection. Therefore, we customize TamiFlex [5] to only instrument reflection calls related to field accesses and use it together with JMtrace. Instrumentation to other reflection calls is disabled to avoid redundant profiling.

Table 4.1 compares the capability of handling different kinds of dynamic features between static call graph analysis, TamiFlex, and JMtrace performed using the test project. The test project is included in its entirety in Appendix C.

4.3 Type Dependency Analysis

Traditional reachability analysis only keeps track of invoked methods and accessed fields, which is sufficient for method and field removal. Previous bytecode debloating techniques consider a class unused if none of its methods or fields is reachable from entry points [57, 25, 24]. However, we find this definition of unused classes is *problematic* in practice. Because modern Java allows developers to reference classes in various ways not just limited to variable and method declaration or class inheritance, but through pluggable annotations, class literals, `throws` clauses, etc. Therefore, a program could only reference a class without instantiating it, invoking a method on it, or accessing its field member. 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 JVM or lead to `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 type dependency 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 bytecode debloating techniques [57, 25, 24]. 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 `ClassNotFoundError`s at runtime. Therefore, JShrink *partially* debloats reference-only classes to ensure type safety, as explained in class hierarchy collapsing.

4.4 Bytecode Debloating Transformations

Inspired by Tip et al. [57], JShrink provides the following bytecode debloating transformations. We do not replicate class attribute removal and constant pool compression as they are already implemented in Soot [60].

4.4.1 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 usage. 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 when using debloated software in the future.

4.4.2 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.

4.4.3 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 [15, 16]. In modern Java context, JShrink applies three constraints to ensure type safety. 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 4.6). Furthermore, JShrink does not inline `synchronized` methods.

4.4.4 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 be

able to 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. Specifically, 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 further injects a default constructor as enforced by JVM. If a reference-only class is an interface, JShrink keeps those method declarations whose method 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, JShrink moves all used method and field members of X into Y while removing unused class members in Y. Secondly, JShrink 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, we add 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. Though merging classes may not lead to significant size reduction, it reduces constant pool duplication across classes and reduces unnecessary use of indirection and abstraction.

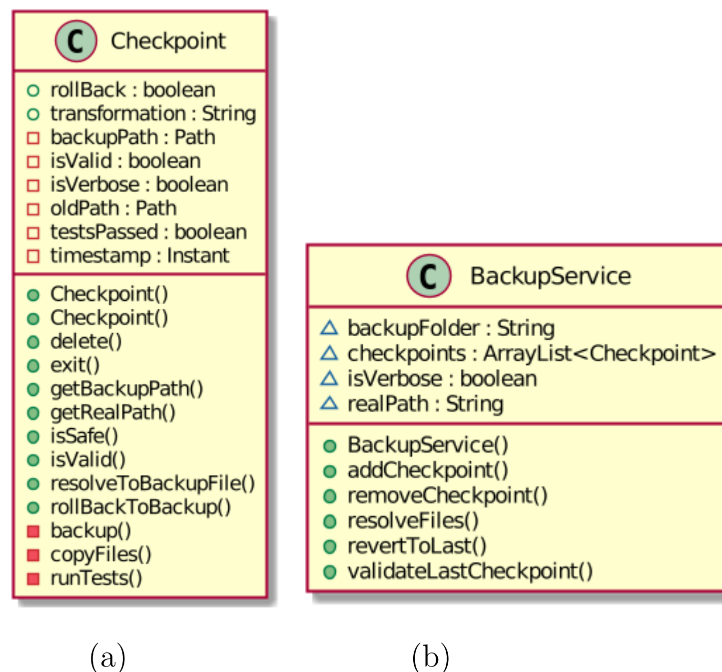
4.5 Checkpointing

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 5.1.5). Therefore, JShrink implements an additional strategy of checkpointing to ensure safety as an optional feature, which enables JShrink to ensure 100% semantic preservation at the cost of code size reduction. Another motivation for the implementation

of this feature was to let the users of JShrink extract the most benefit from the tool with the assurance of preservation of the functionality of their code. The preservation is maintained by leveraging the test suite of the program. It checkpoints each type of debloating transformation, runs tests, and reverses failure-inducing transformations. In this section, we discuss the details of this feature.

4.5.1 Checkpoint

A checkpoint refers to the state of the program at a fixed time during the processing of the program by JShrink. The state of the program is represented by the actual class files of the application code as well as the library dependency class files and JARs. The checkpoint service in JShrink provides an interface in the code for copying the state of the program to a specified location, running the test suite for the program, resolving original file paths for program artifacts to the checkpoint location of those artifacts, and rolling back. The class diagram for the Checkpoint class is shown in Figure 4.15 (a).



(a) (b)

Figure 4.15: Class Diagram for (a)`Checkpoint` (b)`BackupService`

4.5.2 Backup Service

The backup service is a high level abstraction above checkpoints which manages various checkpoints during the execution of JShrink for a particular program. It manages checkpoints in a first-in, last-out fashion, with the latest checkpoint being available for verification and manipulation in JShrink. It provides an interface to create a new checkpoint, resolve class paths to a checkpoint, verify the current state at a particular checkpoint, and rolling forward/backward the project code to any particular checkpoint. The class diagram for the Backup Service class is shown in Figure 4.15 (b).

4.5.3 Workflow

The workflow of this feature in JShrink is as follows. The workflow of the Checkpointing feature is also illustrated in the code snippet from JShrink in Figure 4.16.

1. BackupService is initialized inside the Application class.
2. An initial checkpoint is created before applying any transformation.
3. At the end of each transformation on the SOOT classes (in memory representation of classes), workflow is kicked off for checkpoint verification.
4. A new checkpoint is created. This copies the original project folder to a temporary location as mentioned at a path in `backupFolder`.
5. This temporary project copy is modified through JShrink.
6. Tests are executed for this copy.
   (a) If all test cases pass, we move on.
   (b) If test cases fail:
      i. This checkpoint is deleted.
      ii. Modified files are copied from the last checkpoint folder to the project folder.

7. Modified files are copied from the last checkpoint folder to the project folder.

8. All checkpoints are deleted (All temporary copies of project are removed).

9. The Application exits after completing execution.

```
1  //add new checkpoint before applying tranformation
2  backupService.addCheckpoint(transform);
3
4  //update files in the checkpoint location
5  jShrink.updateClassFilesAtPath(backupService.resolveFiles(jShrink.getClassPaths()));
6
7  //execute test set
8  if(!backupService.validateLastCheckpoint()){
9    //if not safe
10   //remove the current checkpoint
11   backupService.removeCheckpoint();
12   //revert application code to previous checkpoint
13   backupService.revertToLast();
14   //clean up all checkpoints
15   while(backupService.removeCheckpoint()){}
16   System.err.println("Exiting after checkpoint failure - "+transform);
17   toLog.append(jShrink.getLog());
18   return false;
19 }
20 else{
21   //all tests successfully passed
22   return true;
23 }
```

Figure 4.16: Code snippet illutsrating checkpointing workflow

4.6 Implementation and Nuanced Extensions

We implement those bytecode debloating transformations using a Java bytecode analysis and manipulation framework called Soot [60]. We use the CHA implementation in Soot for static analysis, use ASM [6] to gather field accesses, and implement JMtrace using JVM TI API [40]. In addition to dynamic profiling and type dependency analysis, we highlight several nuanced extensions we implemented to ensure type safety and behavior preservation of debloated software.

