Trimming Mobile Applications for Bandwidth-Challenged Networks in Developing Regions

Qinge Xie, Qingyuan Gong, Xinlei He®, Yang Chen®, Senior Member, IEEE, Xin Wang, Member, IEEE, Haitao Zheng, Fellow, IEEE, and Ben Y. Zhao, Member, IEEE

Abstract—Despite continuous efforts to build and update mobile network infrastructure, mobile devices in developing regions continue to be constrained by limited bandwidth. Unfortunately, this coincides with a period of unprecedented growth in the sizes of mobile applications. Thus it is becoming prohibitively expensive for users in developing regions to download and update mobile apps critical to their economic and educational development. Unchecked, these trends can further contribute to a large and growing global digital divide. Our goal is to better understand the source of this rapid growth in mobile app code size, whether it is reflective of new functionality, and identify steps that can be taken to make existing mobile apps more friendly to bandwidth constrained mobile networks. We hypothesize that much of this growth in mobile apps is due to poor resource/code management, and do not reflect proportional increases in functionality. Our hypothesis is partially validated by mini-programs, apps with extremely small footprints gaining popularity in Chinese mobile platforms. Here, we use functionally equivalent pairs of mini-programs and Android apps to identify potential sources of “bloat,” i.e., inefficient uses of code or resources that contribute to large package sizes. We analyze a large sample of popular Android apps and quantify instances of code and resource bloat. We develop techniques for automated code and resource trimming, and successfully validate them on a large set of Android apps. We hope our results will lead to continued efforts to streamline mobile apps, making them easier to access and maintain for users in developing regions.

Index Terms—Mobile applications, mini-programs, code bloat, lightweight applications, constrained network

1 INTRODUCTION

In the rapid development of 5G technology today, there still exists a persistent gap of mobile network access between developing and developed areas. In developing regions, bandwidth for mobile devices is still a very limited resource, where most users rely on cellular networks dominated by older infrastructure (2G or 2G+EDGE) [1], [2]. The result is overall poor quality of Internet access [3], with bandwidth of only hundreds of kbps [4], [5]. Despite efforts ranging from long-distance wireless links [6], [7], [8], localized cellular networks [2] to affordable commodity WiFi hotspots [9], growth in mobile bandwidth is still slow. Actual bandwidth available to users is often constrained by multiple factors including cost, last mile congestion, and limited access to backhaul links. Upgrading network infrastructure in developing regions still remains a challenge as the incremental costs of equipments and limited availability of technical knowledge [10], [11].

Unfortunately for users in developing regions, mobile applications (a.k.a., apps) worldwide are growing in size at an unprecedented pace, in part due to the growth of cheap or unlimited cellular plans. For example, traffic required to download the top 10 most installed U.S. iPhone apps (e.g., Facebook, Uber, YouTube) has grown by an order of magnitude from 164 MB in 2013 to about 1.9 GB in 2017 [12]. In the U.S., these “growing” app sizes mean that software updates now account for a big chunk of cellular bandwidth across the country [13], [14]. People in developing regions begin to rapidly switch from 2G to modern 4G phones for increased consumption of mobile applications. There are quite a number of 4G phone users in developing regions (17.9 percent in Indonesia and 22.5 percent in Philippines). However, it shows that 2G phones are still more shared and more active on the network due to the lack of network infrastructure in these regions [15], [16]. Unsurprisingly, studies already show that larger mobile applications lead to stability or usability problems on constrained networks [17], [18], [19], [20].

In concrete terms, this means that users in developing regions will find it difficult or impossible to access some of the most popular mobile apps critical to economic and educational development, despite studies that show tremendous impact from mobile apps on agriculture, health and education [21], [22], [23]. For example, Duolingo, a popular app for learning foreign languages, has an install package size of 20 MB, and as of May 2018, provides frequent updates with bug fixes that require a full download of the app each week. Khan Academy, the popular online education app, has an install package of 22 MB, and updates its software roughly once every 2 weeks. Other popular applications also have surprisingly large install packages. CodeSpark Academy is at 59 MB,
Facebook is at 65 MB, and Uber takes 61 MB to download. Even simple apps from American Airlines and McDonald’s require 83 MB and 43 MB to download respectively. Even more importantly, the majority of mobile apps are designed for bandwidth rich regions, and developers issue frequent updates. Recent studies show that 10 out of the 12 most popular apps issue updates very frequently (at least one update every two weeks) [24]. Most updates are similar or the same size as original app installs. For users in developing regions where a significant portion of their disposal income goes towards mobile bandwidth costs [25], this is a severe disincentive towards application usage.

