JRed: Program Customization and Bloatware Mitigation Based on Static Analysis

Yufei Jiang, Dinghao Wu, and Peng Liu
The Pennsylvania State University
University Park, PA 16802, USA
{yzj107,dwu,pliu}@ist.psu.edu

***Abstract*—Modern software engineering practice increasingly brings redundant code into software products, which has caused a phenomenon called *bloatware*, 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 paper, we propose a new static-analysis-enabled approach to trimming unused code from both Java applications and Java Runtime Environment (JRE) 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.**

I. Introduction

With the rapid development of modern software engineering, the size of the software products keep expanding, leading to the problem of "bloatware" [1]. The bloatware has become an emerging urgent issue, especially in object-oriented programming languages such as Java. Encapsulation, polymorphism, inheritance, big libraries, frameworks, and other abstraction and modularization mechanisms improve programmer productivity and software reliability, but on the other hand bring a large amount of unnecessary bloat code into the software product.

Code size has a profound impact on many aspects of a system, including software or especially smartphone apps 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.

For example, there is a Java 7 exploitable bug residing in the Java Runtime JRE library classes. In particular, an attacker can abuse some other methods and classes after he or she disables the SecurityManager of Java Applet by taking advantage of that exploitable bug [2]. More Specifically, by exploiting the method findClass in Class java.lang.ClassLoader and the method methodFinder in Class com.beans.Finder, attackers can call method getField, which can get any private field in class sun.awt.SunToolkit. Thus, attackers can access and modify some sensitive private fields in sun.awt.SunToolkit such as AccessControlContext. 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 take advantage of an exploitable bug to walk around security policies and abuse it [2]. If we cut sun.awt.SunToolkit Class and other unused classes from the JRE libraries for the Applet usage scenario, many similar security problems will go away.

Previous research on bloatware has different scopes with our research or incurs a number of limitations. Some works focus on the local optimization of used code [3], [4], [5]. Xu [5] proposes a method to mitigate the bloatware issue by finding reusable data structures. Coco [6] adaptively replaces ineffective Java collections with effective ones in large software systems. These approaches are mostly for dynamic code reuse and optimization. Pugh [7] and Bradley et al. [8] propose some packing and encoding methods to reduce the size of jar files, but they do not reduce the actual code size. Morgenthaler et al. [9] try to lower the difficulty of dependency management and target building caused by huge monolithic code base. They remove the build files associated with dead code, identify "unbuildable targets", and unnecessary command line flags.

In this paper, we propose a fully automated static approach to trimming unused redundant bytecode from both Java applications 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 trimming.

Our approach is not intended to be a general approach that can be applied anywhere in any scenario. Instead, our approach could be very useful and effective in certain applications and scenarios. For example, part of our contribution is JRE customization. There are some situations where we can perform JRE customization. We might not customize the JRE in our personal laptops where we frequently install and remove applications. However, for those computing environments that run certain fixed Java applications, such as servers, cloud instances, sensors, GPS navigators, and computers in classrooms, labs, test centers, and offices, a customized JRE could be a desired feature. In these scenarios, we can trim JRE library based on these applications. We only trim the intersection of the computed JRE "bloat" for each Java application. Regarding the application customization, a good example is about mobile apps. The disk sizes of mobile devices actually are very limited. Most smartphones are equipped with 16GB disks. After excluding the size occupied by the OS, photos, and other documents, only a few GB are left for installing new apps. Considering the fact that many apps on smartphones are pre-installed by the vendors which cannot be removed, the left space is even tighter. It is very easy to find mobile apps whose sizes exceed 1GB on today's App stores. Potentially, we can

TABLE I: 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%

gain benefit by applying our technologies to those apps.

We have implemented our approach in a prototype tool called JRed on top of Soot [10]. 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%. From the end user point of view, the device disk footprint is reduced by roughly 50% with use of JRed. 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 [11]. Besides these direct impacts on the transformed program, our tool also can provide useful information and guidelines to assist software architects and developers for software customization, configuration, optimization, and refactoring.

