The Pennsylvania State University
The Graduate School
College of Information Sciences and Technology

# PROGRAM ANALYSIS BASED BLOATWARE MITIGATION AND SOFTWARE CUSTOMIZATION

A Dissertation in
Information Sciences and Technology
by
Yufei Jiang



Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2017

The dissertation of Yufei Jiang was reviewed and approved* by the following:

Dinghao Wu
Associate Professor of Information Sciences and Technology
Dissertation Advisor
Chair of Committee

Peng Liu
Professor of Information Sciences and Technology

Anna Squicciarini
Associate Professor of Information Sciences and Technology

Sencun Zhu
Associate Professor of Computer Science and Engineering

Andrea Tapia
Associate Professor of Information Sciences and Technology
Director of Graduate Programs

*Signatures are on file in the Graduate School.

# Abstract

Modern software engineering allows us to build more complex software than ever before. On the other hand, it increasingly brings different types of redundancy into software products. The community uses the term "bloatware" to describe software that contains much bloat. Bloatware causes many negative consequences including larger disk footprint, higher memory consumption, longer loading and downloading time, higher software complexity and maintenance cost, and many security problems.

In this thesis, we review, investigate, and develop techniques to mitigate the problem of bloatware from a holistic point of view from software engineering, software security, and programming language perspectives. Specifically, we customize the bloatware to remove the software bloat from the following three aspects.

First, in current software engineering practice, even if only one method of a class is possibly used in a software, the complete class, the whole library package, and the entire runtime environment are still included as a whole in the software product delivery. The unused code in applications, libraries and the running environment has become one of the important sources of software bloat. Our study targets on these three different parts and takes advantage of the iterative hybrid static and dynamic analysis to identify and remove the unused code from the them.

Second, smartphones, as a platform that has more limited resources than other computation bases, are more subjected to the bloatware problem. We investigate software bloat that would lead to the rapid size increase of Android applications. We categorize the sources of Android application software bloat into two types, compile-time redundancy and install-time redundancy. In addition, we propose a fully automated approach to trimming off both types of software bloat. Our approach is mainly based on static analysis and program transformation.

Last but not least, due to various reasons for marketing, developers are keeping adding features into their software products in an ad-hoc and unsystematic way. According to some surveys, many of these features are useless to many end users. Such a phenomenon is also known as "feature creep" which is a special type of

software bloat. To reverse this unstoppable feature creep process, we propose a novel bloatware mitigation approach called feature-based software customization. We apply static dataflow analysis and an enhanced program slicing technique to customize software features from tangled code.

Our research makes substantial contributions in all of these three aspects. Our experimental results show that Java application sizes can be reduced by 44.5% on average and the JRE code can be reduced by more than 82.5% on average. By trimming redundant code, 48.6% of the known security vulnerabilities in a specific version of JRE has been removed. In the Android domain, by removing different types of redundancy together, we can reduce the size of an Android application by 40.4% on average. As for features customization, our case studies validate the potential of our approach for practical use.

# Table of Contents

# List of Figures

# List of Tables

# Acknowledgments

First and foremost, I would like to express my sincere gratitude to my Ph.D. advisor Dr. Dinghao Wu, who has been guiding me on research, teaching, and long-term career development. He has inspired me in many ways. I would like to thank him for his wisdom, enthusiasm, and patience. I feel fortunate to be mentored by him and work in his team. His great insights and supportive comments are priceless for me. His hard-working and persistence will always encourage me continuously. I truly appreciate everything I have obtained from him during this impressive journey.

I would also like to thank all my committee members, Dr. Peng Liu, Dr. Anna Squicciarini, and Dr. Sencun Zhu. I am lucky to work on multiple projects with Dr. Liu. I really appreciate his invaluable advices. Dr. Anna Squicciarini gave me constructive suggestions to help me improve my proposal, dissertation, and the research. I also want to thank Dr. Sencun Zhu for his guidance during my teaching assistant experience.

I want to give many thanks to all my research collaborators, especially my labmates. I have learned a lot from them. We have established great relationship which will endure.

This dissertation is supported in part by the Office of Naval Research (ONR) under grants No. N00014-13-1-0175, N00014-16-1-2265, and N00014-16-1-2912.

# Dedication

I dedicate my Ph.D. dissertation to my family and many friends. A special feeling of gratitude to my parents, Wei Jiang and Min Yang who give birth to me and support me throughout my life.

# Chapter 1 Introduction

## 1.1 Bloatware

With the rapid development of modern software engineering, the scale of the software products keep expanding on all measurements, which leads to the problem of software bloat [2]. The scientific community use the term "bloatware" to describe the software that has much bloat.

Both academia and industry have paid attention on the bloatware and removing software bloat for some time. When talking about the software bloat, though all existing works roughly refer to the redundant, unnecessary, unwanted, and undesired "things" in the software, the definitions given by them are still varied and vague [2–5]. Before we go any further, we define the term "software bloat" first. In this dissertation, we adopt the definition given by McGrenere [3]. The software bloat is "the result of adding new features to a program or system to the point where the benefit of the new features is outweighed by the impact on the technical resources (e.g., RAM, disk space or performance) and complexity of use" [3].

The problem of bloatware is severe. The most intuitive negative consequence caused by software bloat is the bulk sizes of modern software. Table 1.1 shows the sizes of Unix and Linux shells over last several decades, which is collected by Holzmann [6]. From the table we can see that the size of Ubantu shell is 191 times larger than the size of Unix V5 shell. Table 1.2 shows how the size of a single shell command `true` increased over the years [6]. In the year of 1979, its size is zero. According to the definition of `true`, its function is to "exit with a status code indicating success". An empty script can exactly fulfill this function. Interestingly,

Table 1.1: The Sizes of Unix/Linux Shell

| **Version** | **Year** | **Size (Bytes)** |
|---|---|---|
| Unix V5 | 1974 | 11135 |
| Unix V6 | 1975 | 11594 |
| Unix V7 | 1979 | 67799 |
| Unix V8 | 1984 | 116514 |
| Plan9 rc | 2008 | 195850 |
| Ubantu bash | 2014 | 2131992 |

Table 1.2: The Sizes of Shell Command `true`

| **Year** | **Size (Bytes)** |
|---|---|
| 1979 | 0 |
| 1983 | 18 |
| 1984 | 276 |
| 2010 | 8377 |
| 2012 | 22896 |
| 2014 | 27168 |

in present, the size of `true` is 27168 bytes. The data gives us rough idea about how software is inflating over the years.

Besides bulky sizes, software bloat also imports unnecessary complexity into the software, which increases difficulties to both development and maintenance. The complex logic included in the software makes some comprehensive analyses, detailed inspection, and formal verification too expensive to be conducted, which may make the software less trustworthy. High complexity also makes a software harder to be used, understood, and extended. A famous example is Windows Vista. Windows Vista is designed as a minor update after Windows XP originally. However, its complexity is out of control. After several times of delay, at last, it took 6 years to release, which is the longest time among the whole Windows series. It has 50M lines of code , which is much larger than its successor Windows7. However, it is more confused to users than other Windows releases.

Also, software bloat causes security problems. An apparent reason is that larger code base implies more exploitable bugs. Beyond that, if we consider some advanced attacks like Return Oriented Programming (ROP) and Property Oriented Programming (POP), a larger code base eases attackers to find a gadget or chain properties. If we consider the Android system, an Android application that contains

software bloat caused by those unnecessary features may obtain more-than-enough permissions from the system. In the end, it may become a path where user privacy leaks.

After entering the era of big data, mobile computing, cloud computing, and Internet of things, the ways that people acquire, update, and use the software have been completely changed. These new computing paradigm are amplifying the negative consequences mentioned above. For example, the inappropriate data structure design and ineffective Garbage Collection (GC) strategies in a large-scale data-intensive application will cause enormous software bloat in memory [7]. The example in mobile computing is about iPhone. According to the data released by Apple Inc, the CPU of iPhone 6 is 50 times faster than first generation iPhone. The GPU of iPhone 6 is 84 times faster than first generation iPhone. Those powerful hardwares enable developers to develop much larger and much complex applications. There are quite a few mobile apps whose sizes are larger than 2G bytes. However, the iPhone 6 hard disk size is just 2 times larger than first generation iPhone, which cannot match the increasement of other capabilities. This mismatch is the root cause of the situation that many users cannot install apps on their mobile phones due to out of storage space. Another example is that cloud computing capability acquisition is on demand. Software bloat will directly relate to higher cost on IT spending, which will weaken the competition of a company. What is worse, wireless networks, widely deployed sensors, and wearable computing devices are forming up new hard constrains to the limited resources.

Most previous works focus on the bloat in the memory, especially the bloat caused by the inappropriate usage of the variables, arrays, and other memory allocations. The research on this direction has already yielded great results and impressive achievements. However, software bloat is not just caused by incautious programmers. There are more fundamental reasons behind that. In this dissertation, we are going to review, investigate, and solve this problem from a holistic point of view including software engineering, software security, and programming language perspectives. Specifically, we discuss following three topics in this dissertation. They are

- the unused code in software and runtime environment as the side effect of modern software engineering and high level programming language,

- the multiple types of software bloat in mobile applications, and
- the redundant software features that are caused by agile development, market pressure, and software reuse.

## 1.2 Unused Code in Java

The modern programming paradigm heavily relies on big generic libraries, reusable components, and frameworks. Even in the situation where only a few classes or methods of those libraries, components, and frameworks are invoked by the application, the every piece of code of the whole libraries, components, and frameworks are still being included in the software product. Beyond a direct consequence that these unused code is making the program size larger, code size has profound impacts on many aspects of a system, including software download and installation time, program loading time, disk and memory storage, software testing and maintenance cost, battery or in general energy consumption, code complexity, software and system reliability, and security attack surface, to name a few.

As an example, a Java 7 zero-day exploit is based on bugs in the Java Runtime JRE library classes to disable the SecurityManager of Java Applet [8]. By exploiting method findClass in Class java.lang.ClassLoader and method methodFinder in Class com.beans.Finder, attackers can reflectively call method getField, which can get any private field in class sun.awt.SunToolkit. Thus, attackers can get and modify some sensitive private fields in sun.awt.SunToolkit such as AccessControlContext. Originally, the Java security policies have already banned the use of sun.awt.SunToolkit in the Java Applet scenario to prevent privilege escalation. However, knowing that sun.awt.SunToolkit is dangerous but still leaving it there finally gives attackers chances to find a path to walk around security policies and abuse it [8]. If we cut the JRE libraries for the Applet usage scenario, many similar security problems will go away.

Previous research on bloatware code size mitigation has different scopes with our research or incurs a number of limitations. Pugh [9] and Bradley et al. [10] propose packing and encoding methods to reduce the size of jar files, but they do not reduce the actual code size. Tip et al. [11] develop Jax, a tool to extract used Java code. Jax differs from ours on the handling of reflections. Jax relies

on information annotated by users, while our approach automates this process using iterative dynamic analysis. Tip et al. mainly focus on code size reduction. Our evaluation not only measures the code size reduction, but also demonstrates the impact on code complexity, memory footprint, execution time, and security. Additionally, our approach and evaluation takes JRE into consideration.

We propose a new hybrid iterative static and dynamic approach to trimming unused code from both Java Runtime Environment (JRE) and Java applications automatically. We have built a tool called JRed on top of the Soot framework. We have conducted a fairly comprehensive evaluation of JRed based on a set of criteria: code size, code complexity, memory footprint, execution and garbage collection time, and security. Our experimental results show that, Java application size can be reduced by 44.5% on average and the JRE code can be reduced by more than 82.5% on average. The code complexity is significantly reduced according to a set of well-known metrics. Furthermore, we report that by trimming redundant code, 48.6% of the known security vulnerabilities in the Java Runtime Environment JRE 6 update 45 has been removed.

## 1.3 Software Bloat in Mobile Applications

Smartphones are widely used in our daily life. A large number of Android applications that provide versatile functionalities, as well as CPUs with more computation powers allow and encourage users to install more and larger Android applications in their smart phones. At the meanwhile, as a considerable amount of software bloat in the apk file of each installed Android application, the resources of a smart phone, such as the storage and network bandwidth, has become more and more insufficient than before. Furthermore, software bloat in Android applications may also bring in other security concerns which are challenging to foresee.

Compared with general software running on a server or a desktop, mitigating software bloat issue in mobile applications is a brand new problem. Its uniqueness is, in part, due to its specialized operating systems, the different application execution model and life cycle, diverse usage contexts, the fragmented ecosystem, and more hard limits on resources. Here are some examples. An Android application has multiple execution entries instead of a single main method, which makes the traditional software analysis framework cannot be directly applied. The way that

a user interacts with a mobile device makes mobile applications use callbacks intensively. Mapping call backs and their registrations becomes another challenge to overcome. Coexisting different versions of operating systems, various CPU architectures, diverse display settings force mobile applications to include numerous files to achieve compatibility. The redundancy caused by this fact is also rare in other types of software. Hence, it is worthy to pay attention to Android application software bloat, which not only requires new perspective to classify and identify different types of bloat, but also new technical solutions to trim them off.

In this study, we investigate software bloat that would lead to the rapid size increase of Android applications. We categorize the sources of Android application software bloat into two types, compile-time redundancy and install-time redundancy. We further propose a fully automated approach to trimming off both types of software bloat. Our approach is mainly based on static analysis and program transformation.

For the compile-time redundancy, we statically construct an overapproximate call graph for the Android application being analyzed. Based on this call graph, we can remove the methods and classes that are never used in this call graph. Our approach overcomes several unique challenges in Android application static analysis and call graph construction, including multiple entries of an Android application, intensive usage of call backs, and Android component life cycles. Our approach processes reflections based on static string value analysis without the aid of other information besides the application code. As for the install-time redundancy, we discuss the presences and solutions towards two pervasive redundancy sources, which are multiple Software Development Kits (SDKs), and embedded Application Binary Interfaces (ABIs).

We have implemented our approach in a prototype called RedDroid and evaluated RedDroid on 4,779 Android applications from Google Play. We measured the impact on code sizes, code complexity, reflection call sites, the size of redundant SDKs, and the size of redundant embedded ABIs.

Our experimental results show that, by removing compile-time redundancy solely, on average, around 15% of the original application code can be trimmed off. For the applications that have install-time redundancy caused by redundant SDKs, another 20% of its original size can be trimmed off on average. For applications that have install-time redundancy caused by redundant embedded ABIs, we can

trim off additional 7% on average. If an application has all types of redundancy mentioned above, then on average we can expect to reduce its size by 42%. We report that each Android application in our test set has on average 14.8 reflection call sites, and our evaluation also shows that the distribution of usage frequency of each reflective method is quite biased. Furthermore, we report that code complexity, measured by a set of well-known metrics, is also notably reduced.

## 1.4 Software Feature Bloat

Another type of bloatware is caused by the feature creep phenomenon. The requirements of removing and customizing one or some of the bundled features from software products are raised from both developers and users, for both software engineering reasons and software security reasons.

From software engineering perspective, multiple reasons are accounted for feature creep. First, software products need to satisfy the various requirements of different groups of users at the same time. For a specific user in his or her specific work settings, many features offered by that software become redundant. Some software products have limited feature customization such as individual version and enterprise version which are coarse-grained. Second, when developers build new system based on legacy projects or third-party library, they tend not to modify or delete unnecessary features from those tangled code which is not written by themselves because making any change costs a lot and is prone to error. Third, the pressure of keeping updating and pushing new product to the market may drive developers to add gaudy features to the software which might be undesired by all groups of users.

Taking Microsoft Word as an example. Besides the function of text editing, users actually can use MS Word to read and send email. However, people seldom use MS Word in this way. Many other software products encounter the similar situations. According to a survey, on average, more than 45% functions of a software product are never used by most users [12].

Apparently, feature-creep bloatware causes many negative consequences. It has larger size, higher code complexity, and potentially is less reliable and secure. Developers are difficult to change, maintain, and manage the code. It also means longer testing time, more bug reports, and releasing patches in a higher frequency.

From software security perspective, a feature-based software customization is required to respond to several different threat models. First, it is not rare that some software vendors deliberately collect users' data via the backdoors which reside in some features of the software. Second, developers might need to reply on third-party libraries to build their own applications. The malicious third-party libraries will hurt both developers and users. In current trust model, to respond to these two scenarios, users or developers can use related analysis tools to scan the applications or libraries. Such a scanning helps improve our confidence on the integrity of the applications or libraries but cannot give a guarantee. Once the decision of using a specific application or library is made, we *have to* 100% trust the behavior of those code as a whole. If a feature-based customization solution is available, we do not need to fully trust the application or the library even if we choose to use them. By removing some user-identified unnecessary features which might have sensitive behavior such as writing data to Internet or logging user' profile, we can achieving active defense. In the third threat model, the adversary is the outside attacker. More features offer them a larger attack surface. Removing unnecessary features according to different requirements not only reduces the attack surface, but also achieves moving target defense by increasing software diversity.

Existing technologies and previous research cannot solve this problem very well. Code review can help mitigate these issues to some degree in some scenarios. However, in many cases code review is not a feasible solution. First, the cost of conducting comprehensive code review is high regarding to both time and expenses. Second, in many cases source code is not available for the code review. The effort of automatically removing bloat from bloatware has been made by some researchers. However, they did not touch the bloatware that caused by feature creep.

We propose an approach to customizing Java bytecode by applying static dataflow analysis and enhanced programming slicing technique. This approach allows developers to customize Java programs based on various users' requirements or remove unnecessary features from tangled code in the legacy projects. We evaluate our approach by estimating its algorithm complexity and conducting case studies on removing cross cutting features from real world Java program. The results show that our approach has the potential for practical use. Additionally, we find out that, by increasing the diversity of the software, our approach helps on achieving moving target defense.

## 1.5 Dissertation Structure

The dissertation is organized as follows. We present the related work in Chapter 2. We introduce our research on Java applications and the JRE unused code removal in Chapter 3. We report our efforts on Android application software bloat mitigation in Chapter 4. Chapter 3 and Chapter 4 share some similar insights. Therefore some interesting topics, open problems, and limitations shared by these two chapters are discussed at the end of Chapter 4 altogether. The feature-based software customization is reported in Chapter 5. We conclude in Chapter 6.

# Chapter 2
# Related Work

In this chapter, we first review the related work on bloatware. Then we discuss the work that supports, inspires, or is related to our approach and implementation.

## 2.1 Bloatware

### 2.1.1 Code Size Bloat

On Java program size reduction, Pugh [9] proposes a method to more efficiently pack class files into smaller jar files. Bradley et al. [10] introduce a new Java archive file format called Jazz, which has better compression ratio than the jar file format. Tip et al. [11] develop a tool called Jax to reduce Java archive size, especially applets, to shorten the download and transmission time over the Internet by removing redundant code. Jax takes advantage of manual annotations to handle reflections. Their measurements of interest in evaluation are also different with ours. These works do not consider the Java Runtime JRE customization as well. On the direction of user-input facilitated application extraction, Sweeney and Tip [13] further present a small, modular specification language MEL. Wagner et al. [14] takes a more aggressive step to remove code from those "always-connected" devices. They split code into a frequently used part known as hot code, and an infrequently used part known as cold code. A running device will only receive the hot code at the very beginning, while the cold code still remains on a remote server. The specific part of cold code will be transmitted to a running device only when it is necessary. Lint [15] is a tool to remove redundant registered resources from an Android project. Registered resources are located in "Res" directory of an Android

project. Each registered resource has a global unique ID, which can be directly referred by a static field of class R. However, a large number of resources, such as music, sprite-sheet-based images, animations, and movies that are located in directory "Asset", cannot be optimized by this tool, since they are referred in the program by using their relative path which is a string literal.

There is also research on program optimization and size reduction for other programming languages such as C++ and JavaScript. In practice, JavaScript code is usually optimized, compressed, and minified using compression [16] and minification [17] tools. Souders [18] suggests websites simply gzip all JavaScript components to save the loading time. However, transmitting compressed JavaScript may have some security problems because malicious JavaScript can be obfuscated by being compressed. Likarish et al. [19] raise a methodology to detect obfuscated malicious JavaScript by using classification techniques. Oberlander [20] discusses a method to analyze C++ source code to collapse class hierarchies. Sweeney and Tip [21] developed an approach to remove unused data members in C++ applications. De Sutter et al. [22] apply aggressive whole-program optimization and extensive code reuse on C++ binary to avoid bloat brought by templates and inheritance.

### 2.1.2 Memory Bloat

Regarding to the memory, Bu et al. [7] have pointed out that the negative impact on memory and performance caused by bloatware is being amplified by today's big-data software usage nature. Xu [23] proposes a method to reuse those redundant objects. Xu [24] also presents a tool called CoCo to soundly and adaptively replace Java collections to remove memory bloat from Java software. Hosking et al. [25] mitigate the problem of memory bloat by eliminating partial redundancy for access path expressions. Whitlock and Hosking [26] proposes a framework for persistence-enabled optimization of Java objects stores based on Bytecode-Level Optimizer and Analysis Tool (BLOAT). Xu et al. [27] presents a method to detect runtime bloat by applying abstract dynamic slicing technique. Nguyen and Xu [28] introduce Cachetor, a tool to detect cacheable data to remove bloat. In contrast, our tool JRed focuses on static unused code reduction, but takes a hybrid approach combining static reachability analysis and dynamic testing.

### 2.1.3 Dependency Bloat

A type of bloatware depends on too many libraries, contains cyclic dependency, or requires several incompatible at the same time which increases the cost of building, changing, and dependency management sharply . Morgenthaler et al. try to lower the difficulty of dependency management and target building caused by huge monolithic code base [29]. By removing the build files associated with dead code, identifying "unbuildable targets" and unnecessary command line flags, developers could ease the process of target building and pay down so-called "technical debt". They try to mitigate the problem without changing the code base. Wang et al. follow the similar approach and additionally includes code base itself into consideration [30]. They implement a tool to find out intra- and inter-module dependencies on both symbol level and module level. Developers can use those information to conduct large-scale refactoring on their huge code base. Vakilian et al. propose an approach to decomposing large build targets into smaller ones to avoid frequently triggering build and test tasks [31]. Ryder and Tip propose change impact analysis to precisely identify affected regression testing cases due to the change to the large code base [32].

### 2.1.4 Software Diversity

One of our research's security impacts is improving the software diversity. Software diversity enhances the software security from multiple aspects. Software diversity offers a probabilistic protection mechanism [33]. Additionally, it is a kind of active defense which can defend a wide range of types of attack, including unknown attack methods. Software could be diversified in different levels by different approaches. It offers a large design space to software diversity researchers. Our research offers one way to diversify software via feature-based customization. Other research in this area diversifies software by different granularities. Snow et al. diversify the software on instruction level [34]. Their approaches include equivalent instructions and equivalent instruction sequences substitution. Some other works diversify the software on basic block level [35]. Their technologies include opaque predicate insertion and branch function insertion. On the program level, approach instruction set randomization [36] and virtualization-based obfuscation are proposed. They are efficient on defending code injection attacks.

## 2.2 Others

### 2.2.1 Handling Reflections

Dynamic language features such as reflection pose a challenge to program analysis and optimization. Bodden et al. [1] use a dynamic approach to log the reflection usage information and then transform the reflection into a form that can be statically analyzed. Thies and Bodden [37] implement this technology on Eclipse to improve its code refactoring function. Livshits, Whaley and Lam [38] present a method to statically analyze Java reflection. An interesting point of their work is that they utilize the Java type casting information to better determine the type of the object that is just dynamically created. Braux and Noye [39] introduce a method to resolve reflections at compile time, even though their motivation is not to facilitate static analysis but to improve the performance. Christensen et al. [40] develop a general framework to analyze string expressions and applies the method to resolve Java reflections. Our tool JRed uses a hybrid static and dynamic approach, similar to that of Bodden et al. [1], to resolving Java reflections. Another approach to solving reflections in Java programs is to extract the string values in call sites of reflective calls. Several previous works proposed some methods to conduct string analysis. Java String Analyzer (JSA) [40] is a static analyzer to find the upper approximation values of given string variables in a program. Its first step is to transform Java program into a flow graph. An edge of this flow graph is a "def-use" chain in the program. In second step, JSA works on the flow graph to generate a regular expression to over approximate the values of a given string. Li et al. [41] proposed a new general framework to analyze string values in Java and Android program and implement a tool called Violist. They introduced a new IR which can be used to model string operations. By performing context-sensitive interprocedural analysis, Violist better solves the challenges, including scalability and string operations across procedures. Shannon et al. [42] introduces an approach to using symbolic execution to conduct string analysis. They take advantage of automaton to represent abstract string symbols in the symbolic execution.

### 2.2.2 Call Graph Construction

Call graph construction has a profound impact on the precision and effectiveness of program analysis and optimization. Lhoták [43] proposes a flexible points-to analysis framework for Java. Grove et al. [44, 45] find that context-sensitive call graph construction method does not gain much improvement on the results compared with context-insensitive methods. Agrawal et al. [46] develop a demand-driven technique for call graph construction. Tip and Palsberg [47] discuss several propagation-based call graph construction algorithms, and conclude that RTA costs less but yields similar results compared to other more expensive algorithms. The call graph construction method used by the analyzer of JRed is customizable. In our current implementation, we use points-to analysis, but it can be easily replaced by others.

### 2.2.3 Program Slicing

There has been a substantial amount of research on program slicing. Mark Weiser first raises the idea of program slicing, which could be applied to regression testing, program parallelization and automatic debugging [48]. Along with this idea, he also presents a static program-slicing algorithm. However, this algorithm could only be applied to a program that has a monolithic procedure. Arvind and Shankar presented a methodology to use program-slicing technology to facilitate regression testing [49]. Specifically, their methodology is based on alias analysis and an interprocedural program slicing algorithm proposed by Horwitz et al. K.J. Ottenstein and L.M. Ottenstein developed the program dependence graph (PDG) [50]. This data structure provides an infrastructure to develop a new more effeicent program slicing algorightm. But PDG only facilitates intraprocedural program slicing analysis. Horwitz et al. solved the problem of interprocedural program slicing. They introduce a new form, system dependence graph (SDG), to represent the program [51]. The challenge of conducting interprocedural slicing is analyzing calling context of procedures. Compared with PDG, this approach overcomes the difficulty by importing transitive dependences relationship. Some other researchers make efforts on dynamic slicing techniques. Wang and Roychoudhury implement a Java dynamic tool called JSlice [52]. The strength of this tool is that huge bytecode traces could be represented in an very effective manner. Hammacher implements

a tool called JavaSlicer [53], which is easy to set up and use. To make this tool Java virtual machine implementation independent, the author takes advantage of Java agent technology. Treffer and Uflacker also implement Java dynamic slicing on soot framework [54].

