REDDROID: Android Application Redundancy Customization Based on Static Analysis

Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, Dinghao Wu
Pennsylvania State University
{yzj107, qub14, szw175, xvl5190, dwu}@ist.psu.edu

***Abstract*—Smartphone users are installing more and bigger apps. At the meanwhile, each app carries considerable amount of unused stuff, called *software bloat*, in its apk file. As a result, the resources of a smartphone, such as hard disk and network bandwidth, has become even more insufficient than ever before. Therefore, it is critical to investigate existing apps on the market and apps in development to identify the sources of software bloat and develop techniques and tools to remove the bloat. In this paper, we present a comprehensive study of software bloat in Android applications, and categorize them into two types, compile-time redundancy and install-time redundancy. In addition, we further propose a static analysis based approach to identifying and removing software bloat from Android applications. We implemented our approach in a prototype called REDDROID, and we evaluated REDDROID on thousands of Android applications collected from Google Play. Our experimental results not only validate the effectiveness of our approach, but also report the bloatware issue in real-world Android applications for the first time.**

***Index Terms*—software bloat; static analysis; Android; software customization;**

I. INTRODUCTION

Modern software development paradigms and practice help developers build more complex software products than ever before. On the other hand, it brings software bloat into software. By definition, software bloat is the "results 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 and complexity of use" [1], which is also known as "bloatware". To some extent, bloatware problem can be seen as a disease of affluence in the software world. Most software on contemporary markets suffers this problem more or less.

In this paper, we investigate software bloat that would lead to the rapid size increase of Android applications. Android 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. For example, large code bases generally contain more (exploitable) vulnerabilities, and they also provide a considerable amount of code components that can potentially enable code reusing attacks. Hence, it is critical to inspect the existing Android applications on the market and applications in development to identify and trim off software bloat.

Previous research on reporting and removing Android application bloat has either different scopes comparing with our work or incurs various limitations. Pugh [2] and Bradley et al. [3] proposed new algorithms and approaches to better compressing class files into a package. Xu [4] presented an approach to finding reusable data structures. Specifically, by identifying those data structures which share the same "shape" but have disjoint lifetimes, this approach can reuse just one data structure across all of these data structure reference sites. Thus, we can reduce the code size by removing other instances of the same data structure and reduce the performance penalty by avoiding frequent initializations. Lint [5] is a tool that can help Android developers remove those registered but unused resources that located in the "Res" directory in an Android project. It scans code and other resource files to detect if a resource file is ever referred by its ID. It cannot optimize those resources that do not have IDs. Some researchers [6], [7] raised the approaches to removing unused methods from Java bytecode. However, these approaches cannot be directly adopted to Android application due to the unique application organization and execution patterns of Android systems.

In this paper, we comprehensively inspect the software bloat issue in 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 more than 500 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.

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.
- 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 the paper is organized as follows. Section II provides our observations and insights regarding software bloat issue in the Android application domain. Section III describes the details of the our approach and how we implemented it. We present the evaluation results in Section IV. We discuss some interesting thoughts and future works in Section V. We then present related work in Section VI and conclude our paper in Section VII.

II. OBSERVATION AND INSIGHTS

A. 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.

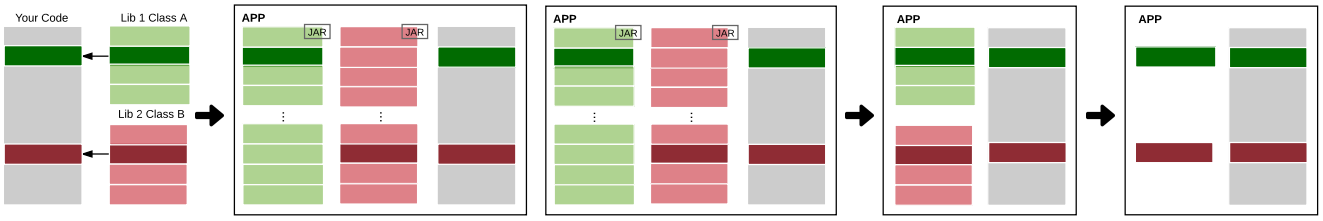
1) Compile-time Redundancy: Modern software engineering rarely implements a software product from scratch. Developers are relying on different kinds of libraries and frameworks to finish their jobs. Libraries usually are implemented for a more general purpose, instead of the requirements from a specific 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 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. Fig. 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.