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, we account for co-variant return types to determine overridden methods when merging two classes. Otherwise, JVM will throw a verification error. An example is shown in Figure 4.17.

```
1 class A {}
2 class B extends A {}
3 class C {
4  A target() { return new A(); }
5 }
6 class D extends C {
7 //Overrides target() in parent class C
8  B target() { return new B(); }
9 }
```

Figure 4.17: An example of co-variant return types in a class hierarchy

2. *Class member visibility.* When inlining a method or merging a class, it is important not to break access controls. For example, 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. Furthermore, 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. Consider the following scenario. Suppose a base class `B` inherits a method `m` from its super class `A` 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, we 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. An example is shown in Figure 4.18 along with a code snippet in Figure 4.19.

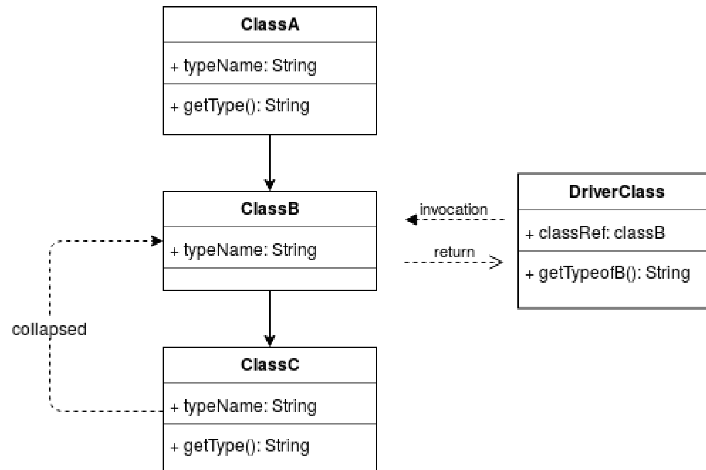


Figure 4.18: An example of an unsafe inheritance hierarchy for class collapsing

```java
class A{
  public void getType(){
    System.out.println("Target A");
  }
}
//Class B inherits target from A
class B extends A{
}
class C extends B{
  @Override
  public void getType(){
    System.out.println("Target C");
  }
}
class UnsafeExample{
  //Unsafe to merge C into B
  public static void main(String args[]){
    A a = new B();
    //if C is merged into B, the behaviour of a.target changes
    a.getType();
    C c = new C();
    c.getType();
  }
}
class SafeExample{
  //Safe to merge C into B
  public static void main(String args[]){
    A a = new A();
    //if C is merged into B, the behaviour of a.target in unchanged
    a.getType();
    C c = new C();
    c.getType();
  }
}
```

Figure 4.19: A code snippet demonstrating unsafe inheritance hierarchy for class collapsing

In summary, in comparison to prior debloating work [57, 25], we make 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 4.6 to handle new language features properly.

CHAPTER 5

Evaluation & Results

We conduct a systematic evaluation of bytecode debloating transformations using modern Java applications in Section 5.1 to tackle the research questions outlined in Chapter 1. In Section 5.2, we conduct a systematic evaluation of our native dynamic profiler, to evaluate its capability of handling different dynamic features in Java. This helps gain further insight into the usefulness of dynamic profiling for bytecode debloating.

5.1 JShrink Evaluation

5.1.1 Benchmark

To conduct a systematic study on modern Java applications, we build an automated infrastructure to construct a benchmark. The infrastructure uses the Google BigQuery API[1] to query GitHub projects and automatically applies a rigorous set of filtering criteria listed below. A complete list of the projects in the current benchmark is included in Appendix A.

- *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

[1] `https://cloud.google.com/bigquery/public-data/`

current implementation supports Maven [37], a popular build system used in Java.

- *Compilable.* After downloading those projects, we exclude those that induce build failures on 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 to keep our research experiment under a reasonable time.

Table 5.1: Project statistics

	Stars	Tests	Libs	SLOC (App Only)	Size (KB: App+Libs)[2]
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	69,189	5,213	332	324,035	346,160
SD	3,595	370	17	22,288	30,766

The final benchmark shown in Table 5.2 covers a wide spectrum of Java programs, including popular libraries, web applications, development and testing frameworks, and desktop applications. Table 5.1 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 size ranges from 30KB to 112MB. The median is 3MB. Cobertura [1] reports 34.1% statement coverage by their existing tests, which we use for assessing behavior preservation after debloating.

5.1.2 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 [57], JRed [25], and ProGuard [20]. 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 4.6's extensions. Jax imposes an additional constraint that requires unused, to-be-removed classes must not have any derived classes. So we modify the class collapsing transformation accordingly.

JRed is recent but only supports method removal and class removal. Both tools rely 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 4.6.

ProGuard shrinks and obfuscates Java bytecode and is publicly available. It has been integrated into Android SDK and is widely used to optimize Android applications. Similar to Jax and JRed, ProGuard also only performs static analysis. It does not construct call graphs but instead only traverses bytecode instructions in a given method to calculate a transitive

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

closure of all referenced classes, methods, and fields. Unlike Jax and JRed, 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 calls. As ProGuard is publically available, we evaluate it directly. We use version 6.3 in this experiment.

5.1.3 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 Chapter 1:

- RQ1 What Java bytecode size reductions are 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?

5.1.4 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 [57] and JRed [25] in their original papers only use `main` methods as the entry points of their static analysis. 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 5.2 shows the size reduction ratio achieved by each transformation.

Table 5.2: Results of debloating the benchmark projects.