In summary, we make the following contributions:

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

II. 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 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 a 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 I. 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 I. 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 I. 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.

[1]We initially built a tool based on Joeq [12] to conduct this case study. Our current prototype implementation JRed is based on Soot [10]. 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.

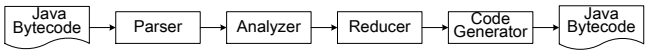

Fig. 1: JRed Architecture

III. Approach

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

A. Overview

Fig. 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 [13]. 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, 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 fields being accessed, then the entire class is removed. In the next step, the code generator transforms the customized IR back into the Java bytecode.

B. 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 [14], to facilitate call graph construction. Points-to analysis builds the call graph on the fly [15]. Compared with some other popular call graph construction techniques, such as Class Hierarchy Analysis (CHA) [16] and Rapid Type Analysis (RTA) [17], points-to analysis builds a more precise call graph. Points-to analysis for Java is different than for C [14]. 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.

C. 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 (e.g. the method invocations before the main method is called). Although precisely determining all method invocations in a sound and complete approach is undecidable, it is possible to over-approximate the problem and trim the code conservatively (soundly). In other words, we do not delete all unused methods. We delete a large proportion of them. The deleted ones are guaranteed not to be used. We thus ensure the correctness of the resulted lean Java code.

To this end, we adopt a set of over-approximate rules. The first rule is not to delete native methods. Native methods offer an interface to call those functions written in C via Java Native Interface (JNI). Setting the analysis boundary at 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 second 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 over-approximate 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.

D. Code Generator

At last, the code generator transforms the customized Soot IR to Java byte code and writes it into corresponding Java class files. In practice, this step may need additional work due to some software engineering issues in real world programs. 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 a challenge for our analysis. In these cases, JRed has to extract the resources files first before analysis, and merge those resource files with the transformed class files at the code generation step.

IV. Evaluation

In this section, we evaluate JRed by applying it to 9 Java programs selected from DaCapo 9.12-bach benchmark [18]. 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.

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 and 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 Java applications, JRE core libraries, and Java-app+JRE together?

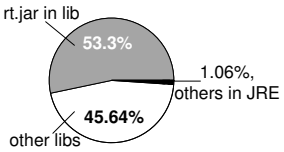

Fig. 2: The Java Runtime JRE Structure

Q2: What is the impact of our redundant code trimming technique on the code complexity of Java applications and JRE core libraries?
Q3: What is the impact of our redundant code trimming technique on memory footprint?
Q4: What is the impact of our redundant code trimming technique on Java application execution and garbage collection time?
Q5: What is the impact of our redundant code trimming technique on software reliability and security?

A. Code Size

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

1) Java Application Code Size: The experimental results are shown in Table II and Fig. 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 II and Fig. 3 shows 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 of a Java program. The second metric and the third metric have subtle difference. The total size that all class files actually occupy on the disk is usually larger than the sum of the sizes of these files; the latter is equal to adding up every file's bytes number. 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 it actually occupies the size of a disk sector on the hard disk. In Table II, 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" displays the number of reduced version size divided by original size.

In Table II and Fig. 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, and 6 out of 9 could be trimmed more than 40% size off. The lowest reduced-original sum of all class files size ratio is 21.78% (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.

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. Fig. 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 45.64% 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 III and Fig. 4. The metrics we used in Table III and Fig. 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 II, we do not have a column "original" in Table III, 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 significant. Meanwhile, it is not surprising to see that 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) Java App+JRE All Together: We define the application and JRE (not only the core libraries in JRE) together as "Java-App+JRE". Fig. 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