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



(a) Modern Software Engineering Paradigm (b) Remove Compile-Time Redundancy in Libraries

Fig. 1: Motivation of RedDroid

as redundancy only after the installation platform information is given.

B. The Focus of This Paper

1) Compile-Time Redundancy from Java Libraries: Fig. 1b illustrates our focus of compile-time redundancy removing in this paper. 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 paper 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 paper.

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 paper, we focus on install-time redundancy caused by embedded ABIs and SDKs.

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 I 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.

TABLE I: 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

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 IV, 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 paper, 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. 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.

III. Design and Implementation

A. Architecture

Fig. 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 Fig. 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.

B. 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 [8], [9]. In our approach, we use a more intuitive method based on Class Hierarchy Analysis (CHA) [10] 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.[2] 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 [11].

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 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.

C. 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

[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] 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.

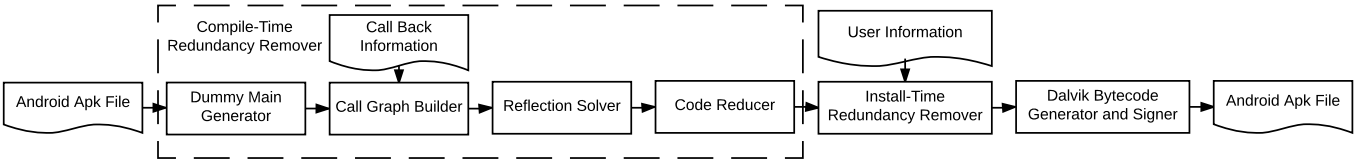


Fig. 2: REDDROID Architecture

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.

D. 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 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 (§ III-B) 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 [12] 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. 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.

E. 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

Listing 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   }
     ......
   }
```

Listing 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  }
```

call graph construction process (§III-B) 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 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 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 [13]. 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.

F. 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.

G. 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 [14] to generate dummy main method for analyzed Android applications. We use Soot [15] 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, 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.

IV. 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?

A. 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.

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. Fig. 3 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%.

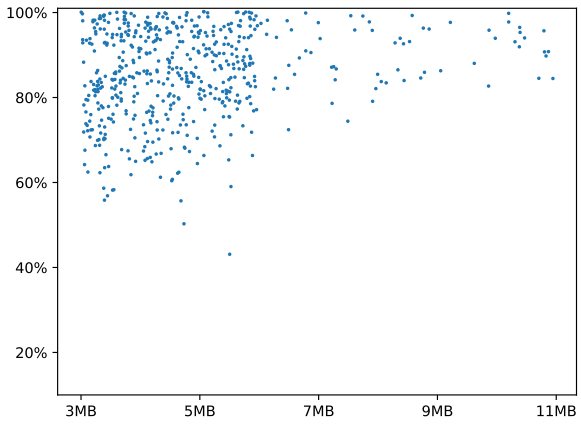


Fig. 3: Reduced Size Distributions

TABLE II: 10 Selected Android Application Code Size Before and After Unused Code Trimming Comparison

Benchmark	Original (Byte)	Reduced (Byte)	Reduced/Original (%)
IFTTT	7,416,304	6,026,077	81.25
Evernote Widget	1,201,311	1,131,289	94.17
Motorola Migrate	4,542,461	3,260,831	71.79
Baidu Browser	5,009,942	4,103,846	81.91
Yahoo Messenger	3,865,985	3,279,559	84.83
OpenTable	3,919,118	2,546,431	64.97
Flashlight	4,767,339	4,091,859	85.83
Marvel Comics	5,503,831	4,475,991	81.33
Papa Johns	5,206,893	3,753,546	72.09
Instagram	10,365,650	9,406,437	90.75

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 II. 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.