S.No.	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
1	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)
2	bukkit	906	15.4%	1.2%	0.2%	1.9%	19.8%	24.0%	72.7%	18.5%	18.5%	18.5%	× (906)	× (906)	× (39)	× (3)	✓ (0)	✓ (0)
3	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)
4	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)
5	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)
6	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)
7	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)
8	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)
9	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)
10	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)
11	junit4	1081	1.71%	0.5%	0.1%	4.8%	2.3%	8.0%	9.0%	6.5%	6.8%	1.37%	× (1081)	× (1081)	× (43)	× (17)	× (13)	✓ (0)
12	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)
13	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)
14	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)
15	mybatis-pagehelper	106	20.1%	1.4%	0.1%	2.3%	25.5%	28.6%	65.0%	24.7%	23.9%	21.55%	× (106)	× (106)	× (85)	× (100)	× (55)	✓ (0)
16	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)
17	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)
18	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)
19	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)
20	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)
21	gwt-cal	92	16.48%	0.69%	0.05%	0.28%	19.37%	20.77%	31.6%	17.51%	17.50%	17.50%	× (3)	× (3)	✓ (0)	✓ (0)	✓ (0)	✓ (0)
22	Java-Chronicle	8	0.00%	1.09%	0.95%	1.44%	0.00%	3.48%	0.0%	3.48%	3.48%	3.48%	✓ (0)	✓ (0)	✓ (0)	× (8)	✓ (0)	✓ (0)
23	maven-config-processor-plugin	77	25.37%	3.20%	0.29%	0.98%	31.5%	35.29%	82.0%	29.82%	29.81%	29.81%	× (21)	× (21)	× (20)	✓ (0)	✓ (0)	✓ (0)
24	jboss-logmanager	42	11.09%	0.48%	0.04%	1.93%	11.69%	14.33%	17.0%	26.18%	13.55%	13.55%	✓ (0)	✓ (0)	× (24)	✓ (0)	✓ (0)	✓ (0)
25	autoLoadCache	11	16.5%	1.5%	0.3%	1.9%	18.2%	21.9%	Crash	20.2%	20.2%	16.54%	× (10)	× (10)	Crash	× (7)	× (9)	✓ (0)
26	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%	12.57%	12.52%	—	—	—	—	—	—

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. Surprisingly, despite the significant amount of extension and engineering effort, *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 5.2 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 achieves the

smallest size reduction (13.0% 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 in ProGuard while performing partial evaluation on strings. Compared with JShrink, ProGuard reduces code more aggressively (33.8% on average) because it performs static reference-based analysis, producing a relatively smaller set of reachable methods. However, ProGuard causes 6X more test failures than JShrink, as elaborated in the next section.

5.1.5 RQ2: Semantic Preservation

Reduction in bytecode size, however, is only meaningful if the semantics of the target project is preserved. To assess how closely JShrink preserves program semantics, we run existing test cases before and after debloating. We consider a program to have broken semantics if there exist any test failures after debloating. Column Test Failures shows the semantics preserving quality for JShrink, Jax, JRed, and ProGuard. "✓" denotes a project has no test failure after debloating, while "×" denotes test failures exist after debloating. The numbers in brackets show the number 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 to 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

```
1 if(object instanceof SuperClass){
2   ①
3 } else if (object instanceof SubClass){
4   ②
5 }
```

Figure 5.1: A failure induced by class reference updating.

respectively. Without checkpointing, only 81 of 5432 test cases fail after debloating using JShrink. This gives 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.

5.1.5.1 An in-depth inspection of test case failures

We rigorously analyzed the cause of other test failures caused by JShrink and classified them into *rare but challenging* corner cases. We find that JShrink causes failures in 4 projects. In total, 81 test cases fail, of 5,432 within the benchmark set. This gives JShrink a pass rate of 98.5%. Of those 81 failing test cases, we outline below in what manner JShrink breaks software semantics.

Conditional class reference update failure When a class is collapsed, the subclass is merged into the superclass, and the subclass is renamed to that of the superclass. E.g.,

subclass B merged into superclass A would result in B being deleted and all instances of B in the target application being renamed to A. In most cases this is not a problem. However, in cases where branching is dependent on an object's type, errors can arise.

Figure 5.1 shows an example of this behaviour. If `object` is an instance of `SubClass` the branch at ② would be executed. However, if `SubClass` is merged into `SuperClass` via the *class heirarchy collapsing* transformation, all instances of `SubClass` are refactored to `SuperClass`. Therefore, the `object` instance, now an instance of `SuperClass`, would result in the branch at ① being executed. This thereby breaks the semantics of the program. We find this problem causes 5.7% of all test case failures. Rectifying this scenario would require the *class hierarchy collapsing* to carry out more complex transformations on such conditional statements.

Soot Issue The *field removal*, *method inliner*, and *class hierarchy collaper* transformations remove fields, classes, and methods. For these removal to be successful, all references to these components must be removed otherwise runtime exceptions can be received when classes, containing references to these non-existant components, are loaded.

However, Soot, the bytecode analysis and modification framework which JShrink builds upon, is still under active development and therefore is unable to process all Java files. Though a rare occurance, this breaks our ability to modify these files. JShrink skips these files for modification but, if these files contain references to removed components, and this unmodifiable class is loaded at runtime, exceptions can be thrown. In some cases, SOOT results in malformed Jimple body, which leads to runtime errors. We find this issue results in a majority of errors received when running JShrink on our benchmark set. The prevalence of originating from Soot will decrease as Soot continues to mature.

Inlining generic class method invoked through reflection An instance of an anonymous class which implements a `generic` abstract class has 2 versions of the overidden methods - the method with the exact type T which was used during initialization, called `method1(T)`

shown in Figure 5.2, and a method with `java.lang.Object` in place of the type `T`, which invokes method1 internally and is the actual face of this method, called `method1(Object)` shown in Figure 5.2. Since `method1(T)` is only invoked in `method1(Object)` with no other call sites, JShrink inlines `method1(T)` into `method1(Object)`. An issue occurs in `junit4` where a class reflectively searches its class hierarchy for an implementation of `method1` to assign the value of a field. After inlining, this field is assigned the value `java.lang.Object` instead of `T`. This causes unexpected behaviour leads to test case failure.

```
1 public abstract class Example<T>{
2   protected abstract boolean method1(T var1);
3 }
4 class Implementer$1 extends Example<MyClass>{
5   public boolean method1(MyClass item){
6     return item.boolField;
7   }
8   public boolean method1(Object item){
9     return this.method1((MyClass)item);
10  }
11 }
```

Figure 5.2: Structure for example anonymous class implementing a `generic` abstract class

Dynamic code generation In one project, `fragmentargs`, we found that errors occurred due to dynamic code generation. Using templates of Java files, code was created and compiled at runtime. An example is shown in Figure . JShrink analyzes all the bytecode prior to execution and does not check, while running our dynamic analysis, whether new classes were created. Therefore, in several instances, we break the semantics of the code. Dynamic code generation results in an incomplete view of the program which would require significant engineering effort to solve. Fortunately this problem is rare, accounting for 4 test case failures out of 81.

5.1.6 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 also compares with an

```java
import com.google.testing.compile.JavaFileObjects;
import static com.google.common.truth.Truth.assert_;
import static com.google.testing.compile.JavaSourceSubjectFactory.javaSource;
@Test
public void assertClassCompilesWithoutError() {
  String template = "ClassTemplate.java";
  assert_().about(javaSource())
          .that(JavaFileObjects.forResource(template))
          .compilesWithoutError()
}
```

Figure 5.3: Dynamically compiled class in test cases.

alternative profiler called TamiFlex [5].

5.1.6.1 Entry point analysis.

As discussed in Section 4.1, 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 5.3 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 5.3: 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%)

5.1.6.2 JMtrace vs. TamiFlex.

As discussed in Section 4.2, 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 [5] only instruments a predefined set of reflection APIs and thus is considered more light-weight. The two TamiF. columns in Table 5.2 show the size reduction and test failures caused by the TamiFlex variant of JShrink. We perform a rigorous comparison between JMTrace and TamiFlex in Section 5.2 and further discuss these results in Section 5.2.2.