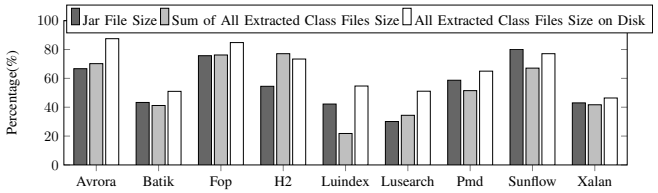

Fig. 3: Reduced-Original Size Ratios of DaCapo

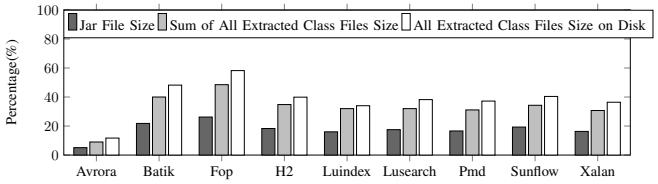

Fig. 4: Reduced-Original rt.jar Ratios of DaCapo

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 Fig. 5, we can see that a big JRE core library (93.9MB) causes the sizes of all Java-App+JREs to be 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.

B. 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 IV, Fig. 6, and Fig. 7. We measure the code complexity by the Chidamber and Kemerer (CK) object-oriented metrics and two other metrics. The CK object-oriented metrics suite is proposed

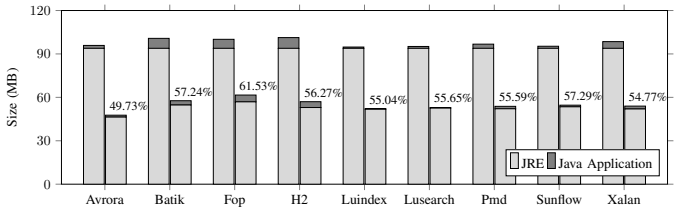

Fig. 5: Java App+JRE Overall Ratios

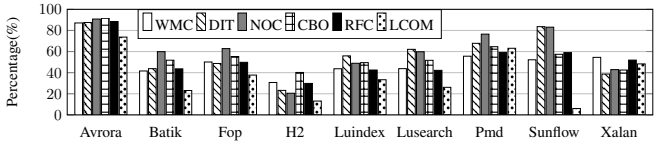

Fig. 6: Reduced-Original Application CK Metrics Ratios

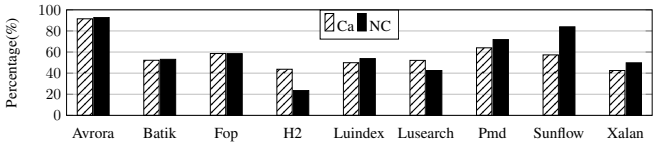

Fig. 7: Reduced-Original Application LCOM & Ca Ratios

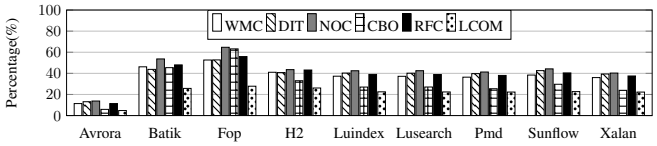

Fig. 8: Reduced-Original rt.jar CK Java Metrics Ratios

by Chidamber and Kemerer [19], [20] 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 aim 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 [21], 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 *immediate* subclasses that 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 larger number of immediate 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

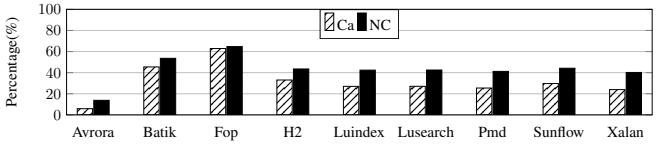

Fig. 9: Reduced-Original rt.jar LCOM & Ca Ratios

TABLE II: 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 III: Customized rt.jar of DaCapo benchmark applications code size before and after unused code trimming comparison

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

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

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 IV show the experimental results of each benchmark application before and after JRed trimming on the CK Java metrics. Fig. 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 41%. 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) [22]. 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. Fig. 7 and the last two rows of each sub-table in Table IV 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.

By comparing Fig. 6 and Fig. 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 Table V, Fig. 8, and Fig. 9. 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.