B. 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 [16], [17]. 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 [18]. RFC is the number of *response set* of a class. Response set, according to the paper of Chidamber and Kemerer [16], 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 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.

Fig. 4 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 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.

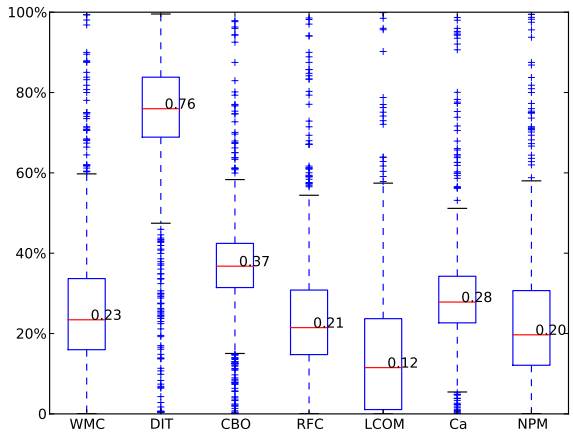


Fig. 4: Code Complexity Results

C. Reflection Call Sites

In this section, we present the results to answer research question Q3. Table III 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 III, 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.

D. Installation Time Redundancy

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

TABLE III: 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

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 IV 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.

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 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 V 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.

Fig. 5 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.

V. Discussion and Future Work

A. Issues caused by multiple Android API levels

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. The other issue we want to call out is that our evaluation was not conducted on the most recent Android application running environment. We will iterate our tool to synchronize with the pace of Android API update in the future.

B. Feature based Customization

Another future work is to perform feature based customization towards an application. Jiang et al. [19], [20] 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.

C. Relationship with Other Android Application Compaction Approach

There are a plenty of Android application code size reducing approach based on packing and compressing. More

TABLE IV: Size and Percentage of Installation Redundency in Wear Applications

Application Name	Original Size (Byte)	Reduced Size (Byte)	Reduced/Original (%)
Mobills: Budget Planner	15,725,488	14,023,717	89.18
AccuWeather	33,442,700	30,437,691	91.01
Wear Tip Calculator	4,070,793	2,011,661	49.42
App in the Air: Flight Tracker	14,826,753	10,847,048	73.16
Weather Live Free	32,659,948	31,086,561	95.18
ViewRanger - Trails and Maps	14,537,273	12,374,449	85.12
Keeper: Free Password Manager	15,905,430	11,531,725	72.50
Instant - Quantified Self	10,038,765	9,544,534	95.08
Google Play Music	17,961,307	15,678,474	87.29
Microsoft Outlook	30,348,958	26,821,576	88.38
Nest	41,391,891	35,063,279	84.71
Robinhood - Free Stock Trading	13,858,114	9,777,723	70.56
Strava Running and Cycling GPS	28,164,055	24,944,648	88.57
Viber Messenger	31,555,265	30,305,049	96.04
Wear Face Collection	27,476,581	17,154,939	62.43
Komoot — Cycling and Hiking Maps	12,150,268	9,573,000	78.79
WatchMaker Premium Watch Face	13,864,136	8,879,229	64.04
Average	N/A	N/A	80.67