One possible solution is to introduce lightweight applications. It is worth pointing out that lightweight apps encourage users to download when they need to use an app, and delete when not needed, which are usually not one-time things compared with regular applications. From the earliest web applications to the current Lite applications, various Internet companies are also working on developing lightweight apps and they are not even just for developing regions. Facebook Lite [26] is a new version of Facebook and was introduced in 2015, which uses less data and works well across all network conditions, even in an extremely slow 2G network. Facebook Lite has already hit over 1 billion downloads and their subsequent Messenger Lite [27] has been downloaded more than 500 million times. The emerging downloads indicate the huge demand of lightweight applications. However, existing lightweight apps require service providers to design and develop a specific version, which can not be adopted directly for Android apps and are not even just for developing regions. Facebook lite is a new version of Facebook and was introduced in 2015, which uses less data and works well across all network conditions, even in an extremely slow 2G network. Facebook Lite has already hit over 1 billion downloads and their subsequent Messenger Lite has been downloaded more than 500 million times. The emerging downloads indicate the huge demand of lightweight applications. However, existing lightweight apps require service providers to design and develop a specific version, which can not be adopted directly for Android applications as well as iOS applications on a large-scale.

At first glance, these trends seem to predict a widening digital divide where developing regions are losing access to critical parts of the mobile app ecosystem. But is the situation truly as dire as it seems? Intuitively, it seems unlikely that this staggering growth in the sizes of mobile apps is truly driven by growth in functionality. What factors other than functionality are contributing to this growth? Perhaps more importantly, how much of this growth is truly necessary for mobile applications as well as iOS applications on a large-scale?

Even if using sub-packages) [30]. Tight limits on app package size allow these platforms to adopt a load-on-demand approach to app discovery, where users can discover and run mini-programs “instantly” with negligible delay. By comparing mini-programs with their Android app counterparts detailedly and empirically, we identify the key source of “code bloat” of Android apps.

Until March 2020, there are 3.9 million available WeChat mini-programs and 450 million daily active users of WeChat mini-programs. The success of mini-programs also indicates the huge demand for such instant and lightweight apps. However, same as we mentioned above, mini-programs also require service providers to develop a specific version and can not be quickly applied to Android apps at large-scale. Starting with mini-programs, our ultimate goal is to propose a framework that can quickly convert existing normal mobile apps to a special type of instant/lightweight apps, i.e., apps that can be downloaded instantly due to small package sizes. Developers no longer need to design a specific lightweight version of the original app, thus the framework can be applied at large-scale.

In our study, we analyze a large selection of Android apps to understand the different software components and their contributions to overall app size. In the process, we identify multiple types of “code bloat” that can be trimmed from app packages without impact to functionality, including unreferenced third-party code and redundant resource files. We also develop generalizable techniques for trimming code bloat from existing mobile apps with the experimental validation of usability after being trimmed. Our results show that combined with compacting images and documentation, eliminating code bloat can significantly reduce the package sizes of many existing apps, making them much more usable in bandwidth-constrained environments such as developing regions.

We summarize our key contributions as follows:

- We collect a useful dataset of 200 pairs of mini-programs and Android apps, and perform code analysis on the dataset to understand potential sources of code bloat. To identify potential benefits of mini-program platforms like WeChat, we also implement a mini-program from scratch with identical functionality as an existing Android app, and analyze them to understand sources of app size discrepancy. As far as we know, we are the first to analyze mini-programs on a large-scale (also the first to analyze lightweight apps on a large-scale).

- By deeply decompiling WeChat, the dominant mobile application in China, we learn the library mechanism of mini-programs. And by comparing the library design mechanisms of mini-programs and Android library mechanism, we find the key difference likely arises from the lack of consistency in application libraries across Android devices. The finding inspires us to

1. Given the dominance of Android smart phones in developing regions [28], we focus exclusively on Android apps in this study.

2. WeChat is the dominant mobile messaging and social platform in China (1B+ users), and AliPay is the dominant mobile payment system in China.
make further optimization to Android apps. Meanwhile, our work can guide the following research on mini-programs, WeChat and lightweight applications.

- We perform a detailed analysis of 3,200 of the highest-ranked Android apps on Google Play, and confirm that linked libraries are a dominant factor in their overall app sizes. We use static analysis to identify unreferenced methods and classes, and carefully consider the preprocessing, decompiling, trimming, re-packing and validation processes to remove unreferenced code from these apps. We run a state exploration tool to estimate resource usage and find that significant pruning is possible for both code and resources. By integrating the analysis processes, we test the framework on the 3,200 Android apps with the validation of usability after being trimmed. The results show the effectiveness and correctness of our trimming process.

We note that Google is already spearheading developer initiatives for more lightweight mobile phone design in their “Building for Billions” initiative. Our effort is complementary, in that we focus more on the