5.1.6.3 An in-depth inspection.

To investigate which extension aided in improving behavior preservation, we chose one project, `java-apns` for a thorough investigation into each failure. This is because there are total 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

examined and determined 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, and each of the remaining enhancements contributes to behavior preservation.

5.1.7 RQ4: Software Debloating 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. As shown in Table 5.4, 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.

5.2 JMtrace Evaluation

In this section, we compare the performance of JMTrace to an alternative profiling technique called TamiFlex [5], using a rigorous benchmark of Java dynamic features. We further

Table 5.4: Results of held-out tests for the benchmark projects.

Application	Tests	Tests left out	Test Failures	Test Failures (Table 5.2)
bukkit	906	180	18	18
disklrucache	61	1	0	0
rxrelay	58	10	0	0
rxreplayingshare	20	2	0	0
retrofit1-okhttp3-client	9	3	**3**	0
Java-chronicle	8	1	0	0
tprofiler	3	1	0	0
jvm-tools	102	40	0	0
qart4j	1	1	0	0
dubbokeeper	1	1	0	0
frontend-maven-plugin	6	6	0	0
gson	1050	209	0	0
gwt-cal	92	15	0	0
jboss-logmanager	42	42	0	0
junit4	1081	215	**18**	13
http-request	163	2	0	0
lanterna	34	3	0	0
maven-config-processor-plugin	77	14	0	0
java-apns	81	17	**45**	0
mybatis-pagehelper	106	22	55	55
algorithms	493	99	0	0
autoLoadCache	11	3	**10**	9
fragmentargs	15	3	4	4
moshi	835	181	0	0
tomighty	26	13	0	0
zt-zip	121	21	0	0

compare JShrink to a variant of JShrink that uses TamiFlex instead of JMtrace through the context of JShrink experiment results presented in Table 5.2.

5.2.1 Benchmark

Sui et al. [53] presents a micro-benchmark[3] to measure the coverage of dynamic Java features by static analysis tools. Their main aim is to capture all features that allow the user to customise some aspects of the execution semantics of a program. It provides a simple structure as deterministic source and target methods for each dynamic feature in Java. In addition, it provides a perfect driver in the form of unit test cases which invoke each source method. This micro-benchmark, although developed to measure static analysis performance, works just as well for measuring coverage through a hybrid approach and for our purpose of gauging the precision of our approach. The benchmark catalogues of a comprehensive list of dynamic features in modern Java and includes implementations of these features in the form of Java classes as discussed below:

5.2.1.1 Reflection

The benchmark includes twelve use cases each for different use cases of reflection. The use cases include different reflection APIs available in Java. It includes simpler reflection cases such as cases where the parameters to `Method.invoke, Class.getMethod` are accessible at the call site and more complex use cases of reflection invocations which require inter method and intra method analysis of data and control flow.

1. `dpbbench.reflection.instantiation.Basic1` - uses the `getConstructor` and the `newInstance` API to instantiate a new object of a Target inner class. The profiling tool needs to log the invocation of the class constructor as part of the call graph.

2. `dpbbench.reflection.instantiation.Basic2` - uses the `newInstance` API to instantiate a new object of a Target class that is not an inner class. The profiling tool needs

[3] `https://bitbucket.org/Li_Sui/benchmark/src/default/`

to log the invocation of the class constructor as part of the call graph.

3. `dpbbench.reflection.instantiation.ConstructorOverloading` - obtains an overloaded constructor of a target class using the `getConstructor` API. The `newInstance` API is then to instantiate a new object of a Target inner class as shown below. The profiling tool needs to log the invocation of the class constructor as part of the call graph.

   ```
   claz.getConstructor(new Class[] { ConstructorOverloading.class,
       String.class }).newInstance(this, "hello");
   ```

4. `dpbbench.reflection.instantiation.Interprocedural1` - uses the `Class.forName` API to load a class with a parameterized name. The name of the class is loaded loaded from an external file. The class is then used with the `getConstructor` and the `newInstance` API to instantiate a new object. The profiling tool needs to log the invocation of the class constructor as part of the call graph.

5. `dpbbench.reflection.instantiation.Interprocedural2` - uses the `Class.forName` API to load a class with a parameterized name. The name of the class is loaded by invoking a method of another class as shown below. The class is then used with the `getConstructor` and the `newInstance` API to instantiate a new object. The profiling tool needs to log the invocation of the class constructor as part of the call graph.

   ```
   claz = Class.forName(new ClassNameProvider().getClassName());
   ```

6. `dpbbench.reflection.instantiation.Intraprocedural1` - uses the `Class.forName` API to load an inner class with a parameterized name declared as a `String` inside the same class. It then uses the `getConstructor` and the `newInstance` API to instantiate a new object. The profiling tool needs to log the invocation of the class constructor as part of the call graph.

7. `dpbbench.reflection.invocation.Basic` - uses the `getDeclaredMethod` API to find a target method and the `Method.invoke` API to invoke it. The name of the method is

declared as a `String`. The profiling tool needs to log the invocation of method as part of the call graph.

8. `dpbbench.reflection.invocation.Interprocedural1` - uses the `getDeclaredMethod` API to find a target method and the `Method.invoke` API to invoke it. The name of the method is parameterized and loaded from an external file. The profiling tool needs to log the invocation of method as part of the call graph.

9. `dpbbench.reflection.invocation.Interprocedural2` - uses the `getDeclaredMethod` API to find a target method and the `Method.invoke` API to invoke it. The name of the method is parameterized and loaded by invoking a method of another class. The profiling tool needs to log the invocation of method as part of the call graph.

10. `dpbbench.reflection.invocation.Intraprocedural1` - uses the `getDeclaredMethod` API to find a target method in the same class and the `Method.invoke` API to invoke it. The name of the method is parameterized and declared as a `String` in the main method. The profiling tool needs to log the invocation of method as part of the call graph.

11. `dpbbench.reflection.invocation.MethodOverloading` - uses the `getDeclaredMethod` API to find an overloaded target method in the same class with a particular parameter type as shown below. It then uses the `Method.invoke` API to invoke it. The name of the method is parameterized and declared as a `String` in the main method. The profiling tool needs to log the invocation of the correct method as part of the call graph.

```
m = MethodOverloading.class.
        getDeclaredMethod("target", new Class[]{String.class});
```

12. `dpbbench.reflection.invocation.ReturnTypeOverloading` -- uses the `getDeclaredMethod` API to find an overloaded target method in another class in the project classpath. It then uses the `Method.invoke` API to invoke it. The profiling tool needs to log the invocation of the correct method as part of the call graph.

5.2.1.2 Reflection with ambiguous resolution

The benchmark includes two use cases for ambiguous reflection where the classes have varied behaviour depending on the exact JVM used for execution.