### 2.2.4 Static Analysis on Android

Cao et al. [55] proposed a comprehensive approach to analyzing all implicit control flow transitions (a.k.a callbacks) through the Android framework. More specifically, by performing backward data flow analysis starting from all methods that can be overridden in user space on an overapproximated call graph, a superset of all call backs and their registrations can be reached. They implement this method into a tool called EdgeMiner to augment the precision of existing static analysis tools. FlowDroid [56] is a state-of-the-art static taint analysis tool on Android applications. The on-demand analysis algorithm allows their approach to achieve high precision (context, flow, field, and object sensitive) with relatively low cost. Octeau et al. [57] implemented Dare, a tool to retarget Android Dalvik bytecode to Java bytecode. They present an inference algorithm to investigate the lost information (e.g., type information) during the process of transforming Java bytecode to Dalvik code. Their approach is based on the Tyde IR and 9 basic transformation rules. Dex2jar [58] is the other widely used open-source tool to transform Dalvik code into Java bytecode. Nimbledroid is a online tool to quickly profile an Android application. It is capable of being integrated with Continuous Integration (CI) process of an industry-strength Android application development. PScout [59] and Stowaway [60] are two static analyzers that map Android framework APIs to Android permissions. PScout first checks permission check points. Then it performs backward reachability analysis to the Android framework APIs that triggered those permissions checking. Intents sending and content providers accessing are considered as two types of implicit permission checking points. Undocumented Android framework APIs are also included in their results. Apktool [61] is a tool to conduct reverse engineering on Android applications. It can transform the Dalvik code to classes in smali representation. In addition, it can decode binary-based resource files back to its original human-readable form. FernFlower [62] is a state-of-the-art Java decompiler. It has rich command line options which makes it easy to be embedded into scripts

and existing tool chains. FernFlower is the default Java decompiler of IntelliJ integrated development environment.

# Chapter 3
# Program Customization Using Iterative Hybrid Static and Dynamic Analysis

## 3.1 Introduction

Modern software engineering practice increasingly brings redundant unused code into software products, which has caused the problem of unused code bloat, leading to software system maintenance, performance and reliability issues as well as security problems. With the rapid advances of smart devices and a more connected world, it is never more important to trim bloatware to improve the leanness, agility, reliability, performance, and security of the interconnected software and network systems. Previous methods have limited scopes and are usually not fully automated.

In this chapter, we propose a fully automated hybrid static and dynamic approach to trimming unused redundant bytecode from both Java application and Runtime JRE library code. We first construct a call graph for the target Java application or library code, using static program analysis. Based on the call graph, we perform a conservative reachability analysis for used methods and classes to identify unused ones. Those unused methods and classes are marked for potential trimming. Due to the reflection and heterogeneous development interface such as Java Native Interface for using native code, purely static program analysis sometimes cannot precisely determine method invocations or path feasibility. We resort to dynamic methods for help. To this end, we design a reinstater which

drives a test suite on the tentatively trimmed lean Java code to catch an missing method or class exceptions until it regresses. This hybrid static program analysis and dynamic testing combination has enabled quick prototyping of our tool and resulted good performance. Our approach is not intended to be a general approach that can be applied anywhere in any scenario. Instead, we believe our approach could be very useful and effective in certain applications and scenarios. For example, part of our contribution is the JRE customization. For those computing environments that only run one Java program such as sensors, wearable devices, GPS navigator, some embedded systems, and certain security surveillance systems, this JRE customization could be a desired feature.

We have implemented our approach in a prototype tool called JRed on top of Soot [63]. We have evaluated JRed on the DaCapo benchmark on code size, code complexity, memory footprint, execution and garbage collection time, and security attack surface. Our experimental results show that JRed is quite effective for practical use.

JRed reduces the size of the Java application code on average by 44.5%. JRed reduces the size of the Java Runtime JRE core library rt.jar by as much as 94.9%. JRed, from the end user point of view, reduces the device disk footprint by roughly 50%. Based on the 8 code complexity metrics including CK Java Metrics, JRed reduces the code complexity of both Java application and Java Runtime JRE core library rt.jar significantly. JRed trims nearly half of the known security vulnerabilities in the specialized Java Runtime JREs for each benchmark programs. Since unknown vulnerabilities are trimmed as well, this roughly leads to reduced attack surface by 50%. By specializing Java Runtime JRE for different applications, we can achieve more diversity, resulting enhanced moving target defense [64].

In summary, we make the following contributions:

- We propose an automated iterative hybrid static and dynamic approach to trimming unused code.
- We have implemented this method in a tool called JRed. Our experimental results show that JRed can significantly reduce code size, code complexity, and attack surfaces.
- Our results also quantitatively unveil the proportion of bloat existing in Java applications and JRE.

Table 3.1: Case study on library and application class and methods actually used by Catalina

| | Methods | Lines |
|---|---|---|
| **java.lang.String** | | |
| All Methods of Class | 78 | 1,099 |
| Methods Ever Called by Catalina | 62 | 890 |
| Called Methods/All Methods | 79.5% | 81.0% |
| **java.lang.Integer** | | |
| All Methods of Class | 36 | 473 |
| Methods Ever Called by Catalina | 14 | 156 |
| Called Methods/All Methods | 38.9% | 33.0% |
| **catalina.connector.Request** | | |
| All Methods of Class | 143 | 2,872 |
| Methods Ever Called by Catalina | 102 | 1,961 |
| Called Methods/All Methods | 71.3% | 68.3% |
| **catalina.core.ApplicationContextFacade** | | |
| All Methods of Class | 25 | 402 |
| Methods Ever Called by Catalina | 2 | 29 |
| Called Methods/All Methods | 8.0% | 7.2% |

- We build specialized Java Runtime JREs for different Java applications, enabling more software diversity and enhanced moving target defense.

The rest of this Chapter is organized as follows. We present a case study in Section 3.2. We introduce our approach and implementation in Section 3.3. The experimental results are reported in Section 3.4. We discuss a few related issues and limitations in Section 3.5.

## 3.2 Example

To investigate how many methods of a class are actually invoked by a large real-world Java application. we conduct a small case study[1] on Catalina, a Servlet container, which is a core sub-project of the Tomcat web server. We select two JRE core library classes, java.lang.String and java.lang.Integer, and two application

[1]We initially built a tool based on Joeq [65] to conduct this case study. Our current prototype implementation JRed is based on Soot [63]. We utilize the type parsing capability of Joeq to run our conservative analysis. The case study is run on a machine with Intel Core2 Duo 3.16 GHz CPU and 4G RAM.

classes, org.apache.catalina.connector.Request and org.apache.catalina.core.ApplicationContextFacade. The String class is frequently used by almost every Java program. In addition, its class hierarchy is quite simple, which only has one super class, java.lang.Object. Developers cannot extend the String class because it is final. The data of String class represents an rough upper bound of JRE class methods usage. The class java.lang.Integer is also widely used by many projects. However, most projects usually only use a few methods of Integer (e.g., Integer.parseInt). We expect that even a big project may only call a small portion of Integer methods, which represents roughly normal cases of most library classes. Based on the same principles, a frequently used application class (Request) and a less frequently used class (ApplicationContextFacade) are chosen.

The results of the String class are shown in the first row of Table 3.1. There are about 79.5% of String methods are actually called in Catalina. The results of the Integer class are shown in the second row of Table 3.1. Among all 36 methods of class Integer, 14 methods are actually called by Catalina, which means only 38.9% of Integer methods are used. Class Integer is a representative of most typical library classes that are instantiated in Catalina project. Therefore, we can roughly conclude that typically there are only about 40–80% of methods of library classes that are actually used, which points to large rooms for software customization and specialization.

We repeat the same experiment on the two application classes, Request and ApplicationContextFacade. The class Request is frequently and comprehensively used in Catalina. Thus it may represent a rough upper bound of method usage in application classes. The analysis result is shown in the third row of Table 3.1. There are 143 methods in total, among which 102, or 71.3%, are actually used in the project. For the less frequently used class AppliationContextFacade, there are 25 methods in total, among which only 2 methods are called. More than 90% methods of this class could be deleted.

The data on lines of source code show similar experimental results, on both library and application classes. The preliminary study results confirm that, for both library and application classes, there are the opportunities of software customization and specialization through trimming redundant unused code.

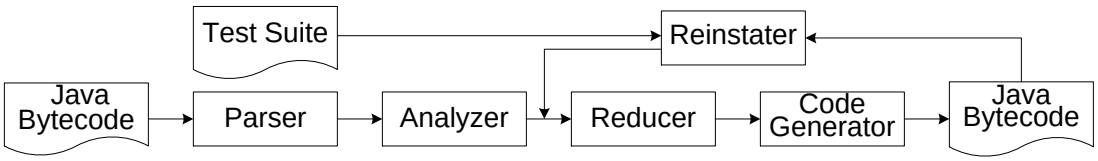



Figure 3.1: JRed Architecture

## 3.3 Approach

We first present the overall architecture of JRed, and then describe the details of the individual components.

### 3.3.1 Overview

Figure 3.1 shows the architecture of JRed. JRed transforms a Java application and the entire JRE into a redundant-code-free version. The input of JRed is a runnable Java program in bytecode. The first component, the parser, reads the bytecode of the Java program. It transforms the Java bytecode into the Soot intermediate representation (IR), Jimple [66]. The analyzer then conducts the analysis based on the Jimple IR. Taking the Java main method as the root node, the analyzer builds a call graph statically through the interprocedural points-to analysis. Call graph contains the information of the used methods and the classes that include those methods. As the output of the analyzer, the call graph is passed to next component Reducer. By iterating all the classes in the IR, the reducer checks if the methods of each class are nodes of the call graph. If a method is not presented in the call graph, the reducer rewrites the IR of the class to delete this method. If a class has no methods being used and no static field being accessed, then the entire class is removed. In the next step, the code generator transforms the customized IR back into the Java bytecode.

However, due to the reflections, some method invocations cannot be determined statically. We use dynamic testing to remedy the problem. The reinstater component of JRed takes advantage of dynamic testing to overcome static uncertainty. It drives the lean Java program and customized JRE to run on the test suites to check if the program can pass the tests. If the program fails, it will throw an exception, NoSuchMethod or AbstractMethod. Reinstater first catches and parses the exception message, and then wakes up the component Reducer by sending the

error information to it. The Reducer then adds the missing method back to the reduced IR, and the process repeats until the entire test suite passes. This hybrid static and dynamic approach has ensured our tool work correctly in practice.

### 3.3.2 Analyzer

Methods in object-oriented (OO) languages usually are encapsulated by classes. To build call graph for an OO language, we need to additionally collect class hierarchy and class instance reference information to determine methods override.

Due to the class hierarchy, there are cases where we cannot determine that the callee information hierarchy statically. A straightforward solution is to add the methods of all classes in the same class hierarchy into the used method worklist. But even with this conservative approach, we still need basic class hierarchy information.

We use SPARK, a flexible points-to analysis framework for Java [43], to facilitate call graph construction. Points-to analysis builds the call graph on the fly [67]. Compared with some other popular call graph construction techniques, such as Class Hierarchy Analysis (CHA) [68] and Rapid Type Analysis (RTA) [69], points-to analysis builds a more precise call graph. Points-to analysis for Java is different than for C [43]. SPARK points-to analysis takes advantages of the Java language features such as type-safety to collect more information for building call graph. Then we take a conservative approach to code trimming based on the call graph. Our analysis is not context or path sensitive. In the evaluation section, we will show that even with this conservative analysis, we can achieve considerable rate of code trimming.

To remove unused classes, it is unnecessary to conduct a complete class usage analysis since the basic loading unit of Java is a class and the methods reside in classes. If a method of a class is determined to be actually used, then the class that encapsulates this method must be retained as well. When an object is initiated, the constructor of the corresponding class must be called. In addition to methods, static class fields may prevent a class from being trimmed as well. Static class fields of a class can be accessed through the class name directly without object instances initialization.

### 3.3.3 Reducer

The Reducer deletes unused methods and classes from IR based on the analysis results. It takes two steps. First, the reducer deletes unused methods. We iterate through each loaded class and the methods in those classes. If a method is used, as a node of the call graph, it is kept untouched. If a method is not in our used method set, we mark it as a potential candidate for trimming. To ensure the correctness of trimming, we cannot delete the method right away since some method invocations cannot be determined statically. Although precisely determining all reflection method invocations statically is undecidable, it is possible to over-approximate the problem and trim the code conservatively. We thus ensure the correctness of the resulted lean Java code.

To this end, we adopt a set of "make-it-sounder" rules. The first rule is to keep all the constructors for those used classes. Reflections allow developers to dynamically create an instance of a class or invoke a method. However, in practice, the cases of dynamically creating an class instance are much more than the cases of dynamic normal methods invocation. This fact indicates that constructors have much higher chance to be used in reflections than other normal methods. Considering the fact that constructors are small proportion of all methods and each class only has a few usually, we make JRed keep all of these constructors to reduce the incorrectness caused by reflections.

The second rule is not to delete native methods. Native methods offer an interface to call those functions written in C via Java Native Interface (JNI). Reserving the native methods and setting the analysis boundary at the native methods avoids making our tool analyze native code and reduces the complexity of the analysis. This is a tradeoff for our prototype implementation. In the future, it would be interesting to investigate the whole system including native code. The third rule is not to delete any methods of classes that are loaded before the start of the Java main method. JVM executes a routine program to bootstrap necessary running environment before executing the main method of the program. The entry of our static analysis is the main method of the program. The number of classes that are loaded before the main method are only 315. Compared with more than 18,000 classes in the Java runtime `rt.jar`, keeping these 315 classes does not affect the overall results much.

If a method is neither a node of the call graph nor qualified for any of these "make-it-sounder" rules, it is then trimmed. The next step is to trim unused classes. If a class has no method or static field being actually used, then the entire class is trimmed.

### 3.3.4 Reinstater

Due to the reflections, some method invocations and class initializations cannot be determined by static analysis. We design a reinstater to mitigate this issue. The reinstater uses a dynamic approach to discovering the methods and classes that are incorrectly trimmed by our tentative results from the static analysis. Specifically, the reinstater is a driver to run the lean Java program based on a test suite. This is in spirit similar to regression testing. A similar hybrid approach has been used to solve problems such as reflection analysis [1]. We adopt it to make our approach sounder.

Missing classes and methods will cause three kinds of exceptions: ClassNotFoundException, MethodNotFoundException, or AbstractMethodException. Whenever the program running halts due to one of these exceptions, the reinstater will catch and parse the exception information. If the exception is caused by method missing, then the trace, class, and method information will be logged. The call graph will then be updated according to the logs. If the exception is ClassNotFoundException, then the full name of the class will be logged. After the missing code information is collected, the reinstater will wake up the reducer and pass these messages to it. Then the component reducer and the code generator together will generate a new lean Java bytecode program based on new information. The reinstater corrects one method or one class per run, and keeps driving the output of code generator until no exception is caught during running.

### 3.3.5 Other Issues in Implementation

Our implementation is based on Soot [63], which is a framework to annotate, optimize and analyze Java bytecode. Here we elaborate a few implementation challenges that are particular to our research.

#### 3.3.5.1 Resource Files

Good software engineering practice should separate resource files and programs (e.g., in the "bin" and "resource" folder, respectively). However, some programs in real world mix them together in the "bin" folder. The existence of these resource files raise challenge for our analysis. In these cases, JRed has to extract the resources files first before analysis, and at the end of the analysis, merge them with the transformed class files.

#### 3.3.5.2 Customized Exception Information

In normal cases, class missing or method missing should cause the throwing of either ClassNotFoundException, MethodNotFoundException, or AbstractMethodException. The limited number of the types of exceptions makes it easy to implement reinstater because we can simply design some routines to handle these expected exceptions. However, some programs do not directly throw these standard exceptions but handle them internally in a customized way or throw some customized exceptions. For example, in the benchmark program Batik, because of reflection, the first round analysis did not transform the class WriterAdapter. Rather than throwing this exception in a standard way, it prints out "Could not write PNG file because no WriterAdapter is available". Such a specialized exception handling method usually provides more information than those standard exceptions and can better help users and programmers to debug. However, in our case, those customized information and exceptions bring challenges to the automation of reinstater. The reinstater has to handle more types of exceptions, some of which are rather ad-hoc. In some extreme cases, iterative manual intervention is needed.

## 3.4 Evaluation

In this section, we evaluate JRed by applying it to 9 Java programs selected from DaCapo 9.12-bach benchmark [70]. Our experiments were conducted on HP SL390S G7 servers high performance computing cluster with 12 Intel X5670 2.93 GHz processors and 48G Memory. The operating system is Red Hat Enterprise Linux Server release 5.10 (Tikanga). The Linux kernel version is 2.6.18. We use JRE 6 Update 45 as our Java running environment.

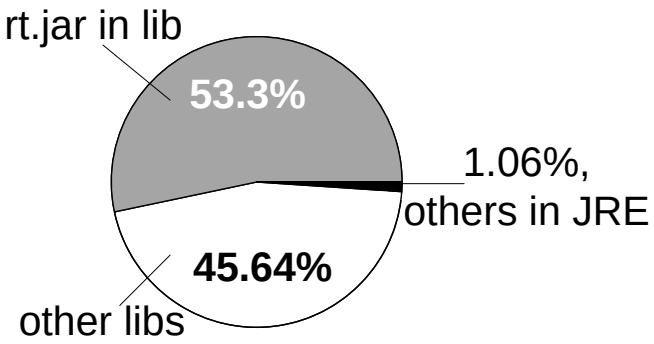



Figure 3.2: The Java Runtime JRE Structure

First we clarify the scope of our evaluation. A mobile device, desktop, or server that runs Java program is composed by three software entities; the OS, the Java Runtime JRE, and the Java application. Our goal is to evaluate the impact of our redundant code trimming technology on the Java application, JRE core libraries, and Java app+JRE. We define the Java app+JRE as the combination of Java applications and the whole JRE which consists of JRE core libraries, Java executable, and other supported files. The OS is out of the scope.

To evaluate JRed, we would like to answer the following research questions.

**Q1:** What is the impact of our redundant code trimming technique on the size of the Java application, the JRE core libraries, and Java-app+JRE together?

**Q2:** What is the impact of our redundant code trimming technique on the code complexity of the Java application and the JRE core libraries?

**Q3:** What is the impact of our redundant code trimming technique on the memory footprint?

**Q4:** What is the impact of our redundant code trimming technique on the Java application execution and the garbage collection time?

**Q5:** What is the impact of our redundant code trimming technique on the software reliability and security?

### 3.4.1 Code Size

In this subsection, we present the experiments to the answer research question Q1, the impact on the size of Java application, Java Runtime JRE, and all together Java App+JRE.

#### 3.4.1.1 Java Application Code Size

The experimental results are shown in Table 3.2 and Figure 3.3. A Java program, consisting of a group of class files that contain Java bytecode, could be stored in two different forms. The first form is that all class files are packed into a jar file. The second form is that all class files are unpacked. For each benchmark program, Table 3.2 and Figure 3.3 show the reduced-original size ratio in three different metrics. The first metric, reduced-original jar file size ratio, measures the impact of code reduction on the program as a jar file. The second and third metrics measure the unpacked cases. The second metric is the sum of the sizes of all class files. The third metric is the size of all class files that actually occupy on the disk. The jar file size metric is the most important metric among all the three metrics, since the jar file is the most common form as which a Java program exists. The second metric and the third metric have subtle difference. Sum of the size of all class files equal to adding every file's bytes number. The size of all class files that actually occupy on the disk usually is bigger than the number of bytes they have. The reason is that the basic unit of hard disk is a "sector" rather than a byte. If the size of a file is 1 byte, then the size it actually occupies on the disk is the size of a disk sector. In Table 3.2, the column "original" presents the data of original size of each benchmark program in these three different metrics. The column "reduced" shows the size of reduced version benchmark or say the left size after trimming. The column "Reduced/Original" showcases the number of reduced version size divided by original size.

In Table 3.2 and Figure 3.3, among the 9 benchmark programs, the lowest reduced-original jar size ratio is 30.08% (lusearch) and the highest one is 80.14% (sunflow). The median number is 54.53%. On average, the reduced-original jar size ratio is 55.52%. 4 out of 9 benchmark programs' jar files could be trimmed more than half off, 6 out of 9 could be trimmed more than 40% size off. The lowest reduced-original sum of all class files size ratio is 21.18% (luindex) and the highest

is 77.11% (h2). The median number is 51.46%. The average number is 63.12%. The lowest reduced-original all class files on-disk size ratio is 46.40% (xalan) and the highest is 87.53% (avrora). The median number is 65.00%. The average number is 67.13%. The results show that typically we can trim more than 40% size on average for the Java applications when they are in packed forms. These results have a significant impact on, for example, smartphone app download and installation time.

#### 3.4.1.2 Java Runtime JRE Code Size

The Java Runtime JRE usually contains four folders: bin, javaws, lib, and plugin. The folder plugin stores plugin files. The folder javaws contains files related to Java Web Start (JavaWS). The two most important folders are bin and lib. Bin includes the Java executable (the console command `java`). Lib contains the JRE core libraries, extension libraries, and other supported files. Figure 3.2 shows the structure of a JRE. The lib folder solely occupies nearly 99% of JRE in size. In this sector, a single jar file, rt.jar, occupies 53.30% in size. Excluding rt.jar, other jar files, shared object files (.so), and supported files (e.g., property files) comprise the rest 45.46% in size. In addition, rt.jar contains the most frequently used packages such as java.lang, java.util, java.io, and java.math. So in the evaluation of JRE core libraries trimming, we select rt.jar as a representative of the whole JRE core libraries. Considering that other libraries in the JRE usually are Java extensions which are designed for specified case rather than general usage like rt.jar, the ratio of the size that we can trim from those libraries should be higher than that of rt.jar.

The experimental results are presented in Table 3.3 and Figure 3.4. The metrics we used in Table 3.3 and Figure 3.4 are the same as the metrics we have used in Java application size measurement. Although all customized rt.jar files are different, the original rt.jar that we trimmed from is the same. So compared with 3.2, we do not have a column "original" in Table 3.3, but we list the data of original rt.jar under the main table.

On rt.jar, the lowest reduced-original jar file size ratio is 5.11% (avrora) and the highest one is 26.15% (fop). The median number is 17.52%. On average, the reduced-original rt.jar file size ratio is 17.45%. No one is higher than 30%. The lowest reduced-original sum of all extracted class files from rt.jar size ratio is 30.72% (xalan) and the highest one 48.52% (fop). The median number is 34.32%. On

average, it is 32.73%. The lowest reduced-original all extracted class files of rt.jar on disk size ratio is 11.71% (avrora) and the highest one is 58.24% (fop). The median number is 38.18%. The average number is 38.24%.

The proportion of the size we can trim from both applications and rt.jar is quite significant. Meanwhile, it is not surprising to see we can trim more on rt.jar as it is a general runtime library. The cohesion of each component inside an application is relatively higher than the cohesion between an application and the JRE core library, and the cohesion of different packages in JRE core libraries. This result meets our assumption that a more general design and a higher abstract level lead to the code with more bloat.

#### 3.4.1.3 Java App+JRE All Together

We define the application and JRE (not only the core libraries in JRE) together as "Java-App+JRE". Figure 3.5 shows the experimental results on Java-App+JRE for the 9 benchmark programs. The Y axis is the size (MB) of Java-App+JRE for each benchmark program. For each benchmark, there is a higher bar and a shorter bar representing the size of original Java-App+JRE and the reduced lean version, respectively. In each bar, the dark gray part represents the size of the application and the light gray part represents the size of JRE. We can see that the left side light gray bar on each benchmark program has the same height, which means each application originally invoke the same JRE. The percentage number above each right-side shorter bar on each benchmark program is the reduced-original Java-App+JRE ratio. The light gray part of each right side shorter bar consists of two parts: a reduced rt.jar and other files in lib folder that are not touched in this experiment. From Figure 3.5, we can see that a big JRE core library (93.9MB) causes the sizes of all Java-App+JREs are around 100MB.

By comparing two bars of each benchmark program, we can see that after trimming, all Java-App+JREs roughly have half size of their original versions. If we additionally analyze and delete other Java bytecode in the lib folder, the percentage could be lower. The results show that JRed can significantly reduce the whole Java package Java-App+JRE by half on code size.