C. 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 VI and Table VII. The lean version benchmark has slightly smaller memory footprint, but mostly 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

TABLE IV: 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

TABLE V: 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

WMC	weighted methods/class	RFC	Response for a Class
DIT	Depth Inheritance Tree	LCOM	Lack of method cohesion
NOC	Number of Children	Ca	Afferent couplings
CBO	Object class coupling	NC	Number of Classes

TABLE VI: 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 VII: Avrora Execution and Garbage Collection Time

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

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.

D. 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 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 VIII. 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. The results show that our tool is effective in reducing software attack surfaces. By specializing Java Runtime JRE for different applications, we can achieve more diversity, resulting enhanced moving target defense [11].

E. Performance

In addition to the effectiveness evaluation, we also measured the running time performance for our tool JRed. We investigate 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.

The results are shown in Table IX. 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 Fig. 6 and Fig. 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.

F. 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 [11].

V. Discussion and Future Work

A. Impact on Reliability and Security

"Complexity is the enemy of security" [23]. However, in real world, the pace of software complexity increasing does not slow down. According to an estimate made by McConnell [24], 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.[2] Assuming the bugs are distributed randomly, the number of bugs we can trim is proportional to the percentage of the code size we can reduce. In the evaluation section, 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, which reduces attack surface.

In addition, redundant code trimming gives opportunities to those expensive security analysis and optimization. Although our prototype implementation is for Java, this methodology can be applied to other programming languages. For example, most research on JavaScript code size reduction focuses on code compression, which may give attackers chances to obfuscate their malicious JavaScript [26]. Trimming redundant code can help reduce JavaScript code size without security concerns.

B. Usage and Application Scenarios

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 applications for which the JRE is customized. Some computing environments just run certain fixed Java applications. For example, ATMs, computers in offices or computer labs, registration kiosks, cloud instances, and in-vehicle entertainment systems are such cases. For those computing environments, a customized JRE would be a desired feature. First, JRE

[2]Note this is a rather conservative estimate. For example, Mockus, Fielding, and Herbsleb [25] found that the Apache Server has about 2.64 defects per KLOC.

TABLE VIII: 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

TABLE IX: JRed Performance

Benchmarks	avrora	batik	fop	h2	luindex	lusearch	pmd	sunflow	xalan
Transformation Time (s)	92	303	454	116	59	107	103	148	106

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. It is also a promising way to save resources on servers or clouds for both service providers and clients. Additionally, because Java programs are 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.

2) Code Trimming on Applications: JRed can analyze, customize, and transform the applications running on JVM. JRed is equally applicable to Dalvik by just replacing the parser of current implementation [27].

C. Limitations

First, regarding the customized JRE, our approach restricts applications to dynamically extend their functions without going through another round of customization process. A application that runs on a customized JRE cannot load plug-ins that needs some methods that have been deleted.

Second, our approach causes that whenever a new software update is released, we need to perform a new round of applications or JRE customization. If the code trimming happens at the developer side, this iterative updating and trimming process could be combined with Continuous Integration and Continuous Deployment (CI/CD) process or nightly building process. Nevertheless, from the client perspective, this is a limitation of our method.

Finally, JRed does not analyze native code. So JRed does not delete any native code. This policy may cause some false negatives. However, these false negative are not severe problems, since the number of the methods written in native code that are invoked via JNI interface is very small.

D. Future Work

We have not try our approach to a system with more than one application running. Applying our approach to a system with more than one application running does not cause additional challenges. The union of each single-application-customized JRE should contain all necessary classes and methods for running those applications. It is interesting to see how the customized JRE size and the number of vulnerabilities increase with the number of running applications in the system. We will conduct an experiment and present the result in the future.

VI. Related Work

There has been a substantial amount of research on program optimization. Here we discussed the most relevant work related to bloatware.

a) Code Size: On Java program size reduction, Pugh [7] proposes a method to efficiently pack class files into smaller jar files. Bradley et al. [8] introduce a new Java archive file format called Jazz, which has better compression ratio than the jar file format. Wagner et al. [28] present a method to mitigate the bloatware problem in "always connected" embedded devices. Specifically, they store the library code in a remote server. The classes that are needed will be downloaded on demand. In addition, by applying some more sophisticated analysis, some library code can be downloaded in advance before they are actually executed to improve the performance.