1. `dpbbench.reflectionAmbiguous.Invocation` - iterates over all the methods of the class using the `getDeclaredMethods` API to find and invoke a method with a particular annotation. Since multiple methods can have the same annotation, the resolution is ambiguous and any one of the methods can be invoked. The profiling tool needs to log the method which was actually executed at runtime as part of the call graph.

2. `dpbbench.reflectionAmbiguous.ReturnTypeOverloading` - finds a method using the `getDeclaredMethod` API to find and invoke a method that has been overloaded with different return types as shown below. Since the return type cannot be specified in the `getDeclaredMethod` API, the resolution is ambiguous and any one of the methods can be invoked. The profiling tool needs to log the method which was actually executed at runtime as part of the call graph.

```
public Set target(){ return null; }
public List target(){ return null; }
```

5.2.1.3 Dynamic classloading

The benchmark includes one use case for dynamic classloading.

1. `dpbbench.dynamicClassLoading.CustomClassLoader` - uses a custom class loader to load a class from a byte array and creates a new instance using the `newInstance` API. The profiling tool needs to log the invocation of the correct constructor and the custom class loader methods as part of the call graph.

5.2.1.4 Dynamic proxy

The benchmark includes one use case for dynamic proxy.

1. `dpbbench.dynamicProxy.DynamicProxy` - declares a dynamic proxy for the methods of a class at runtime as shown below. The profiling tool needs to log the invocation of the proxy method(`MyInvocationHandler.invoke`) in addition to the invoked method(`MyInterface.foo`) as part of the call graph.

```
public class MyInvocationHandler implements InvocationHandler {
  public Object invoke(Object obj, Method m, Object[] arg) {
        return target((String) arg[0]);
  }
}
MyInterface proxy = (MyInterface) Proxy.newProxyInstance(
        MyInterface.class.getClassLoader(),
        new Class[] { MyInterface.class },
        new MyInvocationHandler());
proxy.foo("hello");
```

5.2.1.5 Invokedynamic

The benchmark includes four examples to test the support for a typical usage pattern of invoke dynamic vs general support for the feature. It includes different types of lambda functions and hardcoded invokedynamic instructions in bytecode.

1. `dpbbench.invokedynamic.LambdaConsumer` - uses a lambda method to implement a consumer interface that accepts a single input argument and returns no result. The profiling agent needs to log the lambda method and the internally invoked method as part of the call graph.

2. `dpbbench.invokedynamic.LambdaFunction` - uses a lambda method to implement a function that accepts a single input argument and returns a result. The profiling agent needs to log the lambda method and the internally invoked method as part of the call graph.

3. `dpbbench.invokedynamic.LambdaSupplier` - uses a lambda method to implement a supplier interface that accepts no input arguments and returns a result. The profiling agent needs to log the lambda method and the internally invoked method as part of the call graph.

4. `dpbbench.invokedynamic.DynamoClient` - invokes a target method through a custom `invokedynamic` instruction in bytecode as shown below. The profiling agent needs to log the invoked method as part of the call graph.

```
6: invokedynamic #46, 0
// InvokeDynamic #1:target:(Ldpbbench/invokedynamic
/target/DynamoTarget;Ljava/lang/String;)V
```

5.2.1.6 Serialization

The benchmark includes one use case for deserialization.

1. `dpbbench.serialisation.Deserialisation` - loads a serialized object into the JVM by deserializing it and invokes a method on the object. The profiling agent needs to log the invoked method as part of the call graph.

5.2.1.7 Java Native Interface (JNI)

The benchmark includes two use cases for native code.

1. `dpbbench.jni.Callbacks` - uses a native method to register a callback to a Java method. The profiling agent needs to log the invoked method as part of the call graph.

2. `dpbbench.jni.Thread` - uses a basic thread setup to invoke a Java method. The profiling agent needs to log the invoked method in the thread as part of the call graph.

5.2.1.8 Unsafe

The benchmark includes four use cases with different methods in the `sun.misc.Unsafe` API.

1. `dpbbench.unsafe.UnsafeDynamicClass` - compiles and loads a class at runtime that is not in the classpath using the `defineClass` method of the `sun.misc.Unsafe` API. It then creates a new instance of the class. The profiling agent needs to log the constructor of the dynamically loaded class as part of the call graph.

2. `dpbbench.unsafe.UnsafeInitialization` - invokes a method on an uninitialized object using the `allocateInstance` API as shown below. The profiling agent needs to log the invoked method and not log the constructor of the class part of the call graph.

   ```
   Field f = sun.misc.Unsafe.class.getDeclaredField("theUnsafe");
   f.setAccessible(true);
   sun.misc.Unsafe unsafe = (sun.misc.Unsafe) f.get(null);
   TargetClass tc = unsafe.allocateInstance(TargetClass.class);
   tc.target();
   ```

3. `dpbbench.unsafe.UnsafeException` - throws a checked `Exception`. The profiling agent needs to log the constructor of the custom exception thrown at runtime as part of the call graph.

4. `dpbbench.unsafe.UnsafeTypeConfusion` - changes the value of a reference variable of one class to an object of another class using the `objectFieldOffset` API as shown below. It then invokes a target method on the reference. The profiling agent needs to log the invoked method of the object that replaced the earlier reference as part of the call graph.

   ```
   unsafe.putObject(this,
   Utility.getUnsafe().objectFieldOffset(
     UnsafeTypConfusion.class.getDeclaredField("target")),
     new Target2());
   target.target();
   ```

Table 5.5: Comparison of dynamic feature coverage for Li Sui et al. benchmark [53]

Category	Total	Package	Name	Tamiflex	JMTrace
Reflection	12	dpbbench.reflection.instantiation	Basic1	✓	✓
			Basic2	✓	✓
			ConstructorOverloading	✓	✓
			Interprocedural1	✓	✓
			Interprocedural2	✓	✓
			Intraprocedural1	✓	✓
		dpbbench.reflection.invocation	Basic	✓	✓
			Interprocedural1	✓	✓
			Interprocedural2	✓	✓
			Intraprocedural1	✓	✓
			MethodOverloading	✓	✓
			ReturnTypeOverloading	✓	✓
Reflection-ambiguous	2	dpbbench.reflectionAmbiguous	Invocation	×	✓
			ReturnTypeOverloading	✓	✓
Dynamic class loading	1	dpbbench.dynamicClassLoading	CustomClassLoader	✓	✓
Dynamic proxy	1	dpbbench.dynamicProxy	DynamicProxy	×	✓
Invokedynamic	4	dpbbench.invokedynamic	LambdaConsumer	×	✓
			LambdaFunction	×	✓
			LambdaSupplier	×	✓
			DynamoClient	×	✓
JNI	2	dpbbench.jni	Callbacks	×	✓
			Thread	×	✓
Serialisation	1	dpbbench.serialisation	Deserialisation	✓	✓
Unsafe (sun.misc.Unsafe)	4	dpbbench.unsafe	UnsafeDynamicClass	✓	✓
			UnsafeException	×	✓
			UnsafeInitialization	×	✓
			UnsafeTypeConfusion	×	✓

Table 5.6: JShrink execution time breakdown.