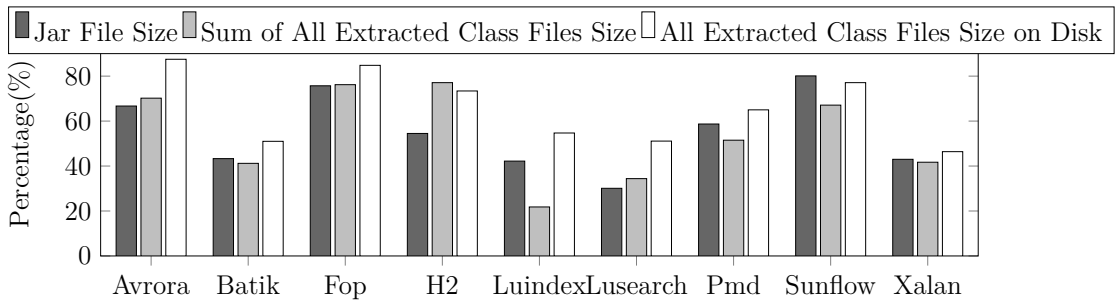



Figure 3.3: Reduced-Original Size Ratios of the DaCapo Benchmark Applications

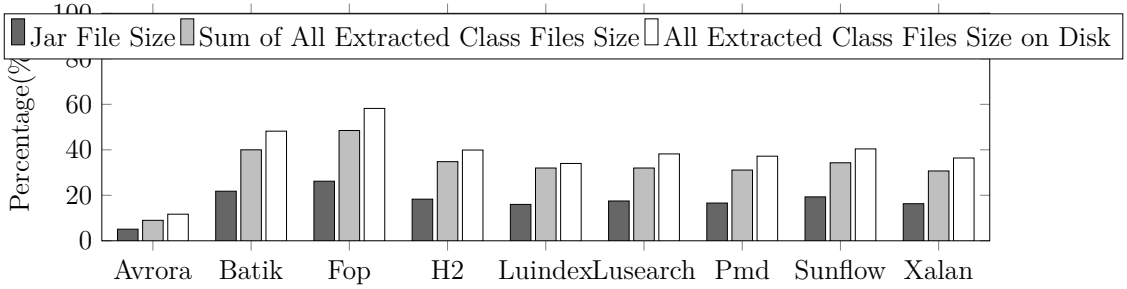



Figure 3.4: Reduced-Original rt.jar Ratios of DaCapo Benchmark Applications

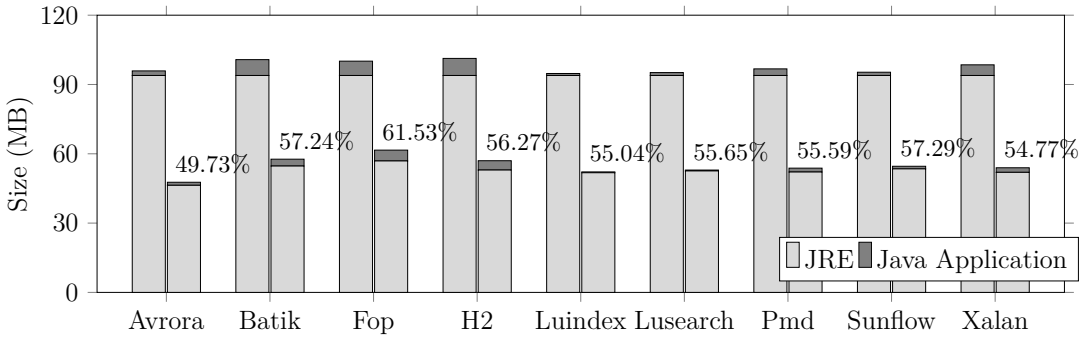



Figure 3.5: Java App+JRE Overall Ratios

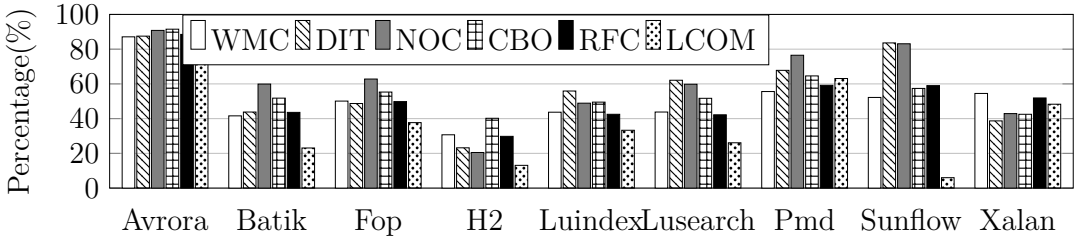



Figure 3.6: Reduced-Original Application CK Java Metrics Ratios

Table 3.2: DaCapo benchmark applications code size before and after unused code trimming comparison

| Benchmark | Size of Jar Files (MB) | | | Size of All Files (MB) | | | Size of All Files on Disk (MB) | | |
|---|---|---|---|---|---|---|---|---|---|
| | Original (MB) | Reduced (MB) | Reduced/Original (%) | Original (MB) | Reduced (MB) | Reduced/Original (%) | Original (MB) | Reduced (MB) | Reduced/Original (%) |
| avrora | 1.98 | 1.32 | 66.67 | 3.15 | 2.21 | 70.16 | 7.46 | 6.53 | 87.53 |
| batik | 6.86 | 2.97 | 43.29 | 13.70 | 5.64 | 41.17 | 25.90 | 13.20 | 50.97 |
| fop | 6.20 | 4.69 | 75.65 | 12.30 | 9.37 | 76.18 | 24.40 | 20.70 | 84.83 |
| h2 | 7.39 | 4.03 | 54.53 | 33.20 | 25.6 | 77.11 | 40.60 | 29.80 | 73.40 |
| luindex | 0.86 | 0.36 | 42.24 | 1.56 | 0.63 | 21.78 | 2.87 | 1.57 | 54.70 |
| lusearch | 1.26 | 0.38 | 30.08 | 1.83 | 0.63 | 34.43 | 3.31 | 1.69 | 51.06 |
| pmd | 2.86 | 1.68 | 58.74 | 5.46 | 2.81 | 51.46 | 11.80 | 7.67 | 65.00 |
| sunflow | 1.41 | 1.13 | 80.14 | 2.31 | 1.55 | 67.10 | 4.10 | 3.16 | 77.07 |
| xalan | 4.61 | 1.98 | 42.95 | 9.48 | 3.95 | 41.67 | 16.70 | 7.75 | 46.40 |
| average | 3.71 | 2.06 | 55.52 | 9.22 | 5.82 | 63.12 | 15.24 | 10.23 | 67.13 |

Table 3.3: Customized rt.jar of DaCapo benchmark applications code size before and after unused code trimming comparison

| Benchmark | Size of Jar Files | | Size of All Files | | Size of All Files on Disk | |
|---|---|---|---|---|---|---|
| | Reduced (MB) | Reduced/Original(%) | Reduced (MB) | Reduced/Original(%) | Reduced (MB) | Reduced/Original(%) |
| avrora | 2.56 | 5.11 | 4.25 | 47.20 | 10.80 | 11.71 |
| batik | 10.90 | 21.75 | 19.90 | 40.04 | 44.40 | 48.16 |
| fop | 13.10 | 26.15 | 22.90 | 48.52 | 53.70 | 58.24 |
| h2 | 9.17 | 18.33 | 16.40 | 34.75 | 36.81 | 39.91 |
| luindex | 8.00 | 15.97 | 15.10 | 31.99 | 31.29 | 33.95 |
| lusearch | 8.78 | 17.52 | 15.10 | 31.99 | 35.20 | 38.18 |
| pmd | 8.31 | 16.57 | 14.70 | 31.14 | 34.28 | 37.20 |
| sunflow | 9.67 | 19.30 | 16.21 | 34.32 | 37.21 | 40.35 |
| xalan | 8.17 | 16.31 | 14.50 | 30.72 | 33.62 | 36.44 |
| average | 8.74 | 17.45 | 15.45 | 32.73 | 35.26 | 38.24 |

| Original Size of rt.jar | Original Size of all files of rt.jar | Original Size of all files of rt.jar on disk |
|---|---|---|
| 50.1 | 47.2 | 92.2 |

Table 3.4: Java Application Code Complexity Measurements

| Benchmark | Avrora | | | Batik | | | Fop | | |
|---|---|---|---|---|---|---|---|---|---|
| | Original | Reduced | Reduced/Original (%) | Original | Reduced | Reduced/Original (%) | Original | Reduced | Reduced/Original (%) |
| WMC | 8377 | 7300 | 87.14 | 35272 | 14667 | 41.58 | 50324 | 25212 | 50.10% |
| DIT | 702 | 614 | 87.46 | 3748 | 1642 | 43.81 | 4722 | 2298 | 48.67 |
| NOC | 1010 | 971 | 90.79 | 1695 | 1015 | 59.88 | 2884 | 1812 | 62.83 |
| CBO | 10065 | 9211 | 91.51 | 18239 | 9439 | 51.75 | 31281 | 17296 | 55.29 |
| RFC | 19797 | 17514 | 88.46 | 82546 | 36022 | 43.64 | 133805 | 66587 | 49.76 |
| LCOM | 83590 | 61597 | 73.69 | 282950 | 65357 | 23.10 | 355659 | 134006 | 37.68 |
| Ca | 10065 | 9211 | 91.51 | 18089 | 9439 | 52.18 | 29586 | 17296 | 58.56 |
| NC | 1644 | 1528 | 92.94 | 4622 | 2455 | 53.12 | 6559 | 3856 | 58.48 |
| **Benchmark** | **H2** | | | **Luindex** | | | **Lusearch** | | |
| | Original | Reduced | Reduced/Original (%) | Original | Reduced | Reduced/Original (%) | Original | Reduced | Reduced/Original (%) |
| WMC | 22454 | 6885 | 36 | 5124 | 2241 | 43.74 | 5136 | 2249 | 43.79 |
| DIT | 1433 | 333 | 23.24 | 438 | 245 | 55.93 | 441 | 274 | 62.13 |
| NOC | 1051 | 215 | 20.46 | 284 | 139 | 48.94 | 286 | 171 | 59.79 |
| CBO | 10431 | 4196 | 40.23 | 2705 | 1339 | 49.50 | 2718 | 1406 | 51.73 |
| RFC | 66258 | 19734 | 29.78 | 12958 | 5502 | 42.46 | 12986 | 5481 | 42.21 |
| LCOM | 607960 | 79493 | 13.08 | 28280 | 9411 | 33.28 | 28280 | 7385 | 26.11 |
| Ca | 9593 | 4196 | 43.74 | 2684 | 1339 | 49.89 | 2697 | 1406 | 52.13 |
| NC | 2118 | 498 | 23.51 | 638 | 343 | 53.76 | 17518 | 7441 | 42.48 |
| **Benchmark** | **Pmd** | | | **Sunflow** | | | **Xalan** | | |
| | Original | Reduced | Reduced/Original (%) | Original | Reduced | Reduced/Original (%) | Original | Reduced | Reduced/Original (%) |
| WMC | 19525 | 10856 | 55.60 | 4828 | 2520 | 52.20 | 25574 | 13937 | 54.50 |
| DIT | 1788 | 1212 | 67.79 | 562 | 470 | 83.63 | 2960 | 1144 | 38.65 |
| NOC | 913 | 698 | 76.45 | 219 | 182 | 83.11 | 1158 | 497 | 42.92 |
| CBO | 10434 | 6737 | 64.57 | 4203 | 2412 | 57.39 | 15037 | 6387 | 42.48 |
| RFC | 44041 | 26060 | 59.17 | 12617 | 7440 | 58.97 | 60528 | 31433 | 51.93 |
| LCOM | 307277 | 193909 | 63.11 | 121318 | 7267 | 5.99 | 316124 | 152573 | 48.26 |
| Ca | 10372 | 6737 | 63.95 | 4200 | 2404 | 57.25 | 15033 | 6387 | 42.49 |
| NC | 2369 | 1702 | 71.84 | 657 | 551 | 83.87 | 2806 | 1396 | 49.75 |

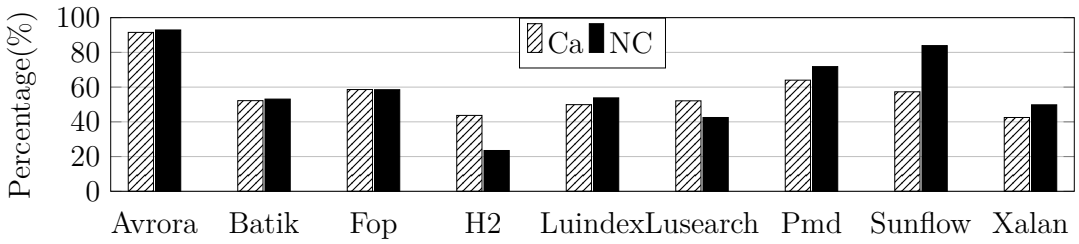



Figure 3.7: Reduced-Original Application LCOM and Ca Ratios

Table 3.5: The Java Runtime Rt.jar Code Complexity Measurements

| **Original rt.jar** | | | |
|---|---|---|---|
| WMC | 157,448 | RFC | 377,100 |
| DIT | 34,059 | LCOM | 2,564,567 |
| NOC | 17,505 | Ca | 46,346 |
| CBO | 46,385 | NC | 17,518 |

| **Benchmark** | **Avrora** | | **Batik** | | **Fop** | |
|---|---|---|---|---|---|---|
| | Reduced (MB) | Reduced/Original (%) | Reduced (MB) | Reduced/Original (%) | Reduced (MB) | Reduced/Original (%) |
| WMC | 17897 | 11.37 | 72806 | 46.24 | 82876 | 52.64 |
| DIT | 4472 | 13.12 | 14924 | 43.77 | 17983 | 52.74 |
| NOC | 2411 | 13.77 | 9392 | 53.65 | 11325 | 64.70 |
| CBO | 2721 | 5.87 | 21059 | 45.40 | 29311 | 63.19 |
| RFC | 42822 | 11.36 | 181002 | 48.00 | 210779 | 55.89 |
| LCOM | 122205 | 4.77 | 660102 | 25.74 | 713399 | 27.82 |
| Ca | 2721 | 5.87 | 21059 | 45.44 | 29154 | 62.91 |
| NC | 2411 | 13.76 | 9392 | 53.61 | 11325 | 64.65 |
| **Benchmark** | **H2** | | **Luindex** | | **Lusearch** | |
| | Reduced (MB) | Reduced/Original (%) | Reduced (MB) | Reduced/Original (%) | Reduced (MB) | Reduced/Original (%) |
| WMC | 64590 | 41.02 | 58735 | 37.30 | 58573 | 37.20 |
| DIT | 13895 | 40.75 | 13732 | 40.28 | 13693 | 40.16 |
| NOC | 7619 | 43.52 | 7432 | 42.46 | 7441 | 42.51 |
| CBO | 15313 | 33.01 | 12529 | 27.01 | 12552 | 27.06 |
| RFC | 162775 | 43.16 | 147044 | 38.99 | 146614 | 38.88 |
| LCOM | 669830 | 26.12 | 576999 | 22.50 | 573980 | 22.38 |
| Ca | 15313 | 33.04 | 12529 | 27.03 | 12552 | 27.08 |
| NC | 42378 | 38.87 | 7619 | 43.49 | 7432 | 42.42 |
| **Benchmark** | **Pmd** | | **Sunflow** | | **Xalan** | |
| | Reduced (MB) | Reduced/Original (%) | Reduced (MB) | Reduced/Original (%) | Reduced (MB) | Reduced/Original (%) |
| WMC | 57272 | 36.38 | 60422 | 38.38 | 56668 | 35.99 |
| DIT | 13547 | 39.73 | 14517 | 42.58 | 13426 | 39.38 |
| NOC | 7225 | 41.27 | 7744 | 44.24 | 7048 | 40.26 |
| CBO | 11792 | 25.42 | 13757 | 29.66 | 11099 | 23.93 |
| RFC | 143398 | 38.03 | 152322 | 40.39 | 141428 | 37.50 |
| LCOM | 571330 | 22.28 | 582335 | 22.70 | 570646 | 22.25 |
| Ca | 11791 | 25.44 | 13756 | 29.68 | 11099 | 23.95 |
| NC | 7225 | 41.24 | 38841 | 35.63 | 36465 | 33.45 |

### 3.4.2 Code Complexity

In this subsection, we present the experimental results to answer research question Q2, the impact of JRed on the code complexity of Java applications and runtime JRE. The results on Java applications are shown in Table 3.4 Figure 3.6, and Figure 3.7. We measure the code complexity by the Chidamber and Kemerer (CK)

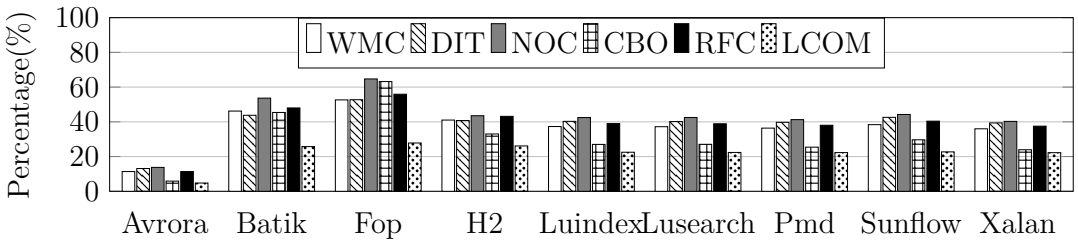



Figure 3.8: Reduced-Original rt.jar CK Java Metrics Ratios

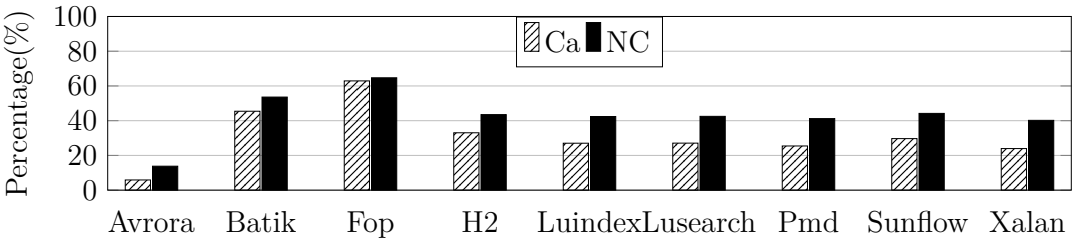



Figure 3.9: Reduced-Original rt.jar LCOM and Ca Ratios

object-oriented metrics and two other metrics. The CK object-oriented metrics suite is proposed by Chidamber and Kemerer [71, 72] for measuring the complexity of object-oriented software. It contains 6 metrics, including Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Coupling Between Objects (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM). These metrics are measured at the class level. In our experiment, we add the data of each class in an application or a JRE together to calculate the complexity of that application or JRE, because we want to do complexity comparison on the whole program level.

On WMC, we assign all methods the same weights, which means WMC simply indicates the total number of the methods of the classes. According to the study conducted by Misra and Bhavsar [73], the number of bugs are positively proportional to the average number of WMC. DIT indicates the number of parents a class has. A deeper inheritance tree may ease the OO design and software reuse. However, a deeper inheritance tree also involves more design complexity. NOC is the number of the *intermediate* subclasses a class has. Usually a big NOC is worse than a big DIT since the depth of class hierarchies promotes more reuse than the width. If a class has a big number of intermediate subclasses, then more classes will be affected when this class is changed and more testing is necessary.

CBO measures how intensively an object invokes or accesses the methods, fields or objects outside its own class inheritance hierarchy. Good software engineering design practice requires high degree of cohesion but low degree of coupling. Frequent inter-object reference usually breaks the modularity, decreases the chance of reuse, makes code less understandable, and requires more testing endeavor in general. RFC indicates the number of the methods of a class invoked from outside of this class. This metric could be understood as a passive version CBO. Similarly, a high RFC hints less understandable codes and demands higher testing effort in general. LCOM measures the cohesion of a class. Each method of a class $M$ operates on a set of class fields $F$, LCOM equals to the maximum number of the $F$ sets that are completely disjoint. A high LCOM indicates that the methods in a class operate on several separate data sets and share few common properties or functions. High LCOM usually is caused by incorrect methods, unnecessary methods or unused methods that are inappropriately encapsulated in a class. Low cohesion often makes a class unnecessarily complicated.

The first 6 rows of each sub-table in Table 3.4 show the experimental results of each benchmark application before and after JRed trimming on the CK Java metrics.

Figure 3.6 visualizes the reduced-original ratio on all six CK metrics. For the CK Java metrics, the results vary on different applications due to their own design nature. On benchmark H2, no reduced-original ratio among all 6 CK metrics is more than 40%. The reduction ratios on avrora are around 20%. In summary, all 6 metrics on all 9 applications are reduced significantly, resulting a reduced code complexity after JRed trimming.

Besides the CK Java metrics, we also measure two other metrics on code complexity. The first one is Afferent Couplings (Ca) [74]. It is the number of the methods of the classes in a specific package invoked by the classes in other packages. Ca is similar to RFC, but the granularity is coarser since it measures the inter-package couplings. The other one is the number of the classes. We have measured the total number of methods in the CK Java metrics. So we are also curious about how the number of classes changes before and after trimming. Figure 3.7 and the last two rows of each sub-table in Table 3.4 show the reduced-original ratio of these two more measurements. Overall, there are significant reduction on the metrics Ca and NC for all the 9 benchmark programs.

Table 3.6: Avrora Memory Footprint

| | | Original (MB) | Reduced (MB) |
|---|---|---|---|
| **Heap** | allocated all pools | 15.0 | 15.0 |
| | used survivor space | 0.3 | 0.3 |
| | used tenured space | 9.3 | 9.2 |
| **Non Heap** | allocated all pools | 35.0 | 35.0 |
| | used perGen[shared rw] | 7.3 | 7.3 |
| | used perGen[shared ro] | 7.4 | 7.4 |
| | used perGen | 3.5 | 3.4 |
| | code cache | 1.6 | 1.6 |

Table 3.7: Avrora Execution and Garbage Collection Time

| | Original | Reduced |
|---|---|---|
| full execution time(s) | 129.4 | 128.5 |
| GC time(s) | 1.8 | 1.9 |

By comparing Figure 3.6 and Figure 3.7, we can see that all 8 metrics are roughly positive proportional to each other. They together indicate we can reduce the code complexity from the original application. Overall, if the original program's design is compact and the project scale is limited, it usually contains less code bloat and low degree of code complexity. The complexity we can reduce is also related to the nature of the application functions.

The impact on the code complexity of JRE core libraries is presented in Figure 3.8, and Figure 3.9. Due to the page limit, we do not show the data of JRE core libraries' code complexity in table. We use the same metrics on JRE by comparing the data before and after unused code trimming. Again, compared with original JRE, customized JRE reduced the code complexity significantly. Compared with applications, the reduction proportion of code complexity in JRE is bigger.

### 3.4.3 Memory Footprint and Execution Time

In this subsection, we answer the research questions Q3 and Q4. We did experiments to compare the performance and the memory usage between each original Java application and its lean version. We select benchmark program avrora's memory footprint and execution time data here which is shown in Table 3.6 and Table 3.7. The lean version benchmark has slightly smaller memory footprint, but mostly

Table 3.8: Vulnerabilities Removed from the Customized JREs

| Benchmark | CVE-2013- | | | | | | | | | | | | | Trimmed | Trimmed/Original (%) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 2473 | 2472 | 2471 | 2465 | 2463 | 2461 | 2457 | 2454 | 2453 | 2452 | 2450 | 2448 | 2446 | 2444 | |
| **avrora** | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | ✗ | | ✗ | ✗ | ✗ | 13 | 92.9 |
| **batik** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **fop** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **h2** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **luindex** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **lusearch** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **pmd** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **sunflow** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **xalan** | | | | | | ✗ | ✗ | ✗ | ✗ | | | ✗ | ✗ | | 6 | 42.9 |
| **average** | | | | | | | | | | | | | | | 6.8 | 48.6 |

remains the same size as the original version. Since the JVM loads class files on demand, class trimming does not contribute to the reduction of memory usage. All memory usage savings are from unused method trimming: given the same number class files, lean version class files have fewer methods, which leads to less memory usage. In most Java application memory footprints, byte code only occupies a small portion, where the heap and stack occupy the most memory. In addition, to avoid frequently requesting memory allocation from system, rather than allocating memory on demand, JVM usually uses a more-than-enough memory (allocated all pools) to run the Java program to give flexibility to garbage collection. Due to these factors, JRed does not reduce memory footprint significantly. In our future work, we would like to consider trimming unused fields as well after unused method and class trimming, which might have more impact on the memory footprint as it affects the object sizes.

On performance, our measurement does not show significant improvement. This is mainly due to that JRed does not perform optimizations on the reduced code. However, JRed might potentially create more opportunity for the whole program optimizations. Also, due to the reduction of code size, the program loading and starting time can be significantly reduced and thus from end user point of view, JRed does improve the performance for certain Java applications. This might have a bigger impact in a smart device environment.

### 3.4.4 Security

In this subsection, we address the research question Q5. We surveyed all the known security vulnerabilities in the CVE database that affects Oracle JRE 6 update 45. In total, we found 14 security vulnerabilities reported, excluding the vulnerabilities

Table 3.9: JRed Performance

| **Benchmarks** | avrora | batik | fop | h2 | luindex | lusearch | pmd | sunflow | xalan |
|---|---|---|---|---|---|---|---|---|---|
| **Transformation Time (s)** | 92 | 303 | 454 | 116 | 59 | 107 | 103 | 148 | 106 |
| **Reinstatement Rounds** | 4 | 88 | 223 | 0 | 5 | 11 | 98 | 2 | 0 |

that only involve native code or do not offer enough Java code patch information. We then checked the number of those vulnerabilities that still exist in the customized Java Runtime JREs for each of the 9 benchmark programs. The results are shown in Table 3.8. For avrora, the specialized JRE only contains 1 vulnerability; the other 13 are trimmed. For others, JRed trimmed 6 out of 14. On average, JRed trimmed nearly half of the security vulnerabilities. By specializing Java Runtime JRE for different applications, we can achieve more diversity, resulting enhanced moving target defense [64].

### 3.4.5 Performance

In addition to the effectiveness evaluation, we also measured the running time performance for our tool JRed. We measure the performance of JRed from two perspectives. First, we measure how much time JRed takes to finish the transformation. The transformation time is sum of the time taken by the parser, analyzer, reducer, and code generator until the start of reinstater. Second, we measure reinstatement iteration rounds.

The results are shown in Table 3.9. The data is averaged over 10 runs. The transformation time on the 9 benchmark programs ranges from 1 to 7 minutes. The transformation time is related to two factors. The first one is the original size of the application. The transformation time is roughly proportional to application code size. For example, the original size of Fop (6.20MB) is 3.13 times to Avrora (1.98MB); the transformation time of Fop is 4.93 times to Avrora. The second factor is the cohesion of the application. For instance, if we only consider application size, then H2 would be an exception: the H2 original application size is 7.39MB but its transformation time is only 116s. By checking Figure 3.6 and Figure 3.7, we found that H2 is the one of the applications that are trimmed off most in terms of code complexity, which indicates H2 has relatively low cohesion. Overall, no transformation time is over 10 minutes on 9 non-trivial benchmark programs.

The data of reinstater iteration rounds is listed in the second row of Table 3.9.

The range of iteration rounds is from 0 to 223. Reinstater adds one missing method per iteration, so the number of reinstatement rounds is actually equal to the number of methods being incorrectly deleted by static analysis due to the reflection. On average, reinstater drives each benchmark to run 48 rounds on test suite to fix all incorrect trimming. Compared to 19,624, the number of methods each benchmark application has on average, 48 iterations indicate 1 per 410 methods might be incorrectly removed by static analysis due to reflection. The time of each round varies. An exception may be caught in 0.1 second after the start of a test round or happen during the time when the application has passed 99% of the test suite. The time of passing the entire test suite depends on the design of the test suite and the task nature of the application.

### 3.4.6 Experimental Result Summary

In summary, JRed is quite effective in trimming code size, reducing code complexity, and minimizing attack surfaces.

1. JRed reduces the size of the Java application code on average by 44.5%.
2. JRed reduces the size of the Java Runtime JRE core library rt.jar by as much as 94.9%.
3. JRed, from the end user point of view, reduces the device disk footprint by roughly 50%.
4. Based on the 8 code complexity metrics including CK Java Metrics, JRed reduces the code complexity of both Java application and Java Runtime JRE core library rt.jar significantly.
5. JRed trims nearly half of the known security vulnerabilities in the specialized Java Runtime JREs for each benchmark program. Since unknown vulnerabilities are trimmed as well, this roughly leads to reduced attack surface by 50%. By specializing Java Runtime JRE for different applications, we can achieve more diversity, resulting enhanced moving target defense [64].