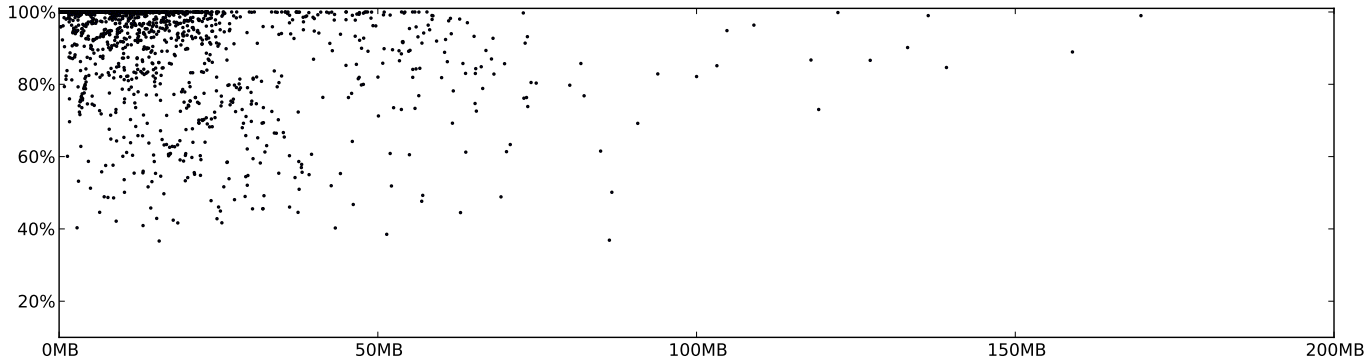


Fig. 5: ABI details

TABLE V: Proportions of applications that contain redundant ABIs by different size groups

Size (MB)	All Analyzed Apps	Modified Apps	Modified/All (%)
1	717	411	61.51
2	416	244	58.65
3	303	168	55.45
5	495	209	42.22
10	908	445	49.01
20	1,029	391	38.00
30	240	13	5.42
50	604	148	24.50
200	67	12	17.91
Total	4,779	2,041	42.71

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.

D. Reflections, obfuscation and Soundness

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 III 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 VI lists our strategies to each refection pattern. If a class, a method, or a field is referred statically, or referred

We have a case cannot handle, that is if some one initialize a class but not never use it. Or say it only uses its constructor, then that is unknownClass.unknownMethod case. If this case is rare, then my argument is valid. Otherwise, I cannot argue in this way. I need to do more experiments on this.

Software obfuscation may increase the cost of analysis on redundancy and reflections. To overcome this difficulty, some

software deobfuscation technologies [21], [22] can be applied. Detailed investigation on software deobfuscation is out of the scope of this paper.

VI. Related Work

A. Static Analysis on Android Applications and Frameworks

Cao et al. [12] 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 [14] 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. [23] 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 [24] 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 [25] and Stowaway [26] 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 [27] 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 [28] 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.

B. String and Reflection Analysis

An 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) [29] 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. [13] 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. [30] 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. Bodden et al. [31] presents another approach to taming reflections without depending on string analysis. They implemented their approach into a tool chain called TamiFlex. It first logs all reflection calls recorded during runtime. Then it uses direct method call to "materialize" those indirect reflective calls. Thus, those enriched Java programs can be soundly processed by static analyzers. In the case that a program needs to be transformed, TamiFlex can also help dynamically reinsert off-line transformed classes back to the running program.

C. Code and File Compaction Techniques

Pugh [2] proposes a better way to compress a group of Java class files to reduce the total size of a package, such as a jar file or an apk file. His optimization consists of three parts. First, each class file is reorganized into a more compact format. Second, he improved the compression algorithm. Third, he made some information is shared across class files in a package. Wagner et al. [32] 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. Several previous works [6], [7] proposed some approaches to reducing sizes of Java applications by removing its unused methods and classes. But their approaches cannot be applied to Android applications. Lint [5] 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. The optimization for these parts of resources will also depend on the string analysis techniques discussed in this paper, which can be done in our future work.

VII. Conclusion

In this paper, we 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

TABLE VI: reflection patterns and our strategies

Reflection Pattern	Strategy
unknownClass.knownMethod	Keep all methods that have the name "knownMethodName" in any class.
unknownClass.knownField	Keep all fields that have name the name "knownField" in any class.
knownClass.unknownMethod	Do not change this class.
knownClass.unknownField	Do not change this class.
unknownClass.unknownField (no such case in our data samples)	Delete methods only, do not delete classes.
unknownClass.unknownMethod (no such case in our data samples)	End alert to developers, our approach is unsound to this application.