	Total (s)	Mean (s)
Static call graph analysis	44779	2132
TamiFlex	8559	407
Profile-based Analysis	14424	687
Setup + Transformations	21395	1019

5.2.2 Results

We consider Tamiflex for comparison with JMTrace, as it was well integrated into the SOOT framework used for static analysis in the previous version of JShrink. Table 5.5 shows the comparison for Tamiflex and JMtrace for detection of the relevant nodes in the call graph for the programs in the benchmark. In our experiment, having a perfect driver as the available unit test cases, we aimed to capture all possible call sites and targets to obtain a complete call graph. Tamiflex is able to capture almost all types of reflection, where it often over approximates the invoked methods and captures a few false positives. Since it is a speacialized tool built for tackling reflection[5], it falls short in support for features like JNI, serialization, dynamic proxy, and invokedynamic. JMtrace is able to obtain all relevant nodes of the callgraph as expected.

These results confirm our observations in Section 5.1. The two TamiF. columns in Table 5.2 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. Table 5.6 shows the breakdown of computation overhead induced by different JShrink components in terms of execution time. JMtrace is 6X slower than TamiFlex, though both are dwarfed by the most time-consuming component of static call graph analysis. Thus, we conclude that JMtrace incorporation to JShrink is worthwhile and necessary to improve behavior preservation.

CHAPTER 6

Conclusion

Software debloating is a long standing problem and some even consider that this problem was solved 20 years ago through static reachability-analysis based code transformation. We therefore set out to replicate, extend, and rigorously evaluate prior software debloating work in the context of modern Java. Unlike previous research, we handled dynamic language features, ensured type safety, and took measures to pass the JVM's bytecode verification check. We found that prior work significantly falls short of *behavior preservation*, meaning debloated software no longer passes the same tests, with a test faulure 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 embodied in JShrink, are worthwhile and necessary to improve the test passing rate of prior work from 35% to 98%. ProGuard, a popular software debloating tool, reduced software size by almost double but also resulted in 6X more test failures compared to JShrink. Our in-depth manual investigation of test successes and failures show that incorporating dynamic profile augmentation is crucial for improving behavior preservation, and through our rigorous analysis of the dynamic features in Java, we show that out profiling agent, JMtrace, covers 100% of these features. With checkpointing, JShrink is able to provide 100% behavior preservation guarantees with marginal decrease in size reduction(0.9%). The only limitation of our approach is the reliance on *developer-written tests*. To the best of our knowledge, we are the first to systematically quantify size reduction, behavior preservation, and the benefit of dynamic profile augmentation.

APPENDIX A

List of Benchmark Projects

Table A.1: List of Benchmark Projects

Application	GitHub URL
bukkit	https://github.com/Bukkit/Bukkit
disklrucache	https://github.com/JakeWharton/DiskLruCache
rxrelay	https://github.com/JakeWharton/RxRelay
rxreplayingshare	https://github.com/JakeWharton/RxReplayingShare
retrofit1-okhttp3-client	https://github.com/JakeWharton/retrofit1-okhttp3-client
Java-chronicle	https://github.com/peter-lawrey/Java-Chronicle
tprofiler	https://github.com/alibaba/TProfiler
jvm-tools	https://github.com/aragozin/jvm-tools
qart4j	https://github.com/dieforfree/qart4j
dubbokeeper	https://github.com/dubboclub/dubbokeeper
frontend-maven-plugin	https://github.com/eirslett/frontend-maven-plugin
gson	https://github.com/google/gson
gwt-cal	http://code.google.com/p/gwt-cal/
jboss-logmanager	http://community.jboss.org/
junit4	https://github.com/junit-team/junit4
http-request	https://github.com/kevinsawicki/http-request
lanterna	https://github.com/mabe02/lanterna
maven-config-processor-plugin	http://code.google.com/p/maven-config-processor-plugin
java-apns	https://github.com/notnoop/java-apns
mybatis-pagehelper	https://github.com/pagehelper/Mybatis-PageHelper
algorithms	https://github.com/pedrovgs/Algorithms
autoLoadCache	https://github.com/qiujiayu/AutoLoadCache
fragmentargs	https://github.com/sockeqwe/fragmentargs
moshi	https://github.com/square/moshi
tomighty	https://github.com/tomighty/tomighty
zt-zip	https://github.com/zeroturnaround/zt-zip

APPENDIX B

JVM TI Bytecode Instrumentation vs Callback Tests

Table B.1: Performance comparison of JVM TI callback agent vs instrumentation agent[1]

Project Name	gwt-cal	Java-chronicle	Jboss-logmanager	maven-config-processor-plugin
Total Tests	92	8	42	77
Baseline(secs)	0.91	1.68	1.93	2.29
Callback Agent(secs)	1.55	2.3	10.9	3.91
+method name (secs)	3.22	3.8	35.05	6.78
+class(secs)	4.21	4.73	50.79	8.29
+clasname(secs)	4.92	5.32	57.77	9.75
+logging(secs)	5.42	6.29	72.94	10.87
Callback Total(x times base)	5.98	3.75	37.78	4.75
Instrumentation (secs)	1.548	2.275	8.874	3.885
Instrumentation Total(x times)	1.7	1.36	4.59	1.69

[1] `https://github.com/jay-ucla/jvmmethodinvocations/tree/master/latency-sample-projects`

APPENDIX C

Source Code for Dynamic Features Test Project

Figure C.1: DynamicProxy.java

```
1  package com.test;
2
3  import java.lang.reflect.InvocationHandler;
4  import java.lang.reflect.Method;
5
6  public class DynamicProxy implements InvocationHandler {
7
8      @Override
9      public Object invoke(Object o, Method method, Object[] objects) throws Throwable {
10         System.out.println("Dynamic Proxy hijacked : "+method.getName());
11         return 42;
12     }
13 }
```

Figure C.2: ProxiedClass.java

```
1  package com.test;
2
3  public interface ProxiedClass {
4      public void sayHello();
5  }
```

Figure C.3: ConcreteA.java

```
1  package com.test;
2
3  public class concreteA implements interfaceClass {
4      @Override
5      public void interfacedHello(String str){
6          System.out.println("Hello from class A "+str);
7      }
8  }
```

Figure C.4: ConcreteB.java

```java
1 package com.test;
2
3 public class concreteB implements interfaceClass {
4     @Override
5     public void interfacedHello(String str) {
6         System.out.println("Hello from Class B "+str);
7     }
8 }
```