## 3.5 Discussion

### 3.5.1 Code Trimming on Java Core Library

A consequence of trimming code from the JRE is that the customized JRE may not be capable of running other Java programs but only the application for which the JRE is customized. For those computing environments that run only one Java program, this would be a desired feature. First, JRE customization reduces the program size additionally. It is important to some scenarios where the resources is very limited. For example, the micro-sensor for the military usage and the endoscope for the medical care could save valuable resource from JRE customization. Additionally, each Java program is running with a specialized JRE. This can potentially increase the cost of cyber attacks as the same attack script work for one JRE will unlikely work on another specialized different JRE. We do not expect the JRE customization to be applied anywhere in any scenario. However, this feature could be useful and effective in certain applications and scenarios.

### 3.5.2 Soundness and Limitation of Dynamic Reflection Resolving Method

Our static analysis approach for code reduction is *sound* and *conservative* on most Java static and dynamic features including inheritance, polymorphism, and object reference except for some dynamic features such as reflection. The insight of using the dynamic approach to aiding static analysis is that a control flow graph edge coverage test suite, which is also known as branch coverage test suite in software engineering domain, can cover all intended reflection call targets. Therefore, if a branch coverage test suite is available, we can address the reflection issue safely. In addition, the data presented by Bodden et al. [1] shows possibility that we may loose the requirements on test suite coverage. Intuitively, the limitations of the dynamic approach come from two perspectives: how many reflection call sites could be covered by a well designed test suite, and how many reflection call edges could be explored by a well designed test suite. Someone may argue that, dynamic approach is not practical unless the test suite is designed with branch coverage requirements, which is not always the case. However, from the data of software

Table 3.10: Number of reflection call sites of selected benchmarks in DaCapo discovered by test suites of different sizes, cited from Eric Bodden et al. [1]

| | **Small** | **Default** | **Large** |
|---|---|---|---|
| batik | 41 | 44 | 44 |
| h2 | 31 | 31 | 31 |
| pmd | 32 | 32 | 32 |
| sunflow | 30 | 30 | 30 |
| xalan | 54 | 54 | 54 |

engineering practice, a reasonably-well-designed industry-strength test suite is good enough to achieve our goals. Eric Bodden et al. [1] reported the results shown in Table 3.10. From the table, we have the following observations. The number of reflection callees discovered by test suites quickly converges to a stable number and no longer increases with the expansion of the test suite. Just like many other properties in a program, the usage of reflections show its locality. Developers do not use reflection in random places which may cause poor performance of the program. Reflection usually is used in some hub-like components to make the design can be easily extended. A reasonable test suite will cover such a key part of the program. Except for the programs with intensive user interaction, the data of experiments on reflection callees exploration shows quick convergence feature [1]. Thus, our approach might encounter difficulties on transforming user-interaction intensive programs. But this dynamic reflection resolving approach can be applied to most programs.

# Chapter 4
# Android Application Customization and Redundancy Removal Based on Static Analysis

## 4.1 Introduction

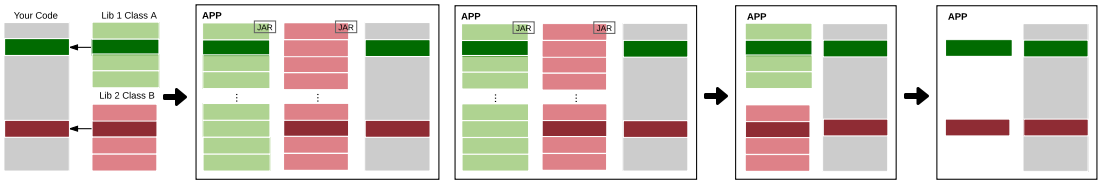
### 4.1.1 Two Types of Redundancy

Android applications contain software bloat due to multiple reasons. We categorize the software bloat into two basic types, compilation-time redundancy and installation-time redundancy. This categorization is based on the time when they can be determined as redundancy.

#### 4.1.1.1 Compile-time Redundancy

Modern software engineering rarely implements a software product from scratch. Developers are relying on different kinds of libraries and frameworks to finish their jobs. Libraries usually are implemented for a more general purpose, instead of the requirements from a specific group of developers. For example, an cryptographic library may contain the implementations of multiple crypto algorithms. However, developers would mostly stick to only one of them in their applications. In fact, it is very common to see only one method from one class in a large library is used by an application.

Java language compilation and runtime has neither "static link" nor "dynamic




(a) Modern Software Engineering Paradigm

(b) Remove Compile-Time Redundancy in Libraries

Figure 4.1: Motivation of RedDroid

link" in the terminology of standard program compilation. After each class of Java source code is compiled into bytecode, there is no static link process to include a library into a monolithic executable file. Making a jar file is simply a process of zipping every single class file in the working directory into one package. During runtime, each Java application runs in its own Java virtual machine. So two Java applications cannot share one copy of a dynamic library through memory mapping as executable files do. Accordingly, current development practice is to include each library entirely in the final software product delivery. Figure 4.1a illustrates this process. Gray box represents the code written by a developer herself. Green bar and red bar in the gray box indicate two method invocations from two classes, respectively. Used methods are highlighted from unused methods. When packaging this application, the jars that contain the classes we referred must be put in the build path of the application and packaged with the application code entirely.

The unused code in the libraries comprises a major part of software bloat in an application. The implementation of application code determines which part of the library code is used or not. Application code can be seen fixed after its compilation. So we categorize the redundancy, such as unused code, that can be decided by checking compiled code as compilation-time redundancy.

#### 4.1.1.2 Install-time Redundancy

The virtual-machine based Java runtime enables all Java programs to "build once, run everywhere". This fact allows Java developers to release a single version of their product for those heterogeneous platforms. Besides bytecode, which is compatible to different platforms, to run a Java software product also requires many other files, including configurations, resource files, and binaries. Developers still need to

create multiple versions of those non-bytecode files to meet the requirements of different platforms. For example, an Android application may contain multiple sets of figures to be compatible with different screen sizes and scales. Another example is that some devices require some additional SDKs which might be unnecessary on other platforms.

Developers cannot foresee which platforms the applications will be installed. However, when an application is installed on a specific platform, all of those files that are created for platform compatible issue will become redundancy immediately. So the install-time redundancy refers to those files can be seen as redundancy only after the installation platform information is given.

### 4.1.2 The Focus of This Chapter

#### 4.1.2.1 Compile-Time Redundancy from Java Libraries

Figure 4.1b illustrates our focus of compile-time redundancy removing in this study. An Android application contains the code written by developers and libraries whose classes are derived from several jar files. By analyzing the application code, we want to distinguish the used library classes from unused library classes and remove those unused ones. In addition, in those used classes, we would like to identify and remove unused methods. Usually, in a real world Android application, all code is in one monolithic Dalvik code file. We need to split it into classes first. Please note that in this study we only remove redundancies from Java libraries. The potential compile-time redundancy in native code and Android framework is out of the scope in this study.

#### 4.1.2.2 Install-time Redundancy from Application Binary Interface and SDKs

Install-time redundancy contains multiple SDKs to support different platforms, multiple sets of embedded Application Binary Interface (ABI) is used to support different CPUs, the components in Android Support Package is designed to support different levels of APIs, different User Interface (UI) layout management, figures with different sizes for being compatible with different screen sizes, as well as many other types of install-time redundancy. In this study, we focus on install-time redundancy caused by embedded ABIs and SDKs.

Table 4.1: Android supported CPU architectures and embedded ABIs

| CPU Architecture | embedded ABI |
|---|---|
| ARMv5 | armeabi |
| ARMv7 | armeabi-v7a |
| x86 | x86 |
| MIPS | mips |
| ARMv8 | arm64-v8a |
| MIPS64 | mips64 |
| x86_64 | x86_64 |

Multiple versions of ABIs contribute to the install-time redundancy. In general, ABI bridges an application code with the operating system on binary level by its definition. In particular, an ABI in an Android application is usually maintained as a shared library (`.so` file) which has been compiled into a specific kind of Instruction Set Architecture (ISA). Not every Android application has an ABI; if an Android application is written in pure Java, its apk file will not have ABIs. However, since many Android applications depend on native libraries, each of those applications should bring in ABIs. Furthermore, considering different Android devices are supported by different CPUs with probably different ISAs, it is recommended to include multiple versions of ABIs in an application's apk file to support the cross-architecture execution. Currently, Android system supports 7 different CPUs. Each type of CPU has its own ABI. Table 4.1 shows all the supported CPUs and their corresponding embedded ABIs by the Android system. We note that once an application is installed, since the architecture (and the ISA) is uniquely determined, except the matched ABI, the rest parts become redundant.

Please notice that it is a recommendation to include multiple ABIs instead of a must. Some applications just included one type of ABI, which usually is armeabi. Such application is still compatible with most Android devices because ARM architecture is backward compatible and most x86 CPUs on Android devices can emulate ARM instructions at the cost of performance. However, to present better experience to the users, many Android applications are likely to include all dedicated ABIs for all the CPU architectures. In Section 4.3, we will evaluate both the proportion of applications that contain install-time redundant ABIs and the impact of removing those redundancies from applications which contain multiple ABIs.

Another type of install-time redundancy comes from multiple SDKs in one Android application. Besides mobile phones, Android applications can also run on other kinds of computing platforms, such as smart watches, televisions, cars, and Internet of Things (IOT) devices. To take advantage of those heterogeneous hardware features, Android provides different set of Software Development Kits (SDKs) for each hardware platform. They are Android API for mobile devices, Android Wear SDK for smart watches, Android TV SDK for televisions, Android Auto SDK for cars, and Android Things for Internet of Things (IoT) devices.

In this chapter, we focus on Android Wear applications to study its install-time redundancy. An Android smart watch cannot connect to the Internet by itself. To connect to the Internet, an Android smart watch should connect to a mobile phone first via Bluetooth, WiFi or a USB cable. Then that mobile phone will send or receive data on the behalf of its paired smart watch. Hence, an Android wear application that involves on-line operations must consist of at least two parts, the mobile phone components and its smart watch counterparts. A typical installation process, in the circumstances that a user has a mobile phone and a smart watch at the same time, will have the following steps.[1] First, a mobile phone will download an apk file to its hard disk. Second, the installer on the mobile phone will install the code running on mobile phones. Third, mobile phone will inject a smaller apk file carried by the original apk, which is usually named "android_wear_micro_apk.apk", to the paired smart watch[2]. However, if a user just has a mobile and does not have a smart watch, which in fact is a more common case, the entire apk will still be downloaded and kept on the mobile phones as a whole, including the code for running on a smart watch.

### 4.1.3 Our Contributions

In summary, we make the following contributions:

- We define and categorize the sources of software bloat in Android applications.
- We propose an automated static approach to identifying and removing those software bloats from Android applications.

[1]Not all Android wear compatible applications use same way to carry smart watch code in the same way, but most applications follow the pattern described here.

[2]Here we describe how applications are installed on standard Android wear 1.x system. The detailed installation process may vary on different customized systems.

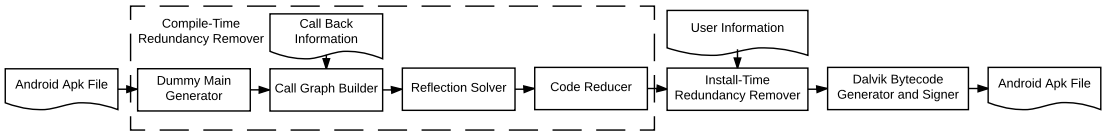



Figure 4.2: REDDROID Architecture

- We have implemented our proposed approach into a prototype called REDDROID. The experimental results we reported not only validate the effectiveness of our approach, but also comprehensively depict the landscape of bloatware issue in the Android application domain for the first time. These results can help developers gain insights about their pain points regarding application resource consumption issue and better plan their optimization in the future.

The remainder of this chapter is organized as follows. Section 4.2 describes the details of the our approach and how we implemented it. We present the evaluation results in Section 4.3. We discuss some interesting thoughts and future works in Section 4.4.

## 4.2 Design and Implementation

### 4.2.1 Architecture

Figure 4.2 illustrates the architecture of REDDROID, which consists of two major components, compile-time redundancy remover and install-time redundancy remover. The tool takes an Android apk file as its input and yields a leaner Android apk file. Compile-time redundancy remover, as shown in the middle part of Figure 4.2, includes several components, which are dummy main generator, call graph builder, reflection solver, and code reducer. The dummy main method generator generates a single entry point for static analysis. Call graph builder statically builds a call graph for the whole Android application. We also use call back information based on Android framework analysis to enhance the results of call graph builder. In addition, the reflection solver helps reinstate some methods which are incorrectly removed due to reflective calls back to the call graph. Based on a more accurate call graph, code reducer will remove the methods and classes not in the call graph. Each component in compile-time redundancy remover responds to a challenging in

Android application static analysis. We will elaborate on each component in the following subsections. Then with user information, install-time redundancy remover will work on the application. We briefly introduce how we build this component at the end of this section. At last, we wrap up the leaner files into a new apk file and sign it. This architecture gives an overall view of our tool in a temporal order. Two removers are not necessarily to execute in one run. There might be a time gap between the running of two removers since installation can happen long after we compile our program.

### 4.2.2 Call Graph

To obtain the information that which classes and methods are used, we build a call graph for the given application. Building an accurate call graph is undecidable, so we over approximate this problem. In other words, in the context of our research, we preserve the soundness of the call graph by ignoring its completeness. Soundness here is defined as all the methods that are not included in a call graph is guaranteed not being invoked. By ignoring completeness we mean some methods that are included in a call graph may also never be invoked. Considering the sizes of some applications are considerable, we do not use some advanced but more expensive call graph building algorithms [69, 75]. In our approach, we use a more intuitive method based on Class Hierarchy Analysis (CHA) [76] to build it.

More specifically, it first establishes class hierarchical information by traversing all the classes. All Java classes and interfaces are inherited from `java.lang.object` Class. So all inheritance relationship will converge into a directed graph.[3] To provide a quick service for the query from next step on if there is a path between two vertexes (if one class is the ancestor of the other one), we in addition compute the transitive closure for all vertex pairs in the graph based on simplified Floyed-Warshall algorithm [77].

Second, we traversed all call sites in an application. During this process, we can obtain the method signature information and the static type of the reference at a call site. But we cannot precisely know what type or subtype of this object can be at static time. Java subtype polymorphism allows runtime to dynamically decide

[3]This directed graph is not a tree (it does resemble a tree though). In Java, a class may implements multiple interfaces. This fact implies there are vertexes having multiple parents in this graph, which contradicts with the definition of a tree.

which version of method to call based on the actual type of an object during run time (a.k.a. dynamic dispatch). We assume that this method can be invoked by the statically-analyzed static type or all subclasses that inherit or overwrite this method. Thus we will add edges from the call site to all versions of this method into the call graph by querying the information generated in the previous step. Our analyzed application starts from the DummyMain. So similarly, all vertexes and edges comprise a directed graph with a root.

### 4.2.3 Android Standard Lifecycle and Dummy Main

A major difference of Android applications, compared with normal Java applications, is that Android applications do not have a main method as its entry point. An android application has multiple entry points. Due to the nature of mobile computing environments and the design of the Android operating system, Android application has a very unique execution model compared with a desktop application with which we are familiar.

An Android application consists of four types of components. They are activities, services, content providers, and broadcast receivers. Each component has the same standard lifecycle. To implement a specific component, a developer must extends a base class of that component and overwrites a set of Android framework callbacks, such as `onCreate`, `onStart`, `onStop`, and `onDestroy`. Then a component can respond to the events of interested, like memory full, the launching of a higher-priority application, or user navigation back to the previous activity. In a sense, the Android framework is "scheduling" on the granularity of components and an application can start from any component which is not disabled by `AndroidManifest.xml`.

To use existing static analysis frameworks and algorithms in analyzing an Android application, we need to generate a dummy main method to model the Android framework invocation behavior and the lifecycle of each component. More specifically, the generated dummy main method will be connected to all possible system callbacks of each component on the call graph. This dummy main method serves as the root of the whole call graph, and our static analysis will start from this dummy entry point.

### 4.2.4 Callbacks

Asynchronous callbacks are implicit control flow transitions, which are widely used to receive and handle User Interface (UI) events in the Android framework. Code listing 4.1 shows an example of asynchronous callback in a simple Android application. At line 8, a button instance in `MainActivity` registers itself to a new `OnClickListener`. Method `setOnClickListener` is a *registration* method. This anonymous class implements the method `onClick` in the original `OnClickListener` interface from line 10 to line 12. The method `onClick` is a *callback*. In this implemented `onClick` method, another method (method implementation is omitted) in the `MainActivity` is invoked (line 11). We note that method `onClick` will not be invoked right after `btnOne` sets its `OnClickListener` (line 8), instead, it will wait until a click event is received, which is the reason we call it an asynchronous callback.

Our previous call graph construction approach (§4.2.2) cannot handle asynchronous callbacks. For example, since `onClick` will be triggered by the Android framework instead of any user-defined method, `onClick` and `anotherMethodInMainActivity` will be reasoned as not being used. Indeed the actual control flow for this example will involve multiple layers of method invocation inside Android framework. Since our customization is essentially focus on application code, one challenge is to capture the implicit control flow transfer between `setOnClickListener` and `onClick` and add additional edge from `setOnClickListener` to `onClick` should be added into the call graph.

To tackle this challenge, we employ a widely-used tool EdgeMiner [55] to analyze a series of Android frameworks. EdgeMiner first identifies a set of methods which are defined in the Android framework and can be overridden in user space. These methods are callback method candidates. Then it iteratively checks each call site of those methods by performing backward analysis. If a call back method candidate $P$ (e.g., `onClick` method) is defined in an Android framework class/interface $C$ (e.g., `Listener` interface), and the type of argument of method $Q$ (e.g., `setListener` method) is $C$, then method $Q$ and $P$ is recognized as a potential registration-callback method pair. The method pairs that satisfy these criteria can overapproximate the real set of actual registration-callback method pairs in the framework.

Then we use the identified registration-callback pairs list to extend our call graph.

Listing 4.1: Callback Example

```
1  public class MainActivity extends AppCompatActivity{
2    private Button btnOne;
3    @Override
4    protected void onCreate(Bundle savedInstanceState) {
5      super.onCreate(savedInstanceState);
6      setContentView(R.layout.activity_main);
7      btnOne = (Button) findViewById(R.id.btnOne);
8      btnOne.setOnClickListener(new OnClickListener() {
9        @Override
10       public void onClick(View v) {
11         anotherMethodInMainActivity();
12       }
13     });
14   }
    ......
   }
```

To this end, for each registration method in the list, we first check if it is in our call graph. If the answer is true, we will analyze all classes that inherit or implement the class or interface in the Android framework to check whether they override the methods mapped with the registration method. In our example, registration method `setOnClickListener` is in our call graph. By checking the list, we found method `setOnClickListener` maps to multiple callbacks. One of these callbacks is method `onClick` declared in the interface `OnClickListener`. By traversing the program we can see that an anonymous class implements the interface and its method `onClick`. So an edge from `setOnClickListener` to `onClick` is added to the call graph. An application may implement multiple versions of callbacks (e.g., multiple versions of `onClick`). Then we will follow the same conservative principle described in the previous subsection. In other words, in the call graph, we will connect the registration method to all possible implementations of a callback based on the class hierarchical information. Then we check the method invocation happened in the method body of newly added callbacks to extend the call graph. If in the callback method body or in the call chain from the callback, we encounter new registration method invocation, then we will recursively repeat this process until a fix point is reached. A fix point is that we do not found any new registration calls in the call chains from the callbacks we discovered in the previous round.

Listing 4.2: Reflection Example

```
1  public void methodExample(String methodName){
2    Class<?> c = Class.forName("com.package.Demo");
3    Object demoInstance = c.newInstance();
4    Method m = c.getDeclaredMethod(methodName,
       new Class<?>[0]);
5    m.invoke(demoInstance);
6  }
```

### 4.2.5 String Analysis and Reflections

Reflection is a dynamic language feature of Java, which allows a Java program to inspect itself and change the behavior during runtime. Investigating reflection invocation targets is one typical challenging task for static program analysis. The call graph construction process (§4.2.2) cannot capture those reflective method invocations, hence some methods might be incorrectly deleted if they are only triggered from call sites of reflections.

Some previous works have proposed several ways to solve reflections, including leveraging annotations from developers and performing test suites. In this research, we tend to use less information from external resources and take advantage of the information carried by the program itself. Hence, we use static analysis to reason the value sets of string variables in the call sites of reflections. Considering Code listing 4.2 which contains two reflection call sites (line 2 and line 5), by statically analyzing potential values of string literals passed to the reflection call sites as parameters, we can reason callees of each reflection call site and use this information to replenish the call graph.

Strings can exist as different forms in a program. For example, string at line 2 in the Code listing 4.2 is a constant literal, and such constant literal is in general easy to handle. On the other hand, reflection call site at line 5 takes a variable of string type as the input, which reveals limited information of potential callees at this call site without further analysis. The major challenges of analyzing string variable are unwrapping loops and solving method invocation contexts. In addition, there exist lots of ways to split, concatenate, and manipulate the values of strings. Precise string analysis requires us to faithfully model those string operation semantics.

Our analysis is based on Violist, a general Java program string static analysis framework [41]. This framework separates representation and interpretation of string

operations, and it provides an IR to represent string values or the string operation data flow relationship. The framework will first perform an intra-procedural analysis to calculate the method summary for each method. Inside a method body, it will first generate the string variable representation for all statements outside loops. Then it treats each nested loop body as a region and uses region-based analysis to generate string variable representations. A string variable in a loop may either depend on the value of a variable, which could be itself, in the previous round of iteration or the same iteration. The framework will not stop its recursively regional analysis until all string variable dependency relationship has reached its fixed point and been reduced to its simplest form. Then the framework will use the method summary of each method to perform inter-procedural analysis to achieve context sensitivity.

Next, interpretation part will parse the results of string variable representation. For example, the constant literals "A" and "B" connected by a plus sign can be represented as (+, "A", "B"). A function of interpretation component is to model the semantics of operations like "+" and output result "AB". We extended the original interpretation part of the framework to support the method signatures and semantics of string operations used in our reflection analysis problem domain.

### 4.2.6 Obfuscation

In the applications we analyzed, there are many obfuscated samples. In this subsection, we present why most obfuscated code will not affect our analysis result. Program obfuscation is an important technique to prevent external users from reverse engineering the software and obtaining the logic, algorithms, or other intellectual properties. It is especially critical to those software that run on virtual machines since it is easier to transform the bytecode back to its source code form. It is notable that program obfuscation is a set of different technologies instead of a single technique. We discuss different cases respectively.

Static obfuscation refers to the obfuscation approaches that do not change program run time behaviors. One of the most widely used static obfuscation approach is symbol name replacement. Though it can dramatically increase the difficulty of understanding the program logic by human beings, its effect is transparent to our analysis. For example, the meaningful symbol names such as `orderName`,

`studentID` in the program will be replaced into `a` and `aa`. This type of obfuscation does not change the call graph. Therefore our analysis results remain correct when we encounter the programs that are statically obfuscated.

Dynamic obfuscation refers to the obfuscation technologies that obfuscate a program by changing its run time behaviors. One major type of dynamic obfuscation is to change the control flow of the software, such as control flow flattening obfuscation. Changing the control flow of a software can hide the original logic of the code. However, this approach just changes the control flow graph inside a procedure. It doesn't change the shape of the call graph. Thus the validity of our analysis results will not be compromised. In recent years, some more advanced dynamic obfuscation approaches are proposed such as virtualization based obfuscation [78]. However, researchers have found several approaches to automatically deobfuscating those virtualization-obfuscated software [79]. By applying those deobfuscation processes before our analysis, we can overcome the difficulty imposed by virtualization obfuscation.

### 4.2.7 Install Time Redundancy Removal

Works of removing install time redundancy cannot be done by removing resources alone. It also requires dependency relationship analysis. Figure 4.3 shows the build process of an Android application. The parts in the dotted line box present how resources in an Android project are processed and linked to other components. Android asset packing tool (aapt) plays a major role in resource compilation. First, except the files in the asset directory and the res/raw directory which will be packaged into the apk file in their original formats, all other resources will be compiled. For example, all XML files will be compiled into binary XML files. Second, except the resources in the asset directory, all other resources will be assigned an ID. Third, it will generate a R.java file which will be compiled with other Java source code by a Java compiler later, and a resources.arsc file which will be put long with all compiled resources. The resource.arsc and complied resources will be packed into a zip file which uses ap_ as its suffix. Android system has its own resource reference mechanism. All resources, are not referred in the program by their paths directly. Instead, they are referred by their IDs. More specifically, during runtime, when a resource is referred, the system first validates the value of

ID by checking R.java. Then it uses resource.arsc as a symbol table to translate the ID into a specific filename. Then AssetManager will use it to open a file. Therefore, conceptually, besides removing redundant resources, the generated symbols and the dependency upon those symbols across all resource files also need to be removed. We elaborate the details of the implementation of this part in the end of this section.

### 4.2.8 Sign the Customized Application

An Android application must be signed to run on Android systems. The Android application sign process includes two steps. First, a message digest is generated for each file in the apk file of an application. Second, the developers or some other people on behalf of the developers use the private key to sign the message digest of every file in the application. If a program has already been signed before it is customized, then the program needs to be signed again to be runnable since RedDroid will modify files in the apk of an application.

### 4.2.9 Implementation

We have implemented our approach in a prototype called RedDroid. RedDroid is mostly written in Java and Unix shell scripts. It includes a compile-time redundancy remover written in Java and a installation-time redundancy remover written in Unix shell scripts. Regarding compile-time redundancy remover part, we rely on FlowDroid [56] to generate dummy main method for analyzed Android applications. We use Soot [63] to convert Dalvik bytecode into the Soot IR Jimple. Our analysis and code modification is based on Jimple. We use Apktool to reverse resource files in an apk file from binary format back to human readable ASCII format.