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 [29] and minification [30] tools. Souders [31] 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. [32] raise a methodology to detect obfuscated malicious JavaScript by using classification techniques. Oberlander [33] discusses a method to analyze C++ source code to collapse class hierarchies. Sweeney and Tip [34] developed an approach to remove unused data members in C++ applications. De Sutter et al. [35] apply aggressive whole-program optimization and extensive code reuse on C++ binary to avoid bloat brought by templates and inheritance.

b) Performance: On the performance side, Bu et al. [3] have pointed out that the negative impact on performance caused by bloatware is being amplified by today's big-data software usage nature. Xu [5] proposes a method to reuse those redundant objects. Xu [6] also presents a tool called CoCo to soundly and adaptively replace Java collections to remove bloat from Java software. Hosking et al. [36] mitigate the problem of bloatware by eliminating partial redundancy for access path expressions. Whitlock and Hosking [37] proposes a framework for persistence-enabled optimization of Java objects stores based on Bytecode-Level Optimizer and Analysis Tool (BLOAT). Xu et al. [38] presents a method to detect runtime bloat by applying abstract dynamic slicing technique. Nguyen and Xu [4] introduce Cachetor, a tool to detect cacheable data to remove bloat.

c) Call Graph Construction: Call graph construction has a profound impact on the precision and effectiveness of program analysis and optimization. Lhoták [14] proposes a flexible points-to analysis framework for Java. Grove et al. [39], [40] find that context-sensitive call graph construction method does not gain much improvement on the results compared with context-insensitive methods. Agrawal et al. [41] develop a demand-driven technique for call graph construction. Tip and Palsberg [42] 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.

d) Others: In addition, Jiang et al. [43], [44] studies feature-based software customization for Java program. In their approach, a feature, such as networking and data base access, is defined through API or methods and all related code is trimmed to get rid of the feature.

VII. Conclusion

In this paper, 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.

VIII. Acknowledgments

This research was supported in part by the Office of Naval Research (ONR) grants N00014-13-1-0175 and N00014-16-1-2265, and the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605.

References

[1] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *FSE/SDP workshop on Future of software engineering research*, 2010.

[2] M. Adam, "Java 7 applet 0day exploit," 2013, http://www.cs.bu.edu/~goldbe/teaching/HW55813/marc.pdf.

[3] Y. Bu, V. Borkar, G. Xu, and M. J. Carey, "A bloat-aware design for big data applications," in *Proc. ISMM '13*, 2013.

[4] K. Nguyen and G. Xu, "Cachetor: Detecting cacheable data to remove bloat," in *Proc. FSE'03*, 2013.

[5] G. Xu, "Finding reusable data structures," in *OOPSLA '12*, 2012.

[6] ——, "Coco: Sound and adaptive replacement of Java collections," in *Proc. ECOOP'13*, 2013.

[7] W. Pugh, "Compressing Java class files," in *Proc. PLDI '99*, 1999.

[8] Q. Bradley, R. N. Horspool, and J. Vitek, "JAZZ: An efficient compressed format for Java archive files," in *Conf. Centre for Adv. Studies on Collaborative Research*. IBM Press, 1998.

[9] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali, "Searching for build debt: Experiences managing technical debt at Google," in *Proc. the Third International Workshop on Managing Technical Debt*, 2012.

[10] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot—a Java bytecode optimization framework," in *Conf. Centre for Adv. Studies on Collaborative Research*. IBM Press, 1999.

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

[12] J. Whaley, "Joeq: A virtual machine and compiler infrastructure," in *Proc. IVME '03*, 2003.

[13] R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying Java bytecode for analyses and transformations," Sable Research Group, McGill University, Tech. Rep., 1998.