Figure C.5: OtherClass.java

```java
1 package com.test;
2
3 import java.lang.reflect.*;
4
5 public class OtherClass {
6     private ProxiedClass proxy;
7     private void privateHello(String str){
8         System.out.println("Hello from the other world "+str);
9     }
10    public OtherClass(){
11        InvocationHandler handler = new DynamicProxy();
12        proxy = (ProxiedClass) Proxy.newProxyInstance(
13                ProxiedClass.class.getClassLoader(),
14                new Class[] { ProxiedClass.class },
15                handler);
16    }
17
18    public void instanceHello(){
19        privateHello("privately");
20    }
21    public void instanceHello(String str){
22        privateHello(str +"privately");
23    }
24    public void unusedHello(){
25        privateHello(". You should not be here.");
26    }
27
28    public static void reflectOnMe(Main m){
29        try {
30            Method h = Main.class.getMethod("reflectiveHello", String.class);
31            h.invoke(m, ". This call came from the OtherClass.");
32        }
33        catch(Exception e){
34            System.out.println("Caught! You're out!"+e.getMessage());
35        }
36    }
37
38    public void dynamicProxyRun(){
39        proxy.sayHello();
40    }
41 }
```

Figure C.6: InterfaceClass.java

```
1  package com.test;
2
3  public interface interfaceClass {
4      public void interfacedHello(String str);
5
6      default void defaultHello(String str) {
7          System.out.println("default Hi from interface" + str);
8      }
9
10     static void defaultstatic(){
11         System.out.println("static hi from interface");
12     }
13 }
```

Figure C.7: InterfaceWithInnerClass.java

```
1  package com.test;
2
3  public interface interfaceWithInnerClass {
4      void interfaceInnerTarget();
5
6      class innerClassToInterface{
7          static String innerClassString = "inner string";
8          void sayHello(){
9              System.out.println("Hi this is "+innerClassToInterface.innerClassString);
10         }
11     }
12 }
```

Figure C.8: InterfaceWithInnerClass.java

```
1  package com.test;
2
3  public interface interfaceWithInnerClass {
4      void interfaceInnerTarget();
5
6      class innerClassToInterface{
7          static String innerClassString = "inner string";
8          void sayHello(){
9              System.out.println("Hi this is "+innerClassToInterface.innerClassString);
10         }
11     }
12 }
```

Figure C.9: ConcreteWithInnerClass.java

```
1  package com.test;
2
3  public class concreteWithInnerClass implements interfaceWithInnerClass {
4      public void interfaceInnerTarget(){}
5  }
```

Figure C.10: CustomClassLoaderTarget.java

```
1  package com.test;
2
3  public class CustomClassLoaderTarget {
4
5      public CustomClassLoaderTarget(){
6          System.out.println("Created instance of class");
7      }
8      private void customClassPrint(){
9          System.out.println("I'm custom loaded");
10     }
11     public void accessMethod(){
12         this.customClassPrint();
13     }
14 }
```

Figure C.11: CustomClassLoader.java

```
1  package com.test;
2
3  import java.io.ByteArrayOutputStream;
4  import java.io.File;
5  import java.io.IOException;
6  import java.io.InputStream;
7
8  public class CustomClassLoader extends ClassLoader {
9
10     @Override
11     public Class findClass(String name) throws ClassNotFoundException {
12         byte[] b = loadClassFromFile(name);
13         return defineClass(name, b, 0, b.length);
14     }
15
16     private byte[] loadClassFromFile(String fileName) {
17         InputStream inputStream = getClass().getClassLoader().getResourceAsStream(
18                 fileName.replace('.', File.separatorChar) + ".class");
19         byte[] buffer;
20         ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
21         int nextValue = 0;
22         try {
23             while ( (nextValue = inputStream.read()) != -1 ) {
24                 byteStream.write(nextValue);
25             }
26         } catch (IOException e) {
27             e.printStackTrace();
28         }
29         buffer = byteStream.toByteArray();
30         return buffer;
31     }
32 }
```

Figure C.12: Main.java

```
1  package com.test;
2
3  import java.lang.reflect.Method;
4  import java.net.URL;
5  import java.util.function.Predicate;
6
7  public class Main {
8
9      public void sayHello(String str){
10         for(StackTraceElement e: Thread.currentThread().getStackTrace()){
11             if(!e.isNativeMethod())
12                 System.out.println(e.getClassName()+":"+e.getMethodName());
13         }
14         try {
15             Thread.sleep(10000);
16         } catch (InterruptedException e) {
17             e.printStackTrace();
18         }
19         System.out.println("Hello world "+str);
20     }
21
22     public static void staticHello(){
23         System.out.println("Hello static world");
24     }
25     public static void staticHello(String str){ System.out.println(str);}
26     public static boolean lambdaPrint(String str){System.out.println(str); return true;};
27     public void instanceHello(String str){
28         System.out.println("Hello instance world "+str);
29     }
30     public void instanceHello(){
31         System.out.println("Hello instance world");
32     }
33     public void innerHello(){
34         this.instanceHello();
35     }
36     public void reflectiveHello(String str){
37         System.out.println("Hello reflective world "+str);
38     }
39
40
41     public static void main(String[] args) {
42 //        for(int i=0;;i++){
43 //            try {
44 //                Thread.sleep(10000);
45 //            } catch (InterruptedException e) {
46 //                e.printStackTrace();
47 //            }
48 //            m.sayHello("Jay"+i);
49 //        }
50         //tests for compiler messages
51         Main.staticHello("Performing static invocation");
52
53         Main.staticHello();
54         Main.staticHello("Created Main class instance");
55         Main m = new Main();
56         Predicate<String> lambdaPrint = Main::lambdaPrint;
57         lambdaPrint.test("lambdaprint");
```

```
59
60         Main.staticHello("Invoked Main instance method");
61         m.instanceHello("J");
62         Main.staticHello("Invoked other Main instance method");
63         m.instanceHello();
64         Main.staticHello("Invoking Main inner method");
65         m.innerHello();
66         Main.staticHello("Creating instance of Other class");
67         OtherClass oc = new OtherClass();
68         Main.staticHello("Invoked OC instance method - should call private hello");
69         oc.instanceHello("J");
70         Main.staticHello("Invoked other OC instance method - should call private hello");
71         oc.instanceHello();
72         Main.staticHello("Invoking main class reflective hello from OC");
73         OtherClass.reflectOnMe(m);
74         Main.staticHello("Invoking dynamic proxy");
75         oc.dynamicProxyRun();
76
77         Main.staticHello("Invoking on object of Concrete A");
78         interfaceClass ic = new concreteA();
79         ic.interfacedHello("J");
80         ic.defaultHello("JJ");
81         interfaceClass.defaultstatic();
82
83         Main.staticHello("Creating anonymous class");
84         interfaceClass ac = new interfaceClass() {
85             @Override
86             public void interfacedHello(String str) {
87                 System.out.println("Hi from the anonymous class");
88             }
89         } ;
90         ac.interfacedHello("");
91         ac.defaultHello("JJJ");
92         //m.sayHello("");
93         CustomClassLoader customClassLoader = new CustomClassLoader();
94         Class<?> c = null;
95         try {
96             //c = customClassLoader.findClass(CustomClassLoaderTarget.class.getName());
97             URL url = customClassLoader.getResource("CustomClassLoaderTarget.java");
98             //Object ob = c.newInstance();
99
100            //Method md = c.getMethod("accessMethod");
101            //md.invoke(ob);
102        } catch (Exception e) {
103            e.printStackTrace();
104        }
105
106        Main.staticHello("Interface with inner class");
107        new concreteWithInnerClass();
108        new concreteWithInnerClass.innerClassToInterface().sayHello();
109    }
110 }
```

REFERENCES

[1] Cobertura: A code coverage utility for java. https://cobertura.github.io/cobertura/. Accessed: 2019-12-13.