Android application install-time redundancy remover consists of two components, Android wear application redundancy remover and redundant embedded ABIs remover. We use Unix shell scripts to implement Android wear application install-time redundancy remover. It first calls apktool to unzip the analyzed apk file and decode resource files into its original form. Then it removes android_wear_micro_apk.apk from directory res/raw and android_wear_micro_apk.xml from directory res/xml. We then search all build files to identify and remove the build targets which rely on android_wear_micro_apk.apk and android_wear_micro_apk.xml. In addition,

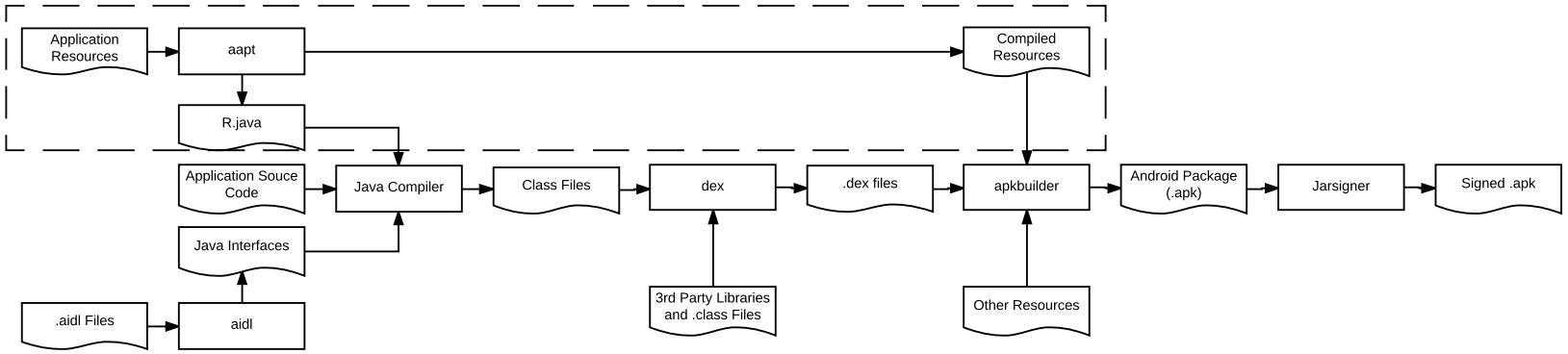



Figure 4.3: Android Application Build Process

we search all resource files that referred to those two files. After these three steps, we use apktool to rebuild the whole project into a new apk file. The redundant Android embedded ABIs remover is implemented in a similar approach. It accepts an ABI name which we want to preserve as its argument. After unzipping the analyzed apk file and decoding the resource files by calling apktool, our tool will search the subdirectories under the lib directory. All subdirectories except the one we want to preserve will be deleted. We then rebuild the whole package into a new apk file and sign it.

## 4.3 Evaluation

In this section, we evaluate RedDroid on Android applications downloaded from Google Play. Our experiments were conducted on a server with an 32-core Intel Xeon CPU E5-2690 @ 2.90GHz processor and 128G Memory. The operating system is Ubuntu 12.04.5 LTS. The Linux kernel version is 3.8.0-30-generic. We use Android API level 14 as our Android application running environment.

To evaluate RedDroid, we want to answer the following research questions.

**Q1:** What is the impact of our compile-time redundancy trimming technique on the size of Android applications?

**Q2:** What is the impact of our compile-time redundancy trimming technique on the code complexity of Android applications?

**Q3:** How many and what types of reflection calls are used by Android applications?

**Q4:** What is the impact of our install-time redundancy trimming technique on Android wear applications?

**Q5:** What is the proportion of applications that include multiple sets of embedded ABIs in their .apk files?

**Q6:** What is the impact of our install-time redundancy trimming technique on Android applications that have redundant embedded ABIs?

### 4.3.1 Code Size

In this section, we present experiments to answer the research question Q1, the impact of trimming off compile-time redundancy from Android applications. We first show the data distribution on all of our 553 Android application samples. Then we present some data from some selected applications to give a glimpse of the details of our results.

#### 4.3.1.1 Results of Tested Android Applications

We first report the overall results of the tested Android applications. We apply RedDroid towards 553 Android applications to remove their unused methods and classes. By dividing the application original size by its size after customization, we get the percentage of the remaining size of an lean apk file. Figure 4.4 presents all 553 data points we yielded. The vertical axis is the percentage of a lean application size. The horizontal axis is the original size of an application. The maximum reduced-original ratio is close to 100% and we report the minimum reduced-original ratio is 43.11%. On average, the reduced-original jar size ratio is 85.59%. The median percentage is 86.44%.

#### 4.3.1.2 Detailed Data of Selected Android Applications

We randomly selected 10 Android applications from our data samples to demonstrate some detailed results. The experimental results are shown in Table 4.2. Among 10 benchmark programs, the lowest reduced-original apk size ratio is 64.97% which is from OpenTable (line 7 in the table). Evernote Widget (line 3) has the highest reduced-original apk size ratio which is 94.17%. Please note that, these percentage numbers present the overall impact on an application apk file as a whole. Besides bytecode, an application also has many other files, including native libraries and resource files. If the proportion of resource file size among overall size is small, then the trimming on the bytecode part is more likely to have bigger impact. In next subsection, our evaluation focuses on the sole bytecode part.

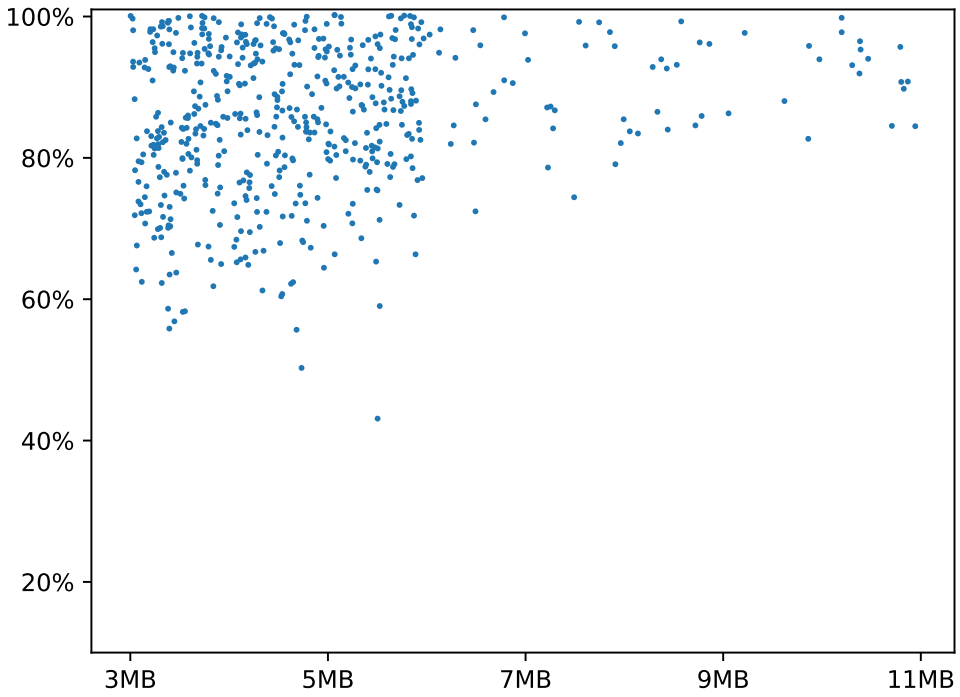



Figure 4.4: Reduced Size Distributions

Table 4.2: 10 Selected Android Application Code Size Before and After Unused Code Trimming Comparison

| Benchmark | Original (Byte) | Reduced (Byte) | Reduced/Original (%) |
|---|---|---|---|
| IFTTT | 7416304 | 6026077 | 81.25 |
| Evernote Widget | 1201311 | 1131289 | 94.17 |
| Motorola Migrate | 4542461 | 3260831 | 71.79 |
| Baidu Browser | 5009942 | 4103846 | 81.91 |
| Yahoo Messenger | 3865985 | 3279559 | 84.83 |
| OpenTable | 3919118 | 2546431 | 64.97 |
| Flashlight | 4767339 | 4091859 | 85.83 |
| Marvel Comics | 5503831 | 4475991 | 81.33 |
| Papa Johns | 5206893 | 3753546 | 72.09 |
| Instagram | 10365650 | 9406437 | 90.75 |

### 4.3.2 Code Complexity

In this subsection, we present the experimental results to answer the research question Q2: the impact of RedDroid on the code complexity of Android applications by removing compile-time redundancy. By using Chidamber and Kemerer object-oriented metrics (CK metrics) and two other software engineering metrics, we can exclude the factors from other parts of an application and dedicatedly evaluate the impact of our approach on the bytecode part of an application.

CK metrics is a set of metrics to measure Object-Oriented(OO) software complexity, which is proposed by Chidamber and Kemerer [71, 80]. We use the following measurements from CK metrics, Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Coupling Between Objects (CBO), Response For a Class (RFC), and Lack of Cohesion in Methods (LCOM). All of these metrics are calculated based on a single class. We sum up the results of all classes of an application to profile the bytecode complexity of an application as a whole.

Each measurement depicts different aspects of bytecode. WMC is the sum of weight of each method. In our evaluation, the weight of all methods is 1. So the number of WMC equals to the total number of methods in an Android application. The number of DIT is the levels from given class to `java.lang.Object` which is the root in the Java inheritance tree. A deeper inheritance tree sometimes can help developers to better model problems and design solutions. However, it may also involve more complexities into code base and runtime. CBO counts the number of classes that are "coupled" to a given class. We define that if class A calls the methods or accesses the variables of class B, then class A is coupled to class B and class B is coupled to class A. A high CBO indicates that the software design violates many OO principles, which can cause many problems. First, modifying the implementation of one high CBO class will risk affecting many other parts of the software. In addition, it will make a software less modular and harder to be reused. At last, it is difficult to test a class with high CBO independently [81]. RFC is the number of *response set* of a class. Response set, according to the paper of Chidamber and Kemerer [71], is "a set of methods that can potentially be executed in response to a message received by an object of that class". A class with higher RFC tends to have higher complexity. If a great number of methods are involved in responding a message, then the developers need to understand more pieces of code

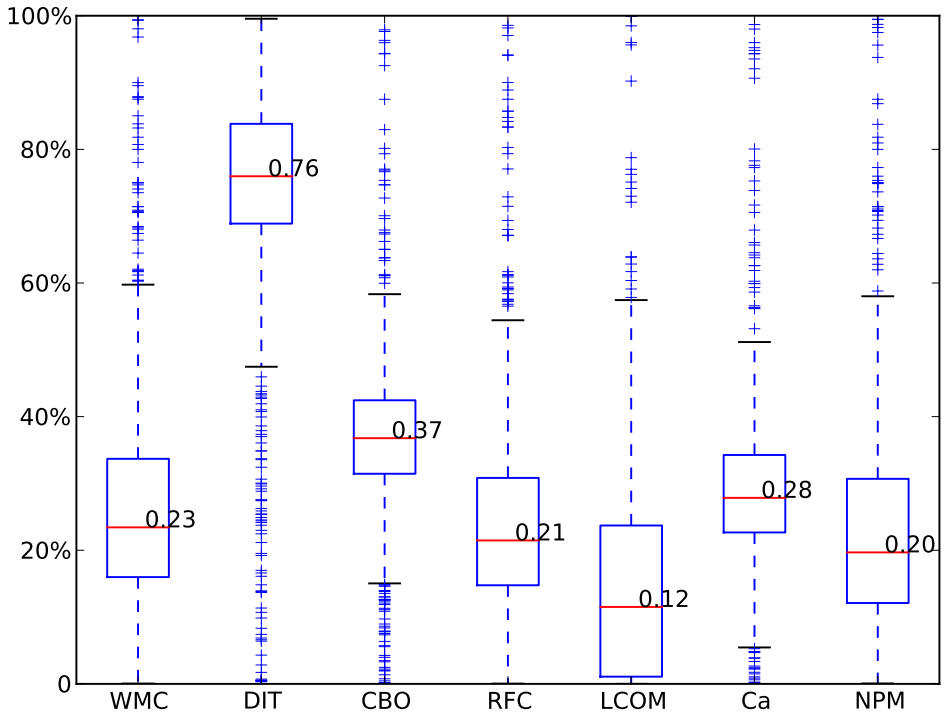



Figure 4.5: Code Complexity Results

to construct the event handling logic, which raises more challenges in development and test. LCOM is calculated based on the following steps. Every pair of methods in a given class is checked. If a pair of methods both access to at least one the same reference or variable, then the number of LCOM minus 1. If a pair of methods does not share any reference or variable, then the number of LCOM plus 1. A high LCOM implies some code in a class should be moved out. The other two metrics are Number of Public Methods (NPM) and Afferent Coupling (Ca). Ca counts how many other classes refer to the class we are measuring.

Figure 4.5 shows the results of code complexity evaluation. We use the data of each metric we collected in the lean version of an application to divide the data from the original version of an application. We collected code complexity data from the 553 Android application data samples. The vertical axis is the reduced-original ratio. The horizontal axis lists every metric. For the 553 data points of every metric, we use a boxplot to depict the distribution of the results. The position of

Table 4.3: Reflection Call Sites

| Method Name | Call Sites | Constants | Variables |
|---|---|---|---|
| java.lang.Class: java.lang.Class forName | 8.579 | 3.779 | 4.800 |
| java.lang.ClassLoader: java.lang.Class loadClass | 2.611 | 1,846 | 0.765 |
| java.lang.Class: java.lang.reflect.Field getField | 2.168 | 1.786 | 0.382 |
| java.lang.Class: java.lang.reflect.Field getDeclaredField | 1.077 | 0.828 | 0.249 |
| dalvik.system.DexClassLoader: java.lang.Class loadClass | 0.302 | 0.035 | 0.267 |
| java.util.concurrent.atomic.AtomicIntegerFieldUpdater: java.util.concurrent.atomic.AtomicIntegerFieldUpdater newUpdater | 0.042 | 0.042 | 0 |
| java.util.concurrent.atomic.AtomicLongFieldUpdater: java.util.concurrent.atomic.AtomicLongFieldUpdater newUpdater | 0.014 | 0.014 | 0 |
| net.sourceforge.pmd.typeresolution.PMDASMClassLoader: java.lang.Class loadClass | 0.014 | 0 | 0.014 |
| org.codehaus.jackson.mrbean.AbstractTypeMaterializer$MyClassLoader: java.lang.Class defineClass | 0.007 | 0 | 0.007 |
| org.codehaus.jackson.mrbean.AbstractTypeMaterializer$MyClassLoader: java.lang.Class findLoadedClass | 0.007 | 0 | 0.007 |
| java.lang.ClassLoader: java.lang.Class findClass | 0.004 | 0 | 0.004 |
| Total | 14.825 | 8.330 | 6.495 |

top and bottom of a box represent third (Q3) and first (Q1) quartiles of a group of data. Interquartile Range (IQR) is defined as $Q3 - Q1$. The highest bar and lowest bar indicates the maximum value and minimum value respectively. The maximum value and minimum value are defined as $Q3 + 1.5IQR$ and $Q1 - 1.5IQR$ in a boxplot. The data out of maximum value and minimum value is seen as "outliers" in a boxplot. The position of red line indicates the median of the data.

### 4.3.3 Reflection Call Sites

In this section, we present the results to answer research question Q3. Table 4.3 presents the results of our reflection analysis. We inspect all Java and Android reflective methods in the 553 Android application samples. We listed the name of methods which are used at least once by those applications in column "method name". A method name we listed consists of three parts. From left to right, they are the class to which the method belongs, the type of return value, and method name. We merged the data of overloaded methods into one entry of the table, so we did not list the parameters of each method. In the second column, we listed the average number of call sites of each method in the Android applications that used reflections from higher frequency to lower frequency. In addition, we also calculate the average number that how many string parameters at reflection call sites are string literal constants or variables. We report them in the third and fourth column. In the last row of Table 4.3, we can see that, each Android application has 14.825 reflection call sites. Among this number, 8.330 call sites directly use a constant literal as their string parameters, while the other 6.495 call sites use a variable as their string parameters. Though Java language provides many reflection methods, in real programs, the distribution of their usage is quite biased. Top 4 entries in the table have more than 97% of all reflective call sites.

Table 4.4: Size and Percentage of Installation Redundency in Wear Applications

| Application Name | Original Size (Byte) | Reduced Size(Byte) | Reduced/Original (%) |
|---|---|---|---|
| Mobills: Budget Planner | 15725488 | 14023717 | 89.18 |
| AccuWeather | 33442700 | 30437691 | 91.01 |
| Wear Tip Calculator | 4070793 | 2011661 | 49.42 |
| App in the Air: Flight Tracker | 14826753 | 10847048 | 73.16 |
| Weather Live Free | 32659948 | 31086561 | 95.18 |
| ViewRanger - Trails and Maps | 14537273 | 12374449 | 85.12 |
| Keeper: Free Password Manager | 15905430 | 11531725 | 72.50 |
| Instant - Quantified Self | 10038765 | 9544534 | 95.08 |
| Google Play Music | 17961307 | 15678474 | 87.29 |
| Microsoft Outlook | 30348958 | 26821576 | 88.38 |
| Nest | 41391891 | 35063279 | 84.71 |
| Robinhood - Free Stock Trading | 13858114 | 9777723 | 70.56 |
| Strava Running and Cycling GPS | 28164055 | 24944648 | 88.57 |
| Viber Messenger | 31555265 | 30305049 | 96.04 |
| Wear Face Collection | 27476581 | 17154939 | 62.43 |
| Komoot Cycling and Hiking Maps | 12150268 | 9573000 | 78.79 |
| WatchMaker Premium Watch Face | 13864136 | 8879229 | 64.04 |
| Average | N/A | N/A | 80.67 |

### 4.3.4 Installation Time Redundancy

#### 4.3.4.1 Install-Time Redundancy from Android Wear Applications

In this section, we answer the research question Q4. We did experiments to compare the reduced size and original size of Android wear applications. At this moment, the number of applications that support Android watch is still limited. We downloaded all applications in the category of "Android Wear". We analyzed those Applications and identified 17 applications that explicitly contain an "android_wear_micro_apk.apk" in their apk files. We applied our tool to all of those 17 applications. Table 4.4 presents our experimental results. Among all 17 applications, the lowest reduced-original ratio is 49.42% which is from Wear Tip Calculator. The application Weather Live Free has the highest reduced-original rate, 95.18%. On average, after removing android install-time SDK redundancy, the size of a customized application will be 80.67% of its original size.

#### 4.3.4.2 Install-Time Redundancy from Android Application embedded ABIs

In this paragraph, we answer the research questions Q5 and Q6. We did experiments to compare the reduced sizes and original sizes of Android applications that

Table 4.5: Proportions of applications that contain redundant ABIs by different size groups

| Size | All Analyzed Apps | Modified Apps | Modified/All (%) |
|---|---|---|---|
| 1M | 717 | 411 | 61.51 |
| 2M | 416 | 244 | 58.65 |
| 3M | 303 | 168 | 55.45 |
| 5M | 495 | 209 | 42.22 |
| 10M | 908 | 445 | 49.01 |
| 20M | 1029 | 391 | 38.00 |
| 30M | 240 | 13 | 5.42 |
| 50M | 604 | 148 | 24.50 |
| 200M | 67 | 12 | 17.91 |
| Total | 4779 | 2041 | 42.71 |

contain redundant ABI. ARM architectures dominate mobile devices. Among ARM architectures, ARMv7 is most pervasive. So in our evaluation settings, we always try to keep ARMv7 ABI if multiple ABIs are present. If ARMv7 ABI does not exist, we turn to keep ARMv5 while we are deleting the rest.

Table 4.5 presents the proportions of applications that contain redundant ABIs by different size groups. In total, we analyzed 4779 Android applications, among which 2041 applications contain more than one type of ABIs. That is to say 42.71% applications in our samples can be additionally customized. We also calculated the data by application size groups. For example, the applications that are in 3M group are larger than 2M and less than or equal to 3M. The applications that are smaller than 1M has the highest proportion of application containing redundant ABIs, which is 61.51%. We can also observe a trend that larger applications have less redundant ABIs.

Figure 4.6 shows the size distribution of all 2041 applications that can be customized. The vertical axis is the percentage of the customized size divided by the original size. The horizontal axis is the original size of an application. On average, after customization, the reduced size is 93.37% of the original size.

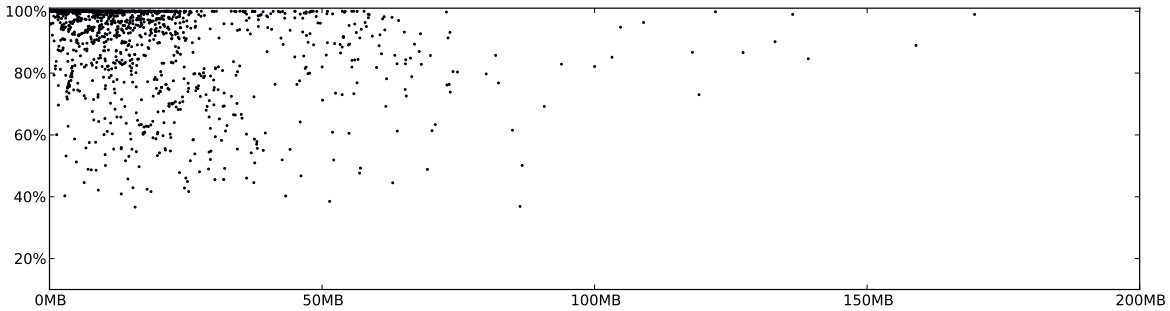



Figure 4.6: ABI details

## 4.4 Discussion and Future Work

### 4.4.1 Install-time Redundancy to Support legacy APIs

Android systems have different levels of APIs, from level 1 which is the oldest one to 25 which is the most recent one. Those APIs are not always backward compatible. Some features are only available on some higher level APIs. Compared with iOS, Android ecosystem is more fragmented. Android users are using many different Android systems which support different levels of API. To bring a unified experience to all users, developers can include some packages provided by Google in the apk file. Those packages provide the implementation of some features that originally only available in some versions of Android systems. If an application is installed on a new version of Android system, those packages will be redundant. Including these packages are not transparent to developers. For example, class `android.app.Fragment` is only available to API levels higher than 22. If developers decide they also want to support those old systems, they must explicitly use `android.support.v4.app.Fragment` which is located in the package can be brought by the application itself, instead of `android.app.Fragment` which is only in Android framework. To optimize this case, we not only need to remove the package, but also need to rewrite the class declarations and package importing in application code. We leave this part as one further work.

### 4.4.2 Feature based Customization

Another future work is to perform feature based customization towards an application. Jiang et al. [82, 83] discuss an approach of feature-based customization over a Java program. The approach is based on analyzing the call sites of some framework APIs. The permissions in Android systems which map to some specific Android framework APIs provide an ideal handler to conduct feature optimization on Android applications. For example, by customizing application features, we can abandon existing "all-or-nothing" permission protocols with users. It is possible for users to only select part of the permissions they allowed and still enjoy (part of) the applications. It is also useful to enforce some policies for some special groups like minors, military personnel, and the employees working in the enterprise where some features (e.g., video streaming) are disallowed.

### 4.4.3 Relationship with Other Android Application Compaction Approaches

There are a plenty of Android application code size reducing approach based on packing and compressing. More specifically, developers can compress the images music, and animation with or without losing their original resolutions. They can also transform all the string literals in .csv or .plist files into binary-based representation. We note that these approaches are orthogonal to our technique. RedDroid can be boosted with other code size reducing approach mentioned above towards Android applications. Our analysis is conducted on off-the-shelf Android applications, and it is reasonable to assume they should have already been optimized by existing trimming approaches. In other words, our evaluation results indicate that our work is still notably effect given the presence of other related techniques.

### 4.4.4 The Relationship Between Our Approach and Dead Code Elimination

To this end, we have used two chapters to discuss how we remove unused code and resources from Java applications, JRE, and Android applications. Unused code removing, by its literals, resemble another well known term, dead code elimination. Dead code elimination is a technique to remove the code that has no affect on

program outputs [84]. By the definition, the approach proposed by us is also a kind of dead code elimination techniques. However, it has a different scope with the well known dead code elimination techniques used in modern compilers.

Modern compilers use a set of methodologies to identify and remove the dead code inside a procedure. Constant folding and constant propagation can replace those variables whose values are never used in their life cycles into constants. Unreachable code can also be identified by a compiler by analyzing the control flow in a function. For example, if an if-else structure contains an always true or always false branch, then that if-else structure can be optimized. The other example is the statements that are after return statements and also cannot be jumped into from other parts of the function. Another type of dead code eliminated by a compiler is the variables that are initialized or written but never read by the program.

Our techniques focus on the whole program analysis and interprocedural analysis. Rather than removing the unused statements or variables in the scope of a function, we remove the unused classes and unused methods from the entire program perspective.

### 4.4.5 The Universality of Our Unused Code Removing Implementation and Approach

The evaluation regarding unused code removal in the previous two chapters is conducted on some language-specific benchmarks. However, our implementation and approach are not language-specific. This issue can be discussed from the perspectives of compilation targets and programming paradigms respectively.

Regarding compilation targets, source code can be compiled into bytecode or binary. First, our implementation can be applied to almost all bytecode. For example, Java Virtual Machine (JVM) can run a considerable number of JVM-compatible languages including Scala, Clojure, Groovy, JRuby, and Jython. Though the source code of those languages are different, after compilation, their bytecode formats are identical. Accordingly, our tools can be used to analyze the applications written in JVM-compatible and Dalvik Virtual Machine (DVM) compatible languages seamlessly. For other types of bytecode, because of the nature of bytecode virtual machine interpretation-based execution and no linkage process during compilation, bytecode instructions usually carry enough type information and do not replace

symbols by constants. Accordingly, in most cases, we can faithfully transform them into other types of IR. Therefore, by properly implementing a language transformation front end, our implementation can also be directly used. Second, our general approach still remains valid for binary. This is because the mechanisms of variable access, class initialization, method invocation, and other control flow transfer are still the same. However, there are additional challenges caused by binary code. A major one is that, due to the linkage process, many symbols are replaced into constants, which caused the unreallocatable code program. It is still an open problem in binary reverse engineering area. But we have seen some promising progresses in solving this problem [85]. By taking advantage of their works to overcome the unreallocatable code issue, our approach is also compatible to binary code.

In terms of programming paradigms, there are OO languages and non-OO languages. OO languages are the major sources of bloatware and exactly the major target of our research. In our research, many efforts are spent on solving some OO-specific challenges including inheritance and polymorphism. For non-OO languages, call graph building approach needs to be adjusted accordingly, but each step of our general framework is still useful. Intuitively, the programming paradigms such as imperative languages or functional languages are not likely to import much unused code into their applications. But it is still interesting to see how great our approach works on those types of languages, which can be investigated in the future.

### 4.4.6 Security Impacts

The works of removing redundancy from software have multiple security impacts. First, less code means less vulnerabilities. According to an estimate made by McConnell [86], there are about 10–20 defects every thousand lines of code (KLOC) during the in-house testing stage. In the final release version, there is about 1 defect per KLOC.[4] Assume the bugs are distributed randomly. The number of the bugs we can be trimmed is positively proportional to the percentage of the code size we can reduce. In the evaluation of the previous chapter, we have confirmed that, on average, nearly half of the known vulnerabilities in JRE can be trimmed. It is notable that our method also trims *unknown* vulnerabilities.

[4]Note this is a rather conservative estimate. For example, Mockus, Fielding, and Herbsleb [87] found that the Apache Server has about 2.64 defects per KLOC.

Second, less code indicates smaller attack surface in general. Property-Oriented Programming (POP attack) is an code reuse attack that takes advantage of available classes and methods (not necessarily used ones) in software written in Java, PHP, and many other bytecode-based languages [88]. As we know, another well-known code reusing attack, Return-Oriented Programming (ROP) is not available to those bytecode-based languages, since the call stacks of bytecode-based languages is spread over real stacks, heaps, and data segments. An overflow cannot take control over multiple places. In addition, the bytecode is at unknown positions of heaps. POP attack walks around those challenges to reuse code in the software by exploiting deserialization functions in languages like Java and PHP. More specifically, by rewriting the serialized objects files, attackers can replace the properties, especially those object references, in serialized objects to redirect the control flow to the points they wish. The key of this attack is to identify and chain some usable classes, instead of "gadgets" in ROP, in all available classes in the software to reach the operations like files deleting or dynamic code execution. Therefore, removing unused classes and methods reduces the attack surface and greatly suffocates the possibility that an attacker is able to create an object chain in the software.

Third, removing unused code can increase software diversity, which also improves security. For example, customized JRE can only run the applications for which it customized. It is a desirable feature for security. More importantly, we will have different versions of JRE to run different applications. It is hard for attackers to use a single attack script to compromise all those different JREs, which can prevent some massive security incidents.

Another security impact is that using the approach of removing unused code to reduce the code size brings in little new security loopholes, compared with other code size reduction approaches. For example, some research on JavaScript code size reduction focuses on code compression, which may give attackers chances to obfuscate their malicious JavaScript [89]. Trimming redundant code can help reduce JavaScript code size without security concerns.

Last but not least, removing unused code can reduce code complexity, which potentially makes other analyses possible. "Complexity is the enemy of security" [90]. But in real world, the pace of software complexity increasing does not slow down, which causes that conducting the whole system analysis and optimization is expensive. The cost of formal methods like model checking is also sensitive to the code

Table 4.6: reflection patterns and our strategies

| Reflection Pattern | Strategy |
|---|---|
| unknownClass.knownMethod | Keep all methods that have the name "knownMethodName" in any class. (Sound) |
| unknownClass.knownField | Keep all fields that have name the name "knownField" in any class. (Sound) |
| knownClass.unknownMethod (no such case in our data samples) | Do not change this class. (Sound) |
| knownClass.unknownField | Do not change this class. (Sound) |
| unknownClass.unknownField | Delete methods only, do not delete classes. (Sound) |
| unknownClass.unknownMethod (no such case in our data samples) | Send alert to developers. |
| unknownClass.<constructor> (no such case in our data samples) | Send alert to developers. |

complexity. Less complex code gives space to those additional measurements to improve security of software.

### 4.4.7 Soundness of Static Reflection Resolving Method

Soundness is important to program transformation. Theoretically, reflections cannot be statically decided. However, the usage patterns of reflections in real world applications allow us to walk around many challenges caused by reflections. Let's review Table 4.3 again. From the table we can see that, 553 applications only use 11 reflection methods in total. None of the return types of these methods is java.lang.reflect.method, which means no methods are invoked in a reflective manner in our data samples. We can see that developers only use reflections to get a Class, a Classloader, a Field, or a Fieldupdater. Table 4.6 lists our strategies to each refection pattern. If a class, a method, or a field is referred statically, or referred by reflection calls but can be determined by analysis, then we label it as "known". Otherwise, we label it "unknown". In total, there are 7 different combinations listed in the table. First 5 cases can be processed soundly. Last 2 cases are unsound. However, we have never seen last 2 cases in our data samples. Therefore, our approach can be applied to almost all applications in the market. More importantly, by using static analysis, we can know when our analysis will be unsound. Compared with developer annotations or the dynamic approach, this is a great improvement, since for other reflection solving approaches, if there is a flaw in annotations or test suites, we cannot know it until the transformed program throws an error. The data we collected regarding the reflection usage and static approach on reflection call resolving is promising. It will be interesting to conduct a dedicated reflection study on real world programs to additionally investigate the static approach.

# Chapter 5
# Feature-based Software Customization: Preliminary Analysis, Formalization, and Methods

## 5.1 Introduction

A typical modern software system delivers a set of features to its users in a bundled way. The requirements of removing and customizing one or some of those bundled features are raised from both developers and users, for both software engineering reasons and software security reasons.

### 5.1.1 Software Engineering Pragmatic Issues

Rinard [91] lists several reasons causing functionality bloat in modern software. Based on his work, we summarize some software engineering pragmatic issues related to feature creep problem. First, feature creep happens in most software development projects. When a software product becomes mature and stable, the developers still update it by adding more functions into original design. Second, software is designed and delivered as general purpose software which contains all functions required by all potential users. However, for a specific user, his or her requirements on the software are special. Only a small part of the functions of the software are useful. Other functions, to this specific user, become redundant features. Third, software reusing also is an important source of functionality bloat. The design of libraries tends to be generalized. The design of legacy projects is

specialized for the purpose of legacy requirements. None of them are built for current projects and requirements. Building applications upon them inevitably brings redundant features into the software. Fourth, development "errors" also import new features. Developers are not always aware of the all effects might caused by the code they are writing. Holzmann [6] calls this phenomenon "dark code" which means "the application somehow can do things nobody programed it to do".

#### 5.1.1.1 Why Customizing a Feature is Difficult?

If a property, feature, component is well abstracted, then it is easy to be changed, extended, or removed. The challenges of features removal actually are caused by the challenges of features abstraction. Modern programing languages (e.g., OO languages), code organization (e.g., package domain, name space) and other software engineering toolkit give developers a way to model a real-world work flow and split them into smaller and smaller units. However, a system could be modeled into different abstractions and concerns. Programmers can only design the software according to the *primary abstractions*. The secondary and third important abstractions might be cross cut with the primary abstractions. Kiczales et al. [92] discuss several cross-cutting features in typical real world applications. For example, to design an online banking system, the concepts and entities that are primarily abstracted would be "balance", "account", "user", and etc. Extending and changing those entities are relatively easy because of well abstraction and encapsulation.

Feature "transaction integrity" also needs to be abstracted and implemented in an online banking system. However, it is not a primary abstraction of an online banking system. Code listing 5.1 shows an example. In that code snippet, feature "transaction integrity" which is enforced by logging cross cuts with money transfer and any other transaction business logic. If developers want to change or remove the implementation of money transfer business logic, they just need to change the business logic code inside method moneyTransfer. However, if developers want to enhance the "transaction integrity" by changing logging policy. They have no way to change the code in one place. They have to change all methods that the transaction integrity cross cuts with such as "moneyTransfer", "directDeposit", "checkDeposit", and many others. Similar examples could be found in the way how people implement network connection and database connection. Code listing 5.2

Listing 5.1: A simplified example showing how the transaction integrity feature cross cuts with the moneyTransfer business logic

```
public void moneyTransfer(int amount, User sender,
        User receiver){
  logger.info("transaction␣starts");
  //do money transfer buisness logic;
  logger.info("money␣was␣deducted␣from␣sender's␣balance");
  //do money transfer buisness logic;
  logger.info("money␣was␣added␣to␣receiver's␣balance");
  logger.info("transaction␣completes");}
```

Listing 5.2: A simplified example showing how the network connection feature cross cuts with the ingestContent business logic

```
public void ingestContent() throws Exception {
  URL oracle = new URL("http://www.example.com/");
  URLConnection yc = oracle.openConnection();
  BufferedReader in = new BufferedReader(
  new InputStreamReader(yc.getInputStream()));
  String inputLine;
  while ((inputLine = in.readLine()) != null){
    //do business logic;
  }
  in.close();}
```

shows how network connection feature cross cuts with other business logic. Code listing 5.3 shows a similar example on database connection feature cross cutting with other business logic. These examples just demonstrate some common cases shared by many projects. In each specific project, it has more specialized features that are tangled with other code.

Correctly customizing a well modularized component in a program is already a challenge. From the examples above we can see that the pervasive features that cross cut with other business logic additionally increase the difficulty of customizing. That probably is one of the reasons that developers keep them there even after recognizing the negative effects that might be caused by some redundant features.

### 5.1.2 Security Concerns

We also have strong motivations to customize some features from the software for the security reasons. Redundant features play a role in at least three security threat models.

First, malicious software vendors might threat users' privacy. There are many

Listing 5.3: A simplified example showing how the database connection feature cross cuts with the userAuthentication business logic

```
public void userAuthentication(){
  Class.forName("org.postgresql.Driver");
  Connection connection = null;
  connection = DriverManager.getConnection(
        "jdbc:oracle:thin:@localhost:1521:fakename",
                "username","password");
  //do business logic;
  connection.close();}
```

cases that the software companies insert backdoors to collect users' or their competitor software's behavior. If network connection feature is not used by users at all, then the users can just trim the network connection feature or at least writing-to-network feature off from the software. For example, many text editors have network connection feature. Trimming off such a feature will not affect the functions of the text editors. If the users of software hold highly sensitive data or they work in a settings with some hard constraints (e.g., military offices), they should remove those features that are not useful to users' business but with sensitive behaviors.

Second, malicious libraries provider might repackage original authenticated libraries to insert code for their own interests which threat the integrity of software that includes such a library. For example, many mobile application developers include adware library (adware here refers to the software that allows third party distributes and displays advertisements on your own apps, webpages, or software) to earn extra revenue. Some adwares might secretly collect both developers' data and users' information. Besides existing scanning and malicious behaviors detecting technologies, developers still have requirements and motivation to customize third party libraries they include in their applications to achieve active defense.

Third, the software systems that lack diversity might be compromised all together at one time by outside attackers. The approach of moving target defense (MTD) is raised to "increase uncertainty and apparent complexity for attackers" [93]. Feature-based software customization according to users' requirements offers a natural way to increase software diversity and achieve moving target defense.

### 5.1.3 Our Approach

In this dissertation, we propose a novel approach to conducting feature-based program customization via multiple-step static analysis. One of the steps of our approach is based on an enhanced program slicing method called *solo slicing*. Based on a set of seed methods defining a feature, our approach investigates its call sites, return value, and parameters. Then starting from the return values and parameters at call sites, we remove any code that depends on return values, and any code that is *only* depended by the parameters. Next, we remove call site itself. At last, if possible, we remove method definition by checking a set of rules.

More specifically, we use return values in the seed method call sites as our forward slicing criteria to find out all statements that depend on the return values on both data flow and control flow. By removing these statements, we guarantee the program is still runnable after seed method call sites removal. We use parameters in the seed method call sites as our solo slicing criteria to find out all statements that are only depended by the parameters on data flow or control flow. By removing these statements, we trim off the redundancy caused by the absence of the seed methods call sites.

We evaluate our methodology by conducting case studies on several real-world Java applications. We aim to remove the network connection feature, database connection feature, and logging feature from those applications respectively. The results of the case studies show that our approach is correct and effective. In summary, we make the following contributions:

- We define the feature and the problem of features-based customization.
- We purpose a multiple-step static analysis which is based on enhanced program slicing technology to perform feature-based customization.
- We identify several features that are prone to be interwoven with other code and contains security-sensitive behavior.
- We conduct a case study to evaluate the feasibility, correctness, and effectiveness of our approach.

The rest of this chapter is organized as follows. We define and formalize the research problem of feature-based software customization in section 5.2. We present

the general approach in Section 5.3. The evaluation and case study are reported in Section 5.4. Discussion is presented in Section 5.5.

## 5.2 Problem Definition

Before we discuss the approach to conducting feature-based program customization, we first define, formalize, and set the scope for the research problem in this section. Based on the previous analysis in last section, we have found that many features cross cut with other business logic. So feature customization cannot be done by modifying one or several methods' definition. Features are implemented as many spread and repeated method calls. Still taking code listing 5.3 as an example, database connection feature in a program is implemented by all invocations of method DriverManager.getConnection(). So we use methods call sites as feature definition basis.

We are going to formally define feature based on interprocedural control flow graph (ICFG) $G = (V, E)$ which regards method invocation as a special kind of control flow [94]. Besides the normal control flow edges $E_n$, three special kinds of control flow edges are imported to handle procedure invocation process: ***call-to-return-site*** edge $E_{\text{call-to-return-site}}$, ***call-to-start*** edge $E_{\text{call-to-start}}$, and ***exit-to-return*** edge $E_{\text{call-to-return}}$. ***Call-to-return*** edge connects the call site node and the node following the call site. ***Call-to-start*** edge connects the node that invokes the method and the entry node of the callee. ***Exit-to-return*** connects the exit node (usually return node) to the node immediate after the call site. Formally, all-edges set E is the unions of those five subsets of edges, i.e., $E = E_n \cup E_{\text{call-to-return}} \cup E_{\text{call-to-start}} \cup E_{\text{exit-to-return-site}} \cup E_{\text{entry}} \cup E_{\text{exit}}$.

**Definition 3.1.** We define ***seeds*** as a set of methods denoted by $SEEDS = \{m_1, m_2, m_3, ..., m_k\}$. Seeds could be the methods in Java standard libraries or part of the application which are specified by users or developers. Seeds usually are the methods conducting sensitive operations or other functions of interests. Simply, we call a set of methods of interest "seed methods".

**Definition 3.2.** We define ***call sites*** of a method $f$ a set of node on this graph G, $C_m = \{c_1^m, c_2^m, \ldots, c_n^m\}$, such that $\forall c_i^m \in C^m, \exists e_i \in E_{\text{call-to-start}}$ connects $c_i^m$ and the entry node of method $m$. Simply, we call the all statements that invoke a method the "call sites" of that method.

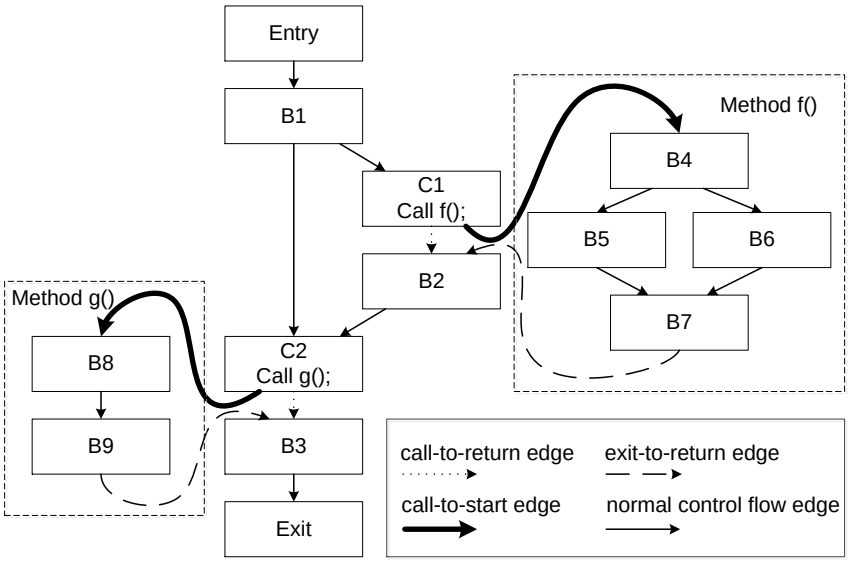



Figure 5.1: Interprocedural Control Flow Graph of Code Listing 5.4

**Definition 3.3.** We define a ***feature*** a set of call sites $F = \{C_{m_1}, C_{m_2}, C_{m_2}, ..., C_{m_n}\}$ such that $\forall C_{m_i} \in F, m_i \in SEEDS$, Simply, a feature consists of all call sites of *SEEDS*.

**Example.** In our example program snippet shown in code listing 5.4, if we specify method $f$ and $g$ as the seed methods, then $SEEDS = \{f, g\}$. The call site of method $f$ is $C_1$. The call site of method $g$ is $C_2$. In this case, the feature is a set consisting of two call sites: $\{C_1, C_2\}$. Taking a network feature for example, the seed set is defined to be the set of network-related APIs and the feature set is the set of call sites to these APIs. In some scenarios, we need to customize a software package to completely remove network features. We will present an approach and conduct case studies for these scenarios in the following sections. Customization contains operation of add, remove, and modify. In this chapter, we discuss an approach of feature removing.

## 5.3 Approach

### 5.3.1 Overview

Based on the definition given by last section, we can clearly define our task is to remove all call sites of the seed methods *safely* and clear all the redundancy caused

Listing 5.4: A code listing example

```
public class CodeListing {
  public static void main(String[] args) {
    int argument=Integer.parseInt(args[0]);
    int ret=0;
    CodeListing instance=new CodeListing();
    if(argument<=42){
          ret=instance.f(argument);
    }
      instance.g();
  }
  public int f(int x){
    int y;
    if(x>0){
      y=1;
    }else{
      y=-1;
    }
    return y;
  }
  public void g(){
    System.out.println("Hello");
    System.out.println("World");
  }
}
```

by this removal. To remove all methods invocation in the program, we potentially need to remove 4 parts of code in order: the code that depends on the return value or side effects (objects and array references that redefined in the callee) of the call site, call site itself, the code that is *only* depended by the parameters of the call site, and the method definition. To better demonstrate our idea, we will use a Java program example shown in code listing 5.5 through the whole section to show how our solution deletes those tangled code step by step. The Java code in listing 5.5 shows an simplified SMTP client that interacts with network. We omit the exception handling and invalid data checking in this example. This program opens a network connection after necessary preparation. It writes string *message* to the network. It reads data from the network and stores the data into array *b* via calling method read of class DataInputStream which is in JRE. We show a simplified implementation of this method in code listing 5.6 to ease the elaboration later. In the end, the program prints the actual length of the data that is read from the network, the first byte of the message, and the value of offset.

Our goal is to customize this program into a data-reading-free program which only writes data to but never receives any data from the network. The seed method in our example is DataInputStream.read(byte[] b, int off, int len). The only call

Listing 5.5: An example that a client reads data from and writes data to the network

```
1  public class SocketInAndOut{
2    public static void main(String[] args) {
3      Socket smtpSocket = null;
4      DataOutputStream os = null;
5      DataInputStream is = null;
6      String message_body=args[1];
7      String message="message_example";
8      int offset=0;
9      byte[] b=new byte[100];
10      smtpSocket = new Socket("hostname", 25);
11     os = new DataOutputStream
        (smtpSocket.getOutputStream());
12     is = new DataInputStream
        (smtpSocket.getInputStream());
13     int array_length=b.length;
14     os.writeBytes(message);
15     String responseLine;
16     int actual_length=is.read(b, offset, array_length);
17     if(actual_length<array_length){
19       System.out.println(actual_length);
19     }
20     System.out.println(b[0]);
21     System.out.println(offset);
22     os.close();
23     is.close();
24     smtpSocket.close();
25   }
31}
```

site in our case is the statement in line 16 of listing 5.5. Figure 5.2 highlights this call site and denotes 4 customization steps. In the first step, all code that depends on the return value `actual_length` and the value of the cells of array $b$ which is changed in the callee would be removed. In the second step, call site itself would be removed. In the third step, we are going to remove the code that *only* affects the parameters of the call site. In the last step, we are going to check if it is possible to delete the method definition of DataInputStream.read(byte[] b, int off, int len). It is not always the case that we need to perform all four steps every time. In our example, the deleting in step 4 will not happen because the seed method we choose is part of JRE. In other cases, we may skip step 2 or step 3 if the method does not have return value and side effects or the number of its parameters is 0. We unified all 4 steps of program customization as Program Dependency Graph (PDG) and System Dependency Graph (SDG) updating and reachability problems solving process. We need to build call graph and SDG as analysis preparation procedure. The details of each step are given by the following subsections respectively.

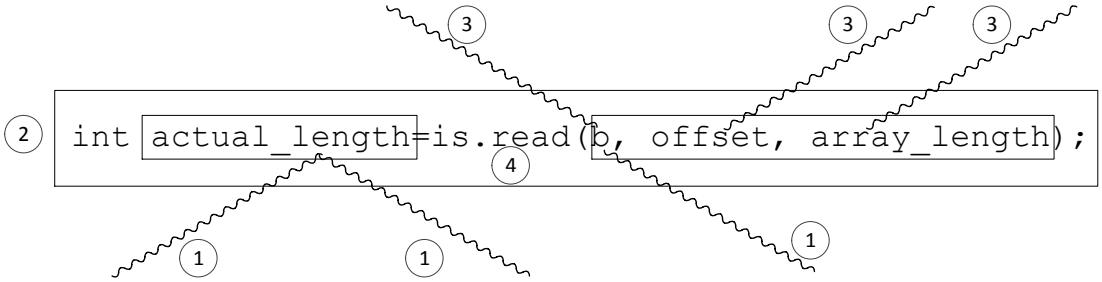



Figure 5.2: Delete Overview

Listing 5.6: A simplified java.io.DataInputStream.read implementation

```
1 public final int read(byte b[], int offset, int len){
2   int c = read();
3   b[offset] = (byte)c;
4   int i=1;
5   for(;i<len;i++){
6     c=read();
7     if(c==-1){
8       break;
9     }
10    b[offset+i]=(byte);
11  }
12  return i;
13}
```

### 5.3.2 First Step: Forward Slicing

Before we start to describe the details of the first step, I would like to clarify the terms we use here. The concept slicing is first formalized by Weiser [48] by given a slicing criterion. Some following up works give their own definitions of slicing [95, 96]. Those definitions resemble each other while there still are subtle difference between them. In this chapter, by "forward slicing" we mean finding out the statements that are affected by the variable $v$. Based on the SDG and PDG built by the preprocessing procedures, we now present the approach of forward slicing to identify the variables that rely on the return value and the side effect of the call site. We use the two-pass SDG-searching algorithm introduced by Horwitz, Reps, and Binkley [97].

The first step is to identify which variables in the program are slicing criteria. The selected variables should be the ones that are defined or redefined in the method and caused the effects out of the scope of that method. Apparently, the return value of the method is such a variable. Besides return value, a method may have

other side effects. The side effects are caused by the change to the value pointed by array or object reference or the change to the array or object reference itself. In SDG, both explicit return value and implicit side effects data flow could be handled uniformly. We show a simplified DataInputStream.read(byte b[], int off, int len) implementation in code listing 5.6 which is called in line 16 of code listing 5.5. Method *read* has one return value and one side effect. The return value is that it returns an integer to the variable *actual_length* in line 16 of code listing 5.5. The side effect is caused by the change to the value of the cells of array $b$ in code listing 5.6. The cells of array $b$ is used as right value after the call site of *read*. We present a partial SDG in Figure 5.3 to show how *read* method interprocedurally depends with the code in the listing 5.5. In Figure 5.3, the nodes in grey are special nodes in SDG. They map the data flow between actual parameters and formal parameters. In the figure we can see that the return value and side effect are modeled in the same way. Thus, from all actual out nodes of the call sites, a graph reachability analysis would be performed to slice out all statements that depend on the "output" of the call sites. The sliced out statements would be deleted. In our example of code listing 5.5, line 17 depends on *actual_length* via data flow. Line 18 depends on *actual_legnth* via control flow. Line 20 depends on the value stored in array $b$ via data flow. However, line 21 would not be removed. Method *read* does not mutate the value of offset, so the *offset* in line 21 does not depend on the call site of *read* in line 16. In summary, line 17 to line 20 would be deleted. After this step, there are no statements depending on the seed methods.

### 5.3.3 Second and Third Step: Call Site Delete and Solo-slicing

In this step, our target is to remove the call sites and the statements that *dedicatedly* produce value for the actual parameters of seed methods call sites. Program slicing cannot help us solve this problem. According to the definition given by Weiser [48], program slicing could be denoted by a slicing criterion. Formally, it is a tuple defined as $C =< i, V >$, where i is a statement of program $P$ and $V$ is a subset of variables of program $P$. Slicing technology helps identify the statements that may affect $V$ in $i$ via data flow or control flow. Theoretically, the slicing result is still an executable program. However, our research question in this step could be abstracted as, which statements *only* affect $V$. This question could also be asked in

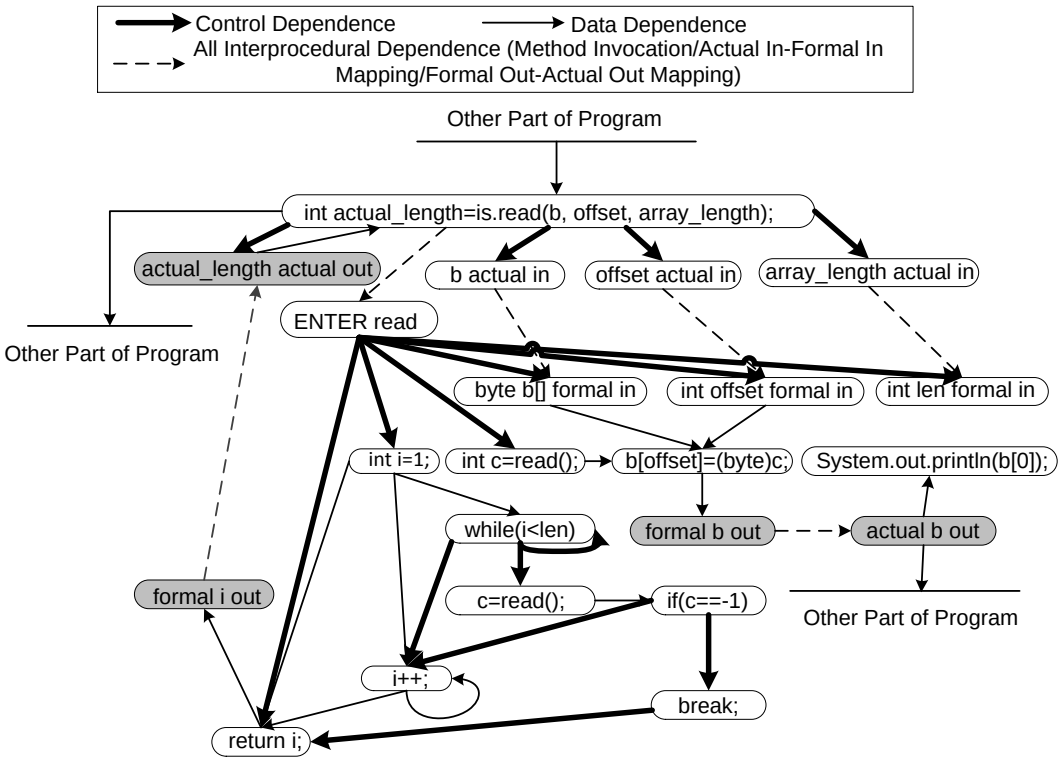



Figure 5.3: Forward Slicing on Return Value and Side Effect

the other way equivalently: after removal of the call sites, which statements before call sites could be removed safely. Here we do not require the sliced-out result still a runnable program. But we require the program left is still runnable.