[14] O. Lhoták, "Spark: A flexible points-to analysis framework for Java," Master's thesis, McGill University, December 2002.

[15] J. Whaley and M. S. Lam, "Cloning-based context-sensitive pointer alias analysis using binary decision diagrams," in *PLDI'04*, 2004.

[16] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP'95*, 1995.

[17] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proc. of OOPSLA '96*, 1996.

[18] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06*, 2006, pp. 169–190.

[19] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Soft. Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[20] Project Analyzer v10.2, "Chidamber and Kemerer object-oriented metrics suite," Jun. 2014. [Online]. Available: http://www.aivosto.com/project/help/pm-oo-ck.html

[21] S. C. Misra and V. C. Bhavsar, "Relationships between selected software measures and latent bug-density: Guidelines for improving quality," in *Computational Science and Its Applications (ICCSA)*. Springer, 2003.

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

[23] D. E. Geer, "Complexity is the enemy," *IEEE S&P*, vol. 6, no. 6, 2008.

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

[25] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the Apache server," in *Proc. the 22nd ICSE*, 2000.

[26] P. Laskov and N. Šrndić, "Static detection of malicious JavaScript-bearing PDF documents," in *27th ACSAC*, 2011.

[27] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*. ACM, 2012, pp. 27–38.

[28] G. Wagner, A. Gal, and M. Franz, ""Slimming" a Java virtual machine by way of cold code removal and optimistic partial program loading," *Science of Computer Programming*, vol. 76, no. 11, 2011.

[29] D. Edwards, "Packer: A JavaScript compressor," 2007, http://dean.edwards.name/weblog/2007/04/packer3/.

[30] D. Crockford, "JSMin: The JavaScript minifier," 2003. [Online]. Available: http://www.crockford.com/javascript/jsmin.html

[31] S. Souders, "High-performance web sites," *Communications of the ACM*, vol. 51, no. 12, pp. 36–41, 2008.

[32] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious Javascript detection using classification techniques," in *Int'l Conf. on Malicious and Unwanted Software (MALWARE)*, 2009.

[33] J. Oberländer, "Applying source code transformation to collapse class hierarchies in C++," *Study Thesis, System Architecture Group, University of Karlsruhe, Germany*, 2003.

[34] P. F. Sweeney and F. Tip, "A study of dead data members in C++ applications," in *Proc. PLDI '98*. ACM, 1998.

[35] B. De Sutter, B. De Bus, and K. De Bosschere, "Sifting out the mud: Low level C++ code reuse," in *OOPSLA '02*, 2002.

[36] A. L. Hosking, N. Nystrom, D. Whitlock, Q. Cutts, and A. Diwan, "Partial redundancy elimination for access path expressions," *Software: Practice and Experience*, vol. 31, 2001.

[37] D. Whitlock and A. L. Hosking, "A framework for persistence-enabled optimization of Java object stores," in *Persistent Object Systems: Design, Implementation, and Use*, 2001.

[38] G. Xu, N. Mitchell, M. Arnold, A. Rountev, E. Schonberg, and G. Sevitsky, "Scalable runtime bloat detection using abstract dynamic slicing," *ACM TOSEM*, vol. 23, no. 3, p. 23, 2014.

[39] D. Grove, G. DeFouw, J. Dean, and C. Chambers, "Call graph construction in object-oriented languages," in *OOPSLA '97*, 1997.

[40] D. Grove and C. Chambers, "A framework for call graph construction algorithms," *TOPLAS*, vol. 23, no. 6, 2001.

[41] G. Agrawal, J. Li, and Q. Su, "Evaluating a demand driven technique for call graph construction," in *CC '02*, 2002.

[42] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proc. OOPSLA '00*, 2000.

[43] Y. Jiang, C. Zhang, D. Wu, and P. Liu, "A preliminary analysis and case study of feature-based software customization (extended abstract)," in *Proceedings of the 2015 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2015.

[44] ——, "Feature-based software customization: Preliminary analysis, formalization, and methods," in *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering, (HASE)*, 2016, pp. 122–131.