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

[4] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. Survey of code-size reduction methods. *ACM Computer Survey*, 35(3):223–267, 2003.

[5] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 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*, pages 241–250. ACM, 2011.

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

[7] Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. 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*, pages 27–38. ACM, 2008.

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

[9] Keith D. Cooper and Nathaniel McIntosh. Enhanced code compression for embedded risc processors. In *Proceedings of the 1999 Conference on Programming Language Design and Implementation — PLDI '99*, pages 139–149. ACM, 1999.

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

[11] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *Transactions on Programming Languages and Systems*, 22(2):378–415, 2000.

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

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

[14] George Fourtounis, George Kastrinis, and Yannis Smaragdakis. Static analysis of java dynamic proxies. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 209–220, New York, NY, USA, 2018. ACM.

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

[16] Neal Glew and Jens Palsberg. Method inlining, dynamic class loading, and type soundness. *Journal of Object Technology*, 4(8):33–53, 2005.

[17] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA):68:1–68:27, October 2017.

[18] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 198–208, New York, NY, USA, 2018. ACM.

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

[20] Guardsquare. Proguard: The open source optimizer for java bytecode. `https://www.guardsquare.com/en/products/proguard`. Accessed: 2019-05-11.

[21] Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 380–394, New York, NY, USA, 2018. ACM.

[22] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the Conference on Programming Language Design and Implementation — PLDI '88*, pages 35–46. ACM, 1988.

[23] IBM. Wala - static analysis framework for java. `http://wala.sourceforge.net/`.

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

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

[26] Christian Gram Kalhauge and Jens Palsberg. 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*, ESEC/FSE 2019, pages 556–566, New York, NY, USA, 2019. ACM.

[27] Darko Kirovski, Johnson Kin, and William H. Mangione-Smith. Procedure based program compression. In *Proceedings of the 1997 International Symposium on Microarchitecture — Micro '97*, pages 204–213. ACM, 1997.

[28] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. Challenges for static analysis of java reflection: Literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, pages 507–518, Piscataway, NJ, USA, 2017. IEEE Press.

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

[30] Charles Lefurgy, Eva Piccininni, and Trevor Mudge. Evaluation of a high performance code compression method. In *Proceedings of the 1999 International Symposium on Microarchitecture — Micro '99*, pages 93–102, 1999.

[31] Haris Lekatsas, Jörg Henkel, and Wayne Wolf. Code compression for low power embedded system design. In *Proceedings of the 2000 Annual Design Automation Conference — DAC '00*, pages 294–299, 2000.

[32] Ondrej Lhoták. Spark: A flexible points-to analysis framework for java. 2002.

[33] Yue Li, Tian Tan, Yulei Sui, and Jingling Xue. Self-inferencing reflection resolution for java. In Richard Jones, editor, *ECOOP 2014 – Object-Oriented Programming*, pages 27–53, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[34] Jie Liu, Yue Li, Tian Tan, and Jingling Xue. Reflection analysis for java: Uncovering more reflective targets precisely. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 12–23. IEEE, 2017.

[35] Benjamin Livshits, Dimitrios Vardoulakis, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, José Amaral, Bor-Yuh Evan Chang, Samuel Guyer, Uday Khedker, and Anders Møller. In defense of soundiness: A manifesto. *Communications of the ACM*, 58:44–46, 01 2015.

[36] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 139–160, Berlin, Heidelberg, 2005. Springer-Verlag.

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

[38] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63.

[39] Nick Mitchell and Gary Sevitsky. The causes of bloat, the limits of health. *Proceedings of the 2007 Conference on Object-Oriented Programming Systems, Languages, and Applications — OOPSLA '07*, pages 245–260, 2007.

[40] Oracle. Java virtual machine tool interface (JVM TI). `https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/`.

[41] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. {RAZOR}: A framework for post-deployment software debloating. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1733–1750, 2019.

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

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

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

[45] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, ISSTA '18, pages 107–112, New York, NY, USA, 2018. ACM.

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

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

[48] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 2018 International Conference on Automated Software Engineering — ASE '18*, pages 329–339. ACM, 2018.

[49] Olin Shivers. *Control-flow analysis of higher-order languages.* PhD thesis, Citeseer, 1991.

[50] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. More sound static handling of java reflection. In *APLAS*, 2015.

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

[52] Venkatesh Srinivasan and Thomas Reps. An improved algorithm for slicing machine code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications — OOPSLA '16*, pages 378–393. ACM, 2016.

[53] Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. 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*, pages 69–88. Springer, 2018.

[54] Li Sui, Jens Dietrich, and Amjed Tahir. On the use of mined stack traces to improve the soundness of statically constructed call graphs. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 672–676. IEEE, 2017.

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

[56] Frank Tip. A survey of program slicing techniques. Technical report, CWI (Centre for Mathematics and Computer Science), Amsterdam, The Netherlands, The Netherlands, 1994.

[57] Frank Tip, Chris Laffra, Peter F. Sweeney, and David Streeter. Practical experience with an application extractor for java. In *Proceedings of the 1999 Conference on Object-oriented Programming, Systems, Languages, and Applications — OOPSLA '99*, pages 292–305. ACM, 1999.

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

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

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

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

[62] R. van de Wiel and P. Hoogendijk. Belt-tightening in software. Philips Res. Passw. Mag., 2001.

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

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

[65] Guoqing Xu. Finding reusable data structures. In *Proceedings of the 2012 Conference on Object-Oriented Programming Systems, Languages, and Applications — OOPSLA '12*, pages 1017–1034. ACM, 2012.

[66] Guoqing Xu. CoCo: Sound and adaptive replacement of Java collections. In *Proceedings of the 2013 European Conference on Object-Oriented Programming — ECOOP '13*, pages 1–26. Springer, 2013.

[67] Guoqing Xu. 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*, pages 111–130. ACM, 2013.

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

[69] Guoqing Xu, Matthew Arnold, Nick Mitchell, Atanas Rountev, Edith Schonberg, and Gary S evitsky. Finding low-utility data structures. In *Proceedings of the 2010 Conference on Programming Language Design and Implementation — PLDI '10*, pages 174–186. ACM, 2010.

[70] Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, and Gary Sevitsky. 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*, pages 421–426. ACM, 2010.

[71] Dacong Yan, Guoqing Xu, and Atanas Rountev. Uncovering performance problems in Java applications with reference propagation profiling. In *Proceedings of the International Conference on Software Engineering — ICSE '12*, pages 134–144. IEEE, 2012.

[72] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002.