We still use code listing 5.5 as our example. If we set the backward slicing criteria as the variable *offset* call site of *read* at line 15, then the statement in line 8 would be sliced out. However, we cannot delete the statement in line 8 because line 21 still depends on it. Figure 5.4 highlights the relationship between these statements. This example demonstrates why slicing cannot solve this program. Some other various versions of slicing technologies such as thin slicing [98] improved the slicing results based on an evolved slicing definition. They also cannot solve the problem we raised here.

To this end, we find out that traditional slicing and its existing variation versions cannot solve the problem we encounter in this step. we define an enhanced program slicing methodology called solo-slicing to solve our problem. Like other slicing technology, we define solo-slicing and develop our algorithm based on Program Dependency Graph (PDG) and the System Dependency Graph (SDG).

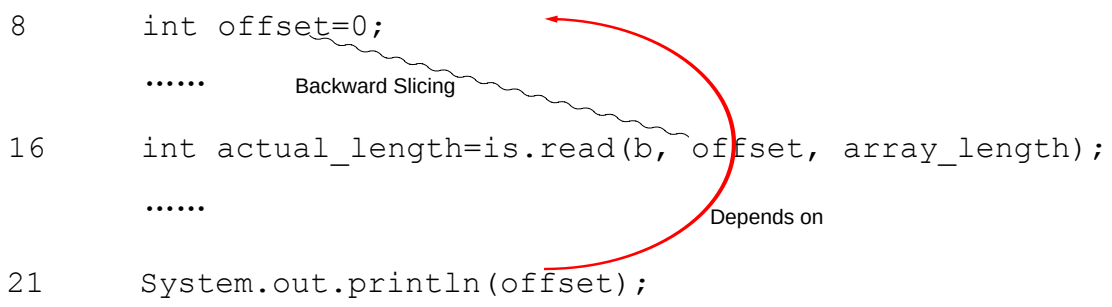



Figure 5.4: Traditional Slicing Fails to Identify the Redundancy Caused by Call Site Removal

#### 5.3.3.1 Program Dependence Graphs and Solo-slicing

PDG [50, 51, 97, 99] represent the data flow dependencies and control dependencies of a program. It is notable that, by the definition of PDG [51], the term "program" only contains scalar variables, assignment statements, conditional statements, and loops. It does not contain procedure calling statement. A PDG could be denoted by a directed graph $G_p = \{V, E\}$ where V are the vertices and E are the edges in PDG [51]. Vertices represent all statements in program $P$, including both assignment statements and control predicates. Edges represent transitive data flow dependencies and control dependencies. The vertex pointed by the directed edge depends on the source of the edge, either on data flow or control flow. Two statements (vertices) are connected by a data flow dependencies edge if 1) one statement defines a value $x$, 2) the other one statement uses that value, and 3) there is a def-use chain across at least one control flow.

The edges denoting the control dependencies are labeled with either true or false. The statement will be executed when the result of control predicate matches the label of the edge that connects that statement and the control predicate. Code listing 5.7 shows a really simple code example. The PDG of this example is shown in Figure 5.5. In the Figure 5.5, the arrows in bold with label represent control dependencies. The rest of the arrows represent the data dependencies.

Our algorithm is based on worklist and graph search which is shown in algorithm listing 1. The main idea is to delete one vertex first from $G_p$ and the edges pointed to it. The first step has removed the statements depending on the return value and side effects of seed method call sites. After removing call site vertex, we

Listing 5.7: A Simple Program Without Method Call

```
int session_pool=100;
int user_number=57;
if(session_pool>=user_number)
  resources=session_pool-user_number;
}
int revenue=user_number*10;
```

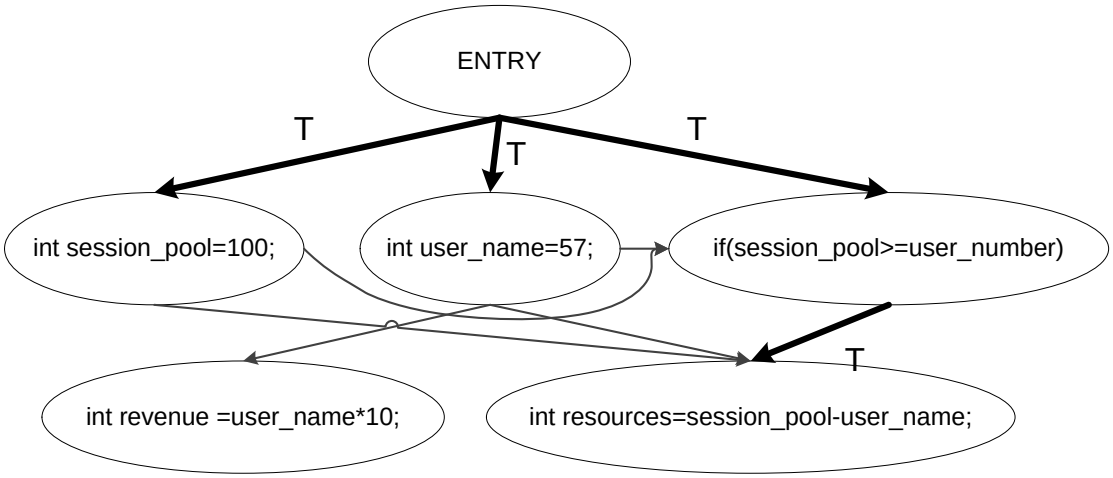



Figure 5.5: The PDG of the Example Code in Listing 5.7

update the SDG and check the out degree of vertices in updated $G_p$. Specifically, if there are vertices whose out degree are zero, we can remove those vertices and the in-degree edges pointing to those vertices. We repeat this process until there are no vertices having zero out degree. When no vertices have zero out degree, we say the fixed point is reached and the algorithm stops. By this algorithm, we can calculate the solo-slicing result and delete the solo-sliced-out statements along side. Figure 5.6 demonstrates two examples of solo-slicing. The number on each vertex denotes the out degree of that vertex. The vertex that has zero out degree is in red. Two examples have the same vertices number and the different initial dependency relationship settings. The figure shows how they reach the fixed point via different number of steps.

Back to the example in code listing 5.5, call site of *read* in line 16 is the slicing criterion of solo-slicing which will be removed first as the trigger of the graph updating. It has three parameters, array *b*, integer *offest*, and integer *array_length*. Among them, array *b* depends on line 9 array *b* is declared. Variable *array_length* depends on line 13 and line 9 via data flow. No other statements depend line 9 and

**Algorithm 1** Solo-slicing Algorithm

```
 1: function SoloSlicing(G,S)
 2:     PDG ← G
 3:     CallSiteVerticesSet ← S
 4:     WorkList ← CallSiteVerticesSet
 5:     while WorkList ≠ ∅ do
 6:         Vertex ← WorkList.getOneVertex()
 7:         PDG.removeGivenVertex(Vertex)
 8:         PDG.updateVerticesOutDegree()
 9:         for each vertex v changes its out degree do
10:             if v.OutDegree = 0 then
11:                 WorkList.add(v)
12:             end if
13:         end for
14:     end while
15: end function
```

line 13. Thus these two statements are removed. Variable *offset* depends on line 8. However, variable *offset* in line 21 also depends on line 8. According to solo-slicing algorithm, line 8 would not be removed. In summary, in step 2 and 3, call site of *read* in line 16, redundancy in line 9 and line 13 are identified and removed.

#### 5.3.3.2 System Dependence Graphs and Solo-slicing

To handle the program in real world with procedure invocation, we need to extend PDG into System Dependence Graphs (SDG) by adding new types of vertices and edges to represent method calling. Here we use the notation of SDG introduced by Horwitz et al. [51]. we extend PDG to SDG by adding five kinds of vertices. They are call site vertex, actual-in vertex, actual-out vertex, formal-in vertex, and formal-out vertex. The pair of actual-in/out vertices model the process of passing the actual arguments to and read return value from the callee. The pair of formal-in/out vertices model the process of formal parameters initialization and the return value at the return site in the callee. Then, we add three kinds of edges to link these newly added vertices. The first one is a call edge, which connects call site and the entry vertex of callee. The second one is a linkage-entry edge, which connects actual-in vertices in caller and their corresponding formal-in vertices int the callee. The third one is a linkage-exit edge which connects formal-out vertices in the callee and their corresponding actual-out vertices in the caller. Figure 5.3

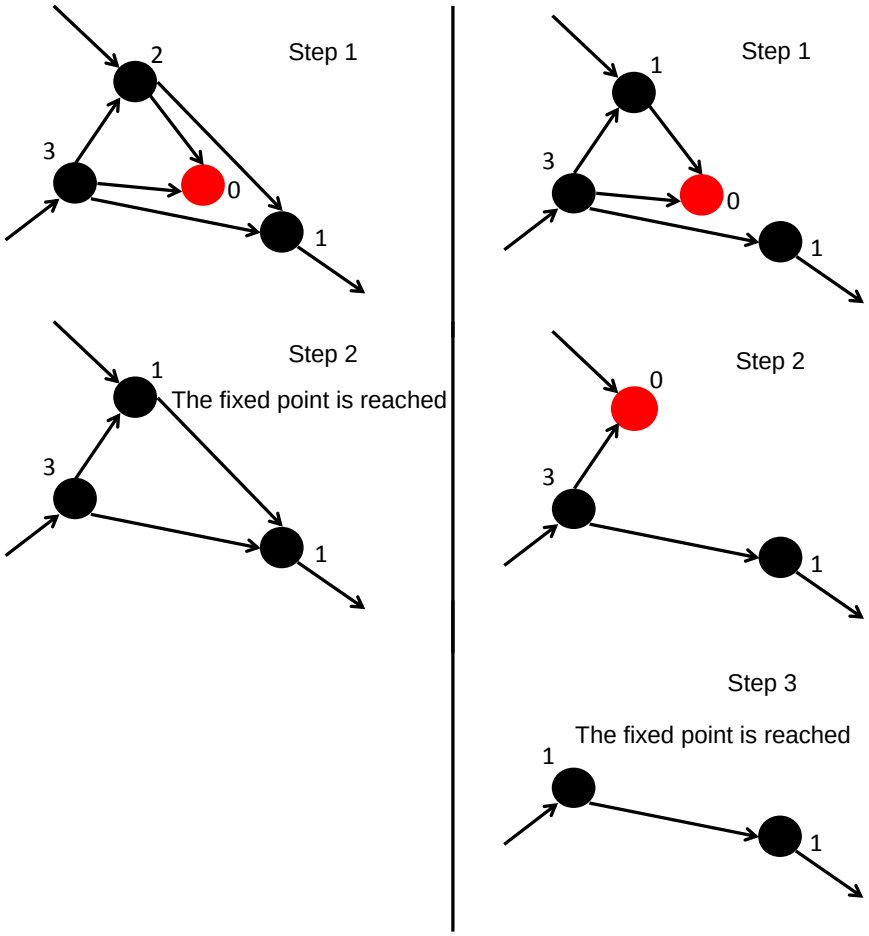



Figure 5.6: Solo-slicing Algorithm Illustration

already shows an example of a partial SDG in the previous subsection. According to the definition of the newly added vertices and edges in SDG, call edge is a special kind of control dependency edge. Linkage-entry edge and linkage-exit edge are special kind of data dependency edges. Thus, the algorithm in listing 1 could easily be extended to work on SDG. Consequently, solo-slicing could solve the problem on a program with method calling.

#### 5.3.3.3 Extending Solo-slicing Algorithm to OO Program

Similarly, by appropriately adding new types of vertices and edges, the dependency relationship in an OO program could be represented by a graph data structure. Larsen and Harrold [100] did this work and call such a data structure *Class Dependency Graph* (ClDG). Traditional slicing algorithm could work on OO program

Listing 5.8: After Customization

```
1  public class SocketInAndOut{
2    public static void main(String[] args) {
3      Socket smtpSocket = null;
4      DataOutputStream os = null;
5      DataInputStream is = null;
6      String message_body=args[1];
7      String message="message_example";
8      int offset=0;
10      smtpSocket = new Socket("hostname", 25);
11     os = new DataOutputStream
        (smtpSocket.getOutputStream());
12     is = new DataInputStream
        (smtpSocket.getInputStream());
14     os.writeBytes(message);
15     String responseLine;
21     System.out.println(offset);
22     os.close();
23     is.close();
24     smtpSocket.close();
25   }
31}
```

by utilizing ClDG. Likewise, we take advantage of ClDG to extend the working scope of solo-slicing to include OO program. Specifically, each *method entry* vertex is connected with the *class entry* vertex via *class member* edge. To represent an inheritance relationship, the derived class reuses the the PDGs of all methods that are inherited from its parent classes. To represent the polymorphism, a *polymorphic choice* vertex is added. It points to all possible callee of a call site. Essentially, ClDG is still a graph representing data flow and control flow dependency. So our algorithm could work on ClDG and solo-slicing could be extended to the OO program.

### 5.3.4 Fourth Step: Method Definition Delete

After all the processes above, in this step, we check if it is possible to remove the seed methods definition. If the a seed method resides in an application class or a third-party library class, then we remove it. If this method is in Java Runtime Environment (JRE), then we do not remove it.

After all four steps, the result of our customization on code listing 5.5 is shown in code listing 5.8. After customization, this code snippet is runnable, does not contain the *read* feature and does not have redundancy code caused by the feature customization.

## 5.4 Evaluation and Case Studies

### 5.4.1 The Complexity of Our Approach

In our four-step feature-based software customization approach, step 1 traditional forward slicing and step 3 backward solo-slicing cost most. The cost of traditional slicing has been given by Horwitz et al. [51]. It is bounded by $O(P(V + E) + CX)$, where $P$ is the number of procedures in the system, $V$ represents the biggest number statements in a single procedure, $E$ is the biggest number of edges in a single $PDG$, $C$ is the total number of the call sites in the system, and $X$ is the sum of the biggest number of formal parameters in any procedure plus the number of global variables in the system. The cost of solo-slicing could be estimated in the same way. Compared with traditional slicing, in each iteration of worklist, there are three operations. The first one is removing current vertex which is a constant time in a single iteration and linear to the size of worklist in the whole run. The second one and third one are updating affected vertices' out degree and checking if any one of them become zero. The number of affected vertices equal to the reachable vertices in traditional slicing. The number of vertices that are newly added to the worklist in each iteration of solo-slicing is smaller or equal to the vertices that are added to the worklist in each iteration of traditional slicing, when given the same initial settings. So it is easy to see the cost of solo-slicing is also bounded by $O(P(V + E) + CX)$. Thus the complexity of the whole 4-step approach is bounded by $O(P(V + E) + CX)$ which is efficient.

### 5.4.2 The Pervasiveness of Cross Cutting Features in Real World Java Program

In this subsection, we present the results to the research question "how pervasive the cross cutting features are in the real world Java program". We select three features. They are network connection, database connection, and logging.

We conduct experiments on DaCapo 9.12-bach benchmarks, which contain 10 programs. These 10 benchmarks are typical desktop standalone applications that are designed by following the principle of "small is beautiful". For most of them, network connection and data persistence are not their proposed functions. If one

Table 5.1: Network, Database, and Logging Features

| **Benchmarks** | avrora | batik | fop | h2 | jython | lucene | pmd | sunflow | tomcat | xalan |
|---|---|---|---|---|---|---|---|---|---|---|
| **Number of Network Feature Call Sites** | 0 | 83 | 28 | 1 | 1 | 8 | 3 | 0 | N/A | 9 |
| **Number of Database Feature Call Sites** | 0 | 0 | 0 | N/A | 2 | 6 | 3 | 0 | 8 | 1 |
| **Number of Logging Feature Call Sites** | 0 | 0 | 87 | 1 | 5 | 962 | 0 | 0 | 195 | 0 |

benchmark's main function happens to be network connection and data persistence, we are going to skip that application.

#### 5.4.2.1 Presence of Network Connection Call Sites

The number of network connection call sites of each benchmark is shown in the first row of Table 5.1. Among the benchmarks, tomcat is a webserver whose main business logic includes network connection. In this case, we do not consider network connection feature as a cross cutting feature for benchmark tomcat. So we do not calculate this number for the benchmark tomcat. From the table, we can see that 7 out of 9 benchmarks have cross cutting network connection API call sites. The benchmark batik has the highest call sites number 83. The benchmark avrora and sunflow do not network connection call sites. On average, each benchmark has 15 network connection call sites.

#### 5.4.2.2 Presence of Database Connection

The number of database connection call sites of each benchmark is shown in the second row of Table 5.1. Among the benchmarks, h2 itself is a database. So we do not calculate database connection call sites number for benchmark h2. From the table, we can see that 5 out of 9 benchmarks have cross cutting database connection API call sites. The benchmark tomcat has the highest call sites number 8. On average, each benchmark has 2 database connection call sites.

#### 5.4.2.3 Presence of Logging

The number of logging call sites of each benchmark is shown in the third row of Table 5.1. From the table, we can see that 5 of out of 10 benchmarks have cross cutting logging API call sites. The benchmark lucene has the highest call sites number 962. On average, each benchmark has 125 logging call sites.

Table 5.2: Call Sites of method *openConnection* and *openStream* in DrJava

| **SEEDS: java.net.URL.openConnection/openStream** |
|---|
| edu.rice.cs.drjava.ui.NewVersionPopup$6.updateAction |
| edu.rice.cs.drjava.ui.NewVersionPopup.getManualDownloadURL |
| edu.rice.cs.drjava.ui.MainFrame._generateJavaAPISet |
| edu.rice.cs.drjava.ui.NewVersionPopup.getBuildTime |

### 5.4.3 Case Studies

#### 5.4.3.1 DrJava: Network Connection

DrJava is a lightweight Java programming environment for pedagogic purpose [101]. The core functionality of DrJava has nothing to do with network connection. However, it does have network connections in its code for checking updates. DrJava has 687 classes. The total number of lines of code are 163,566. We conduct a case study on removing network based feature from Dr.Java. Network related features are defined by methods *openConnection* and *openStream* in class *java.net.URL*. Specifically, if developers want to have network connection in their program, they must call those APIs. Thus, in this case, *SEEDS* consists of methods *openConnection* and *openStream*. Table 5.2 shows the specific call sites of the seed methods. We use them as seed methods to conduct feature-based customization based on the approach we proposed.

Among these call sites, we use method *updateAction* in class *NewVersionPopup* as an example. Code listing 5.9 shows the forward slicing results of *openConnection* inside method *updateAction*. The call site is at line 321. The return value of this call site is *uc* whose type is *URLConnection*. This call site does not have side effect. So the only slicing criteria is *uc* in the statement of line 321. The statements shown in code listing 5.9 would be deleted. The method *openConnection* does not have parameters. So the backward solo-slicing would not be performed. At the end, we find out that the method *openConnection* is in JRE. So the method definition will not be removed. After customization, we test Dr.Java by the test cases designed by us. First, DrJava could start up successfully after feature customization. Second, DrJava cannot access Internet to check and download update. Third, rest of functions could work normally.

Listing 5.9: DrJava openConnection Callsite Forward Slicing Results

```
321  URLConnection uc = fileURL.openConnection();
322  final int length = uc.getContentLength();
323  InputStream in = uc.getInputStream();
324  ProgressMonitorInputStream pin =
       new ProgressMonitorInputStream
      (_mainFrame, "Downloading␣"+fileName+"␣...", in);
325  ProgressMonitor pm = pin.getProgressMonitor();
326  pm.setMaximum(length);
327  pm.setMillisToDecideToPopup(0);
328  pm.setMillisToPopup(0);
330  { public void run() { closeAction(); } });
331  BufferedInputStream bin = new BufferedInputStream(pin);
334  edu.rice.cs.plt.io.IOUtil.copyInputStream(bin,bout);
335  bin.close();
337  if ((!destFile.exists())
       || (destFile.length() != length)) {
338    abortUpdate("Could␣not␣download␣update."); return;
339  }
```

#### 5.4.3.2 Hadoop: Database Connection

Apache Hadoop is an open source software for scalable distributive computing. In this case study, we want to remove the database connection feature from Apache Hadoop project. Database connection related features are defined by method *getConnection* in class *java.sql.DriverManager*. Specifically, to connect with database, developers must write a sequence of routine code to perform a series of operations which starts with *DriverManager.getConnection*. So in this case, *SEEDS* contains only one method *getConnection*. It has two call sites. Both of them are in method *getConnection* of class *DBConfiguration*. Code listing 5.10 shows two call sites of our seed method in *DBConfiguration.getConnection*. It is notable and interesting that *DBConfiguration.getConnection* actually is a wrapper method of Java standard API *DriverManager.getConnection*. They even have the same method name. *DBConfiguration.getConnection* directly uses the return value of seed method as its own return value (line 151 and line 153). This fact causes that all call sites of *DBConfiguration.getConnection* are also removed from the program in the forward slicing stage.

The backward solo-slicing starts from the parameters of *DriverManger.getConnection*. We use the call site at line 151 as an example. That seed method call site uses the *anonymous* return value of *conf.get(DBConfiguration.URL_PROPERTY))* as its parameter. Apparently, an anonymous return value is impossible to be used somewhere else but its call site. So the call site *conf.get* is also removed. After

Listing 5.10: The Code of Call Sites of *DriverManager.getConnection*

```
150  if(conf.get(DBConfiguration.USERNAME_PROPERTY) == null) {
151    return DriverManager.getConnection(
         conf.get(DBConfiguration.URL_PROPERTY));
152  } else {
153    return DriverManager.getConnection(
         conf.get(DBConfiguration.URL_PROPERTY),
         conf.get(DBConfiguration.USERNAME_PROPERTY),
         conf.get(DBConfiguration.PASSWORD_PROPERTY));
155  }
```

this removal, the number of statements that depend on constant value *DBConfiguration.URL_PROPERTY* is one less. But the SDG shows that its out degree is still greater than zero at this moment. So the solo-slicing stops here and the static constant field *DBConfiguration.URL_PROPERTY* will not be removed. The backward solo-slicingfrom parameters of call site in line 153 follows the same manner. But the results are different. The static constant fields *DBConfiguration.USERNAME_PROPERTY* and *DBConfiguration.PASSWORD_PROPERTY* are removed in the end because line 153 is the *only* statements that depend on these two fields. At last step, we check if it is possible to delete the method definition. Our seed method is part of JRE. So the seed method definition will not be deleted. But its wrapper method *DBConfiguration.getConnection*, whose call sites are removed as well, is in application space. So the method definition of *DBConfiguration.getConnection* will be deleted.

After customization, we test *Hadoop-mapreduce-client* by the test cases designed by us. We find out that the database connection is disabled. The project's rest of functions could work normally.

#### 5.4.3.3 Maven: Logging

Apache Maven is a software project management tool written in Java which can facilitate building automation, documents generation, and dependency resolving. In this case study, we want to remove debugging information logging feature from Maven. In Java, there are multiple logging frameworks. But their design are quite similar. By calling different methods of logger, logger can log the information and label these information with different importance-level tags. Thus administrators can handle or retrieve these logs according to their importance levels for different purposes. Maven uses the logging framework from *Plexus* project.

In this framework, the logging importance levels, from least important to most important, are ranked as *LEVEL_DEBUG, LEVEL_INFO, LEVEL_WARN, LEVEL_ERROR,* and *LEVEL_FATAL*. To log debug information, developers need to call *Logger.debug* method. So *SEEDS* in this case study, contains one method *org.codehaus.plexus.logging.Logger.debug*. By removing this seed method, we can remove debug information logging feature.

The seed method has 60 call sites in 17 classes as shown in Table 5.3. Some package names are omitted due to its length exceeding the page limit. We use one of these call sites in class *DefaultProjectDependenciesResolver* as an example. The relevant code is highlighted in listing 5.11. The code from line 253 to line 277 is omitted due to the page limit. This code listing displays a quite typical scenario that logging feature cross cuts with other business logic. The business logic of method *visitEnter* is about node dependency resolving. Logging code is interwoven with the main business logic of the method here. The call site of *Logger.debug* is at line 283. It does not have return value. So we do not need to perform forward slicing. It takes *buffer.toString()* as its parameter. The backward solo-slicing will start from *buffer.toString()*. After removing call site, *buffer* in line 281 will lose its only dependent. Thus line 281 would be deleted which causes *buffer* in line 280 lose its only dependent. Such a solo-slicing chain will go all the way back to line 250. The solo-slicing will not stop until line 250 is deleted. In other 59 call sites, the operation and removing process is similar. After we remove all call sites of the seed method. We can remove the seed method definition because this seed method is in a third party library.

To this end, we have removed the debugging information feature from Apache Maven project. We test Maven by the test cases designed by us. We find that Maven cannot do debugging information logging any more. Other functions of Maven work normally.

## 5.5 Discussion

### 5.5.1 Solo-slicing

To solve the problem of redundancy removal, we propose a new slicing concept and technique, solo-slicing. Solo-slicing could slice out the statements that are *only*

Table 5.3: *Logger.log* Call Sites in Apache Maven project

| **SEEDS: java.util.logging.Logger.log** | |
|---|---|
| **The Classes that have SEEDS call sites** | **Call Sites Number** |
| org.apache.maven.bridge.MavenRepositorySystem | 1 |
| org.apache.maven.classrealm.DefaultClassRealmManager | 8 |
| org.apache.maven.DefaultMaven | 3 |
| org.apache.maven.lifecycle.internal.LifecycleDebugLogger | 20 |
| org.apache.maven.····.DefaultLifecyclePluginAnalyzer | 1 |
| org.apache.maven.lifecycle.internal.MojoDescriptorCreator | 1 |
| org.apache.maven.····.MultiThreadedBuilder | 2 |
| org.apache.lifecycle.DefaultLifecycles | 1 |
| org.apache.maven.LoggingRepositoryListener | 2 |
| org.apache.maven.plugin.internal.DefaultMavenPluginManager | 5 |
| org.apache.maven.····.DefaultPluginPrefixResolver | 3 |
| org.apache.maven.····.DefaultPluginVersionResolver | 5 |
| org.apache.maven.plugin.DebugConfigurationListener | 2 |
| org.apache.maven.project.DefaultProjectBuildingHelper | 2 |
| org.apache.maven.project.artifact.MavenMetadataSource | 1 |
| org.apache.maven.project.DefaultProjectDependenciesResolver | 1 |
| org.apache.maven.toolchain.DefaultToolchainsBuilder | 1 |

Listing 5.11: The Code Around One Call Site of Seed Method in Apache Maven Project

```
249  public boolean visitEnter( DependencyNode node ){
250    StringBuilder buffer = new StringBuilder( 128 );
251    buffer.append( indent );
252    org.eclipse.aether.graph.Dependency dep
         = node.getDependency();
253    if ( dep != null ){
         ...//omit due to page limit.
277    }else{
278      buffer.append( project.getGroupId() );
279      buffer.append(':').append(project.getArtifactId());
280      buffer.append(':').append(project.getPackaging());
281      buffer.append(':').append(project.getVersion());
282    }
283    logger.debug( buffer.toString() );
284    indent += "␣␣␣";
285    return true;
286  }
```

affected by or *only* affect the slicing criteria. As a side product of our research on feature-based software customization, solo-slicing actually could be applied to many other research areas independently. Solo-slicing might improve the slicing efficiency on software debugging, transformation, and binary difference comparison tasks in certain scenarios. The possible impact there is worth further investigation.

### 5.5.2 Definitions of Feature

The term feature may have different meanings in different contexts. The most well-known usage context of the term feature is in software engineering domain. In software engineering, using features, instead of components or modules, to organize teams, plan development schedules, and deliver products is a new trend nowadays. A feature represents some values to users. Or in more plain words, a feature enables a user to do something. Compared with having each team work on a component respectively, making each team work on a feature is more user-centric, which not only avoids developers to measure their workloads by lines of code and work on some "invented works", but also encourage them to think about the problem from a big picture in the shoes of users. Compared with a component, feature is across functions, across front ends and back ends, and across modules in nature.

In this paper, we define a feature as all call sites of an API of interests. To some extents, our feature based software customization can be called as call sites based software customization. We define term feature in this paper in such a way is for making the feature analysis operational. Though this paper and software engineering domain do not share the same definition of the term feature, we both try to use this term to capture and express some common insights behind it, which is across functions, across frond ends and back ends, and across modules.

### 5.5.3 Future Work

This chapter focuses on a novel research question, the formalization and analysis of this research question, and potential techniques for solving the problem. We evaluate our approach from several perspectives. However, we have not done large-scale experiments yet in this preliminary feasibility study. In the future, we will conduct more through empirical study to evaluate our approach in a more comprehensive way.

# Chapter 6
# Conclusion

Bloatware problem is an emerging issue in modern software engineering. This problem has caused many negative consequences including wasted technical resources and security risks. In the era of mobile computing, cloud computing, wearable devices, and Internet of things, new hard constrains are imposed to the limited resources, which makes the bloatware problem more urgent than ever before.

In this dissertation, we present a fully automated tool called JRed for trimming unused methods and classes from both Java application code and the Java Runtime JRE core libraries. We have implemented a prototype on top of Soot. Our experimental results show that JRed can reduce Java code size by 44.5% and 82.5% on average for Java application code and runtime JRE library code respectively. We also evaluated the effectiveness of JRed on trimming security related vulnerabilities in the Java Runtime JRE, and the results show that nearly half of the known security vulnerabilities can be trimmed away with the specialized JREs for each benchmark program. Overall, our evaluation results show that our tool could be very effective on reducing the code size, code complexity, and attack surfaces for both Java applications and runtime JRE libraries in certain scenarios.

We also present an approach to trimming compile-time redundancy and install-time redundancy from Android applications. We have implemented a fully automated tool called RedDroid. Our experimental results show that RedDroid can reduce Android application size by around 15% on average via removing unused bytecode. Code complexity, measured by a set of well-known metrics, is also reduced significantly. RedDroid can also identify and remove redundant Android wear SDKs, which can reduce the size of related applications by another 20% on average. By removing redundant embedded ABIs, the size of applications can be reduced by

additional 7% on average. If an application has all three kinds of software bloat, in sum its size can be reduced by around 42%. Overall, our evaluation results show that our approach is effective on reducing both compile-time redundancy and install-time redundancy. In addition, our results depict the landscape of bloatware issue in the Android application domain for the first time. The results we reported can help developers better identify their pain point regarding application resource consumption issue and better plan their development and build process.

In addition, we discuss and formally define a novel research problem of feature-based software customization. Based on that, we present a multistep static program slicing based approach to conducting feature-based software customization. Additionally, as a part of of the multistep approach, we propose a new concept, solo-slicing, which can slice out the statements that are *only* affected by or *only* affect the slicing criteria. Our approach can help remove a feature from the software safely and clean all redundancy caused by this removal, and can potentially help legacy code retrofitting and maintenance.

Our research still has several limitations at this stage. First, reflections may cause unsoundness in our program analysis and transformation. In our research, we have used both dynamic and static approaches to overapproximate this undecidable problem. The data yielded from both directions are promising. The static approach might be even better since it is sound on almost all programs and it can predict when it will be unsound before the program transformation is done. This conclusion is based on the reflection usage patterns we observed in thousands of applications. If we can analyse even larger scale of application samples, the validation of the approach might be additionally strengthened. This work can be done in the future. The other limitation is that we do not customize the binary part of a program. As we have reasoned in the dissertation, our approach can be applied to binary as well with some minor extension. This is also one of our future works. Other future works include customizing the entire Android system, and connecting our feature based software customization to the Android permission system, which may help us revoke those over-requested permissions claimed by Android applications.

# Bibliography


[1] Bodden, E., A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini (2011) "Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders," in *Proceedings of the 33rd International Conference on Software Engineering*, ACM, pp. 241–250.

[2] Xu, G., N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky (2010) "Software Bloat Analysis: Finding, Removing, and Preventing Performance Problems in Modern Large-scale Object-Oriented Applications," in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, ACM, pp. 421–426.

[3] McGrenere, J. (2000) ""Bloat": The Objective and Subject Dimensions," in *CHI '00 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '00, ACM, New York, NY, USA, pp. 337–338.

[4] McDaniel, P. (2012) "Bloatware Comes to the Smartphone," *IEEE Security & Privacy*, **10**(4), pp. 0085–87.

[5] Allen, D. (April 1993) "Editorial: Fatware Strategies," *Byte*, **18**(4), p. 12.

[6] Holzmann, G. J. (2015) "Code Inflation," *Software, IEEE*, **32**(2).

[7] Bu, Y., V. Borkar, G. Xu, and M. J. Carey (2013) "A Bloat-aware Design for Big Data Applications," in *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, ACM, pp. 119–130.

[8] Adam, M. (2013), "Java 7 Applet 0day Exploit," Http://www.cs.bu.edu/~goldbe/teaching/HW55813/marc.pdf.

[9] Pugh, W. (1999) "Compressing Java Class Files," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, ACM, pp. 247–258.

[10] Bradley, Q., R. N. Horspool, and J. Vitek (1998) "JAZZ: An Efficient Compressed Format for Java Archive Files," in *Proceedings of the 1998*

*Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '98, IBM Press, pp. 7–15.

[11] Tip, F., C. Laffra, P. F. Sweeney, and D. Streeter (1999) "Practical Experience with an Application Extractor for Java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, ACM, pp. 292–305.

[12] Johnson, J. (2009), "CHAOS 2009," .

[13] Sweeney, P. F. and F. Tip (2000) "Extracting Library-based Object-Oriented Applications," in *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, ACM, pp. 98–107.

[14] Wagner, G., A. Gal, and M. Franz (2011) "'Slimming' a Java Virtual Machine by Way of Cold Code Removal and Optimistic Partial Program Loading," *Science of Computer Programming*, **76**(11).

[15] Google, I. (2017), "Improve Your Code with Lint," Https://developer.android.com/studio/write/lint.html.

[16] Edwards, D. (2007), "Packer: A JavaScript Compressor," Http://dean.edwards.name/weblog/2007/04/packer3/.

[17] Crockford, D. (2003), "JSMin: The JavaScript Minifier," Http://www.crockford.com/javascript/jsmin.html.

[18] Souders, S. (2008) "High-performance Web Sites," *Communications of the ACM*, **51**(12), pp. 36–41.

[19] Likarish, P., E. Jung, and I. Jo (2009) "Obfuscated Malicious Javascript Detection Using Classification Techniques," in *Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on*, IEEE, pp. 47–54.

[20] Oberländer, J. (2003) "Applying Source Code Transformation to Collapse Class Hierarchies in C++," *Study Thesis, System Architecture Group, University of Karlsruhe, Germany*.

[21] Sweeney, P. F. and F. Tip (1998) "A Study of Dead Data Members in C++ Applications," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, ACM, pp. 324–332.

[22] De Sutter, B., B. De Bus, and K. De Bosschere (2002) "Sifting out the Mud: Low Level C++ Code Reuse," in *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, ACM, pp. 275–291.

[23] Xu, G. (2012) "Finding Reusable Data Structures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, ACM, pp. 1017–1034.

[24] ——— (2013) "CoCo: Sound and Adaptive Replacement of Java Collections," in *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, Springer-Verlag, Berlin, Heidelberg, pp. 1–26.

[25] Hosking, A. L., N. Nystrom, D. Whitlock, Q. Cutts, and A. Diwan (2001) "Partial Redundancy Elimination for Access Path Expressions," *Software: Practice and Experience*, **31**.

[26] Whitlock, D. and A. L. Hosking (2001) "A Framework for Persistence-Enabled Optimization of Java Object Stores," in *Revised Papers from the 9th International Workshop on Persistent Object Systems*, POS-9, Springer-Verlag, London, UK, UK, pp. 4–17.

[27] Xu, G., N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky (2014) "Scalable Runtime Bloat Detection Using Abstract Dynamic Slicing," *ACM Transaction of Software Engineering Methodology*, **23**(3), pp. 23:1–23:50.

[28] Nguyen, K. and G. Xu (2013) "Cachetor: Detecting Cacheable Data to Remove Bloat," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, ACM, pp. 268–278.

[29] Morgenthaler, J. D., M. Gridnev, R. Sauciuc, and S. Bhansali (2012) "Searching for Build Debt: Experiences Managing Technical Debt at Google," in *Proceedings of the Third International Workshop on Managing Technical Debt*, MTD '12, IEEE Press, Piscataway, NJ, USA, pp. 1–6.

[30] Wang, P., J. Yang, L. Tan, R. Kroeger, and J. D. Morgenthaler (2013) "Generating Precise Dependencies for Large Software," in *Proceedings of the Forth International Workshop on Managing Technical Debt*, pp. 47–50.

[31] Vakilian, M., R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni (2015) "Automated Decomposition of Build Targets," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, IEEE Press, Piscataway, NJ, USA, pp. 123–133.

[32] Ryder, B. G. and F. Tip (2001) "Change Impact Analysis for Object-oriented Programs," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, ACM, New York, NY, USA, pp. 46–53.

[33] Larsen, P., A. Homescu, S. Brunthaler, and M. Franz (2014) "SoK: Automated Software Diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, IEEE Computer Society, Washington, DC, USA, pp. 276–291.

[34] Snow, K. Z., F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi (2013) "Just-In-Time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization," in *Security and Privacy (SP), 2013 IEEE Symposium on*, IEEE, pp. 574–588.

[35] Collberg, C., C. Thomborson, and D. Low (1998) "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, pp. 184–196.

[36] Barrantes, E. G., D. H. Ackley, S. Forrest, and D. Stefanović (2005) "Randomized Instruction Set Emulation," *ACM Transactions on Information and System Security (TISSEC)*, **8**(1), pp. 3–40.

[37] Thies, A. and E. Bodden (2012) "RefaFlex: Safer Refactorings for Reflective Java Programs," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, ACM, pp. 1–11.

[38] Livshits, B., J. Whaley, and M. S. Lam (2005) "Reflection Analysis for Java," in *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, Springer-Verlag, Berlin, Heidelberg, pp. 139–160.

[39] Braux, M. and J. Noyé (1999) "Towards Partially Evaluating Reflection in Java," in *Proceedings of the 2000 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '00, ACM, pp. 2–11.

[40] Christensen, A. S., A. Møller, and M. I. Schwartzbach (2003) "Precise Analysis of String Expressions," in *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, Springer-Verlag, Berlin, Heidelberg, pp. 1–18.

[41] Li, D., Y. Lyu, M. Wan, and W. G. Halfond (2015) "String Analysis for Java and Android Applications," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ACM, pp. 661–672.

[42] Shannon, D., S. Hajra, A. Lee, D. Zhan, and S. Khurshid (2007) "Abstracting Symbolic Execution with String Analysis," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*, IEEE Computer Society, pp. 13–22.

[43] Lhoták, O. (2002) *Spark: A Flexible Points-to Analysis Framework for Java*, Master's thesis, McGill University.

[44] Grove, D., G. DeFouw, J. Dean, and C. Chambers (1997) "Call Graph Construction in Object-Oriented Languages," in *Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, ACM, pp. 108–124.

[45] Grove, D. and C. Chambers (2001) "A Framework for Call Graph Construction Algorithms," *ACM Transaction of Programing Language System*, **23**(6), pp. 685–746.

[46] Agrawal, G., J. Li, and Q. Su (2002) "Evaluating a Demand Driven Technique for Call Graph Construction," in *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, Springer-Verlag, London, UK, UK, pp. 29–45.

[47] Tip, F. and J. Palsberg (2000) "Scalable Propagation-based Call Graph Construction Algorithms," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, ACM, pp. 281–293.

[48] Weiser, M. (1981) "Program Slicing," in *Proceedings of the 5th International Conference on Software Engineering (ICSE '81)*, IEEE Press, Piscataway, NJ, USA, pp. 439–449.

[49] Arvind, D. and P. Shankar (2006), "Slicing of Java Programs using the Soot Framework," .

[50] Ottenstein, K. J. and L. M. Ottenstein (1984) "The Program Dependence Graph in a Software Development Environment," in *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, ACM, New York, NY, USA, pp. 177–184.

[51] Horwitz, S., T. Reps, and D. Binkley (1988) "Interprocedural Slicing Using Dependence Graphs," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, ACM, New York, NY, USA, pp. 35–46.

[52] Wang, T. and A. Roychoudhury (2008) "Dynamic Slicing on Java Bytecode Traces," *ACM Transaction of Programing Language System*, **30**(2), pp. 10:1–10:49.

[53] Hammacher, C., A. Zeller, V. Dallmeier, M. Burger, and S. Hack (2008) "Design and Implementation of an Efficient Dynamic Slicer for Java," *Bachelor's Thesis, November*.

[54] Treffer, A. and M. Uflacker (2014) "Dynamic Slicing with Soot," in *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP '14, ACM, New York, NY, USA, pp. 1–6.

[55] Cao, Y., Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen (2015) "EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework." in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*, pp. 1–15.

[56] Arzt, S., S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel (2014) "FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, ACM, New York, NY, USA, pp. 259–269.

[57] Octeau, D., S. Jha, and P. McDaniel (2012) "Retargeting Android Applications to Java Bytecode," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, ACM, New York, NY, USA, pp. 6:1–6:11.

[58] Pan, B. (2017), "dex2jar," Https://github.com/pxb1988/dex2jar.

[59] Au, K. W. Y., Y. F. Zhou, Z. Huang, and D. Lie (2012) "Pscout: Analyzing the Android Permission Specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM, pp. 217–228.

[60] Felt, A. P., E. Chin, S. Hanna, D. Song, and D. Wagner (2011) "Android Permissions Demystified," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, ACM, New York, NY, USA, pp. 627–638.

[61] Connor Tumbleson, R. W. (2017), "Apktool: A Tool for Reverse Engineering Android apk Files," Https://ibotpeaches.github.io/Apktool/.

[62] (2017), "FernFlower," Https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine.

[63] Vallée-Rai, R., P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan (1999) "Soot - a Java Bytecode Optimization Framework," in *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, IBM Press, pp. 13–23.

[64] Jajodia, S., A. K. Ghosh, V. Swarup, C. Wang, and X. S. Wang (2011) *Moving Target Defense*, Springer.

[65] Whaley, J. (2003) "Joeq: A Virtual Machine and Compiler Infrastructure," in *Proceedings of the 2003 Workshop on Interpreters, Virtual Machines and Emulators*, IVME '03, ACM, pp. 58–66.

[66] Vallee-Rai, R. and L. J. Hendren (1998) *Jimple: Simplifying Java Bytecode for Analyses and Transformations*, *Tech. rep.*, Sable Research Group, McGill University.

[67] Whaley, J. and M. S. Lam (2004) "Cloning-based Context-sensitive Pointer Alias Analysis Using Binary Decision Diagrams," in *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, ACM, New York, NY, USA, pp. 131–144.

[68] Dean, J., D. Grove, and C. Chambers (1995) "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP '95, Springer-Verlag, London, UK, UK, pp. 77–101.

[69] Bacon, D. F. and P. F. Sweeney (1996) "Fast Static Analysis of C++ Virtual Function Calls," in *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, October 6-10, 1996.*, pp. 324–341.

[70] Blackburn, S. M., R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann (2006) "The DaCapo Benchmarks: Java Benchmarking Development and Analysis," in *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '06, ACM, New York, NY, USA, pp. 169–190.

[71] Chidamber, S. R. and C. F. Kemerer (1994) "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, **20**(6), pp. 476–493.

[72] Project Analyzer v10.2 (2014), "Chidamber and Kemerer Object-Oriented Metrics Suite," Http://www.aivosto.com/project/help/pm-oo-ck.html.

[73] Misra, S. C. and V. C. Bhavsar (2003) "Relationships Between Selected Software Measures and Latent Bug-density: Guidelines for Improving Quality," in *Proceedings of the 2003 International Conference on Computational Science and Its Applications: PartI*, ICCSA'03, Springer-Verlag, Berlin, Heidelberg, pp. 724–732.

[74] Spinellis, D. (2006) *Code Quality: The Open Source Perspective*, Addison-Wesley.

[75] Tip, F. and J. Palsberg (2000) "Scalable Propagation-based Call Graph Construction Algorithms," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, ACM, New York, NY, USA, pp. 281–293.

[76] Dean, J., D. Grove, and C. Chambers (1995) "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *European Conference on Object-Oriented Programming*, Springer, pp. 77–101.

[77] Cormen, T. H. (2009) *Introduction to Algorithms*, MIT press.

[78] (2017), "VMProtect," Http://vmpsoft.com/.

[79] Coogan, K., G. Lu, and S. Debray (2011) "Deobfuscation of Virtualization-Obfuscated Software: a Semantics-Based Approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*, ACM, pp. 275–284.

[80] Spinellis, D. D. (2005), "ckjm Chidamber and Kemerer Metrics Software v1.9," Http://www.spinellis.gr/sw/ckjm/.

[81] Virtual Machinery (2017), "WMC, CBO, RFC, LCOM, DIT, NOC - The Chidamber and Kemerer Metrics," Http://www.virtualmachinery.com/sidebar3.htm.

[82] Jiang, Y., C. Zhang, D. Wu, and P. Liu (2015) "A Preliminary Analysis and Case Study of Feature-Based Software Customization (Extended Abstract)," in *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, August 3-5, 2015*, pp. 184–185.

[83] ——— (2016) "Feature-Based Software Customization: Preliminary Analysis, Formalization, and Methods," in *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering, (HASE)*, pp. 122–131.

[84] Aho, A. V., R. Sethi, and J. D. Ullman (1986) *Compilers, Principles, Techniques*, Addison wesley Boston.

[85] Wang, S., P. Wang, and D. Wu (2015) "Reassembleable Disassembling." in *USENIX Security Symposium*, pp. 627–642.

[86] McConnell, S. and D. Johannis (2004) *Code Complete*, 2nd ed., Microsoft Press.

[87] Mockus, A., R. T. Fielding, and J. Herbsleb (2000) "A Case Study of Open Source Software Development: The Apache Server," in *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, ACM, New York, NY, USA, pp. 263–272.

[88] Dahse, J., N. Krein, and T. Holz (2014) "Code Reuse Attacks in PHP: Automated POP Chain Generation," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ACM, pp. 42–53.

[89] Laskov, P. and N. Šrndić (2011) "Static Detection of Malicious JavaScript-bearing PDF Documents," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, ACM, pp. 373–382.

[90] Geer Jr., D. E. (2008) "Complexity is the Enemy," *IEEE Security and Privacy*, **6**(6), pp. 88–88.

[91] Rinard, M. (2011) "Manipulating Program Functionality to Eliminate Security Vulnerabilities," in *Moving Target Defense*, Springer, pp. 109–115.

[92] Kiczales, G. and E. Hilsdale (2001) "Aspect-Oriented Programming," in *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, ACM, New York, NY, USA, pp. 313–313.

[93] Xu, J., P. Guo, M. Zhao, R. F. Erbacher, M. Zhu, and P. Liu (2014) "Comparing Different Moving Target Defense Techniques," in *Proceedings of the First ACM Workshop on Moving Target Defense*, ACM, pp. 97–107.

[94] Reps, T., S. Horwitz, and M. Sagiv (1995) "Precise Interprocedural Dataflow Analysis via Graph Reachability," in *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '95)*, ACM, New York, NY, USA, pp. 49–61.

[95] Jackson, D. and E. J. Rollins (1994) "A New Model of Program Dependences for Reverse Engineering," in *Proceedings of the 2Nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, ACM, New York, NY, USA, pp. 2–10.

[96] Reps, T. and G. Rosay (1995) "Precise Interprocedural Chopping," in *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '95, ACM, pp. 41–52.

[97] Horwitz, S., T. Reps, and D. Binkley (1988) "Interprocedural Slicing Using Dependence Graphs," in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, PLDI '88, ACM, New York, NY, USA, pp. 35–46.

[98] Sridharan, M., S. J. Fink, and R. Bodik (2007) "Thin Slicing," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, ACM, New York, NY, USA, pp. 112–122.

[99] Kuck, D. J., Y. Muraoka, and S.-C. Chen (1972) "On the Number of Operations Simultaneously Executable in Fortran-Like Programs and Their Resulting Speedup," *Computers, IEEE Transactions on*, **100**(12), pp. 1293–1310.

[100] Larsen, L. and M. J. Harrold (1996) "Slicing Object-Oriented Software," in *Proceedings of the 18th International Conference on Software Engineering*, ICSE '96, IEEE Computer Society, pp. 495–505.

[101] Allen, E., R. Cartwright, and B. Stoler (2002) "DrJava: A Lightweight Pedagogic Environment for Java," in *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, ACM, pp. 137–141.

# Vita

## Yufei Jiang

Yufei Jiang is currently a Ph.D. candidate in the College of Information Sciences and Technology of Pennsylvania State University, where he is a member of the Software System Security Research Lab. His research focuses on softwre engineering and security, especially software analysis and program customization, including Java software customization, Android application customization, software feature based customization, software obfuscation, and software analysis for other security issues. He received the B.S. degree in Software Institute from Nanjing University in 2011.

### Publications


[1] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. "RedDroid: Android Application Redundancy Customization Based on Static Analysis," *under review*, 2017.

[2] Xiao Liu, Yufei Jiang, and Dinghao Wu. "A Lightweight Verification Framework for Regular Expressions," *under review*, 2017.

[3] Jiang Ming, Dongpeng Xu, Yufei Jiang, and Dinghao Wu. "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking," in *The 26th Usenix Security Symposium Conference (Usenix Security 2017)*, Vancouver, BC, Canada, August 16–18, 2017.

[4] Xiao Liu, Yufei Jiang, Lawrence Wu, and Dinghao Wu. " Natural Shell: An Assistant for End-user Scripting," *International Journal of People-Oriented Programming (IJPOP)*, **5**(1):1–18, 2016.

[5] Yufei Jiang, Dinghao Wu, and Peng Liu. "JRed: Program Customization and Bloatware Mitigation Based on Static Analysis," in *The 40th IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC 2016)*, Atlanta, Georgia, USA, June 10–14, 2016.

[6] Yufei Jiang, Xiao Liu, Fangxiao Liu, Dinghao Wu, and Chimay J. Anumba. "An Analysis of BIM Web Service Requirements and Design to Support Energy Efficient Building Lifecycle," *Journal of Buildings*, **6**:19–43, 2016.

[7] Pei Wang, Shuai Wang, Jiang Ming, Yufei Jiang, and Dinghao Wu. "Translingual Obfuscation," in *IEEE European Symposium on Security and Privacy (Euro S&P 2016)*, Saarbrucken, Germany, March 21–24, 2016.

[8] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. "A First Step Towards Feature-based Software Customization," in *Proceedings of the 17th IEEE High Assurance Systems Engineering Symposium (HASE 2016)*, Orlando, Florida, USA, January 7–9, 2016.

[9] Yufei Jiang, Nan Yu, Jiang Ming, Sanghoon Lee, Jason W. DeGraw, John I. Messner, John Yen, and Dinghao Wu. "Automatic Building Information Model Query Generation," *Journal of Information Technology in Construction (ITCon)*, **20**:518–535, 2015.

[10] Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. "A Preliminary Analysis and Case Study of Feature-based Software Customization (Extended Abstract)," in *2015 IEEE International Conference on Software Quality, Reliability and Security (QRS 2015)*, Vancouver, Canada, August 3–5, 2015.

[11] Yufei Jiang, Nan Yu, Jiang Ming, Lannan Luo, Chong Zhou, Sanghoon Lee, Abdou Jallow, Prasenjit Mitra, John Yen, Robert Leicht, John I. Messner, and Dinghao Wu. "Simplify, Automate, and Integrate the BIM Data Exchange Process," poster in *2013 EEB Hub Poster Session and Reception*, The Pennsylvania State University, University Park, PA, USA, April 16, 2013.

[12] Yufei Jiang, Nan Yu, Jiang Ming, Lannan Luo, Chong Zhou, Sanghoon Lee, Abdou Jallow, Prasenjit Mitra, John Yen, Robert Leicht, John I. Messner, and Dinghao Wu. "Subtask 3.2: Building Information Modeling (BIM) Data Hub," poster in *2013 Building SyENERGY - The EEB HUB Spring Conference*, The Navy Yard - Philadelphia, PA, USA, March 20–22, 2013.

[13] Nan Yu, Yufei Jiang, Lannan Luo, Sanghoon Lee, Abdou Jallow, John Yen, John Messner, Robert Leicht, and Dinghao Wu. "Integrating BIMserver and OpenStudio for Energy Efficient Building," in *2013 ASCE International Workshop on Computing in Civil Engineering (IWCCE)*, Los Angeles, CA, USA, June 23–25, 2013.

[14] Yufei Jiang, Jiang Ming, Dinghao Wu, John Yen, Prasenjit Mitra, John I. Messner, and Robert Leicht. "BIM Server requirements to support the energy efficient building lifecycle," in *2012 ASCE International Workshop on Computing in Civil Engineering (ASCE 2012)*, Clearwater Beach, FL, June 17–20, 2012.

[15] Yufei Jiang, Yuan Huang, and Ruizhi Gao. "The Video of Xland: Two Core Use Cases of 3D Blog," in *Videos Program of the 2011 ACM Conference on Computer Supported Cooperative Work (CSCW 2011)*, Hangzhou, China, March 19–23, 2011.