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.

VIII. ACKNOWLEDGMENTS

This research was supported in part by the Office of Naval Research (ONR) grants N00014-13-1-0175, N00014-16-1-2265, N00014-16-1-2912, and N00014-17-1-2894, and the National Science Foundation (NSF) grant CNS-1652790.

REFERENCES

[1] J. McGrenere, ""bloat": The objective and subject dimensions," in *CHI '00 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '00. New York, NY, USA: ACM, 2000, pp. 337–338.

[2] W. Pugh, "Compressing Java class files," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: ACM, 1999, pp. 247–258.

[3] 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.

[4] G. Xu, "Finding reusable data structures," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12. New York, NY, USA: ACM, 2012, pp. 1017–1034.

[5] I. Google, "Improve your code with Lint," 2017. [Online]. Available: https://developer.android.com/studio/write/lint.html#overview

[6] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter, "Practical experience with an application extractor for Java," in *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '99. New York, NY, USA: ACM, 1999, pp. 292–305.

[7] Y. Jiang, D. Wu, and P. Liu, "JRed: Program customization and bloatware mitigation based on static analysis," in *Proceedings of the 40th IEEE Computer Society International Conference on Computers, Software and Applications (COMPSAC)*. IEEE, 2016, pp. 12–21.

[8] D. F. Bacon and P. F. Sweeney, "Fast static analysis of C++ virtual function calls," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '96, 1996.

[9] F. Tip and J. Palsberg, "Scalable propagation-based call graph construction algorithms," in *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '00. New York, NY, USA: ACM, 2000, pp. 281–293.

[10] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *European Conference on Object-Oriented Programming*. Springer, 1995, pp. 77–101.

[11] T. H. Cormen, *Introduction to Algorithms*. MIT Press, 2009.

[12] Y. Cao, Y. Fratantonio, A. Bianchi, M. Egele, C. Kruegel, G. Vigna, and Y. Chen, "EdgeMiner: Automatically detecting implicit control flow transitions through the Android framework." in *Network and Distributed System Security Symposium (NDSS)*, 2015.

[13] D. Li, Y. Lyu, M. Wan, and W. G. Halfond, "String analysis for Java and Android applications," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. ACM, 2015, pp. 661–672.

[14] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "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*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 259–269. [Online]. Available: http://doi.acm.org/10.1145/2594291.2594299

[15] 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.

[16] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, Jun. 1994.

[17] D. D. Spinellis, "ckjm chidamber and kemerer metrics software v1.9," 2005, http://www.spinellis.gr/sw/ckjm/.

[18] Virtual Machinery, "WMC, CBO, RFC, LCOM, DIT, NOC - the Chidamber and Kemerer metrics," 2017. [Online]. Available: http://www.virtualmachinery.com/sidebar3.htm

[19] 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.

[20] ——, "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.

[21] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 674–691.

[22] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 275–284.

[23] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*. ACM, 2012.

[24] B. Pan, "dex2jar," 2017. [Online]. Available: https://github.com/pxb1988/dex2jar

[25] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. ACM, 2012, pp. 217–228.

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

[27] R. W. Connor Tumbleson, "Apktool: A tool for reverse engineering android apk files," 2017. [Online]. Available: https://ibotpeaches.github.io/Apktool/

[28] "FernFlower," 2017. [Online]. Available: https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine

[29] A. S. Christensen, A. Møller, and M. I. Schwartzbach, "Precise analysis of string expressions," in *Proceedings of the 10th International Conference on Static Analysis*, ser. SAS'03, 2003, pp. 1–18.

[30] D. Shannon, S. Hajra, A. Lee, D. Zhan, and S. Khurshid, "Abstracting symbolic execution with string analysis," in *Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, ser. TAICPART-MUTATION '07. IEEE Computer Society, 2007, pp. 13–22.

[31] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, "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, 2011, pp. 241–250.

[32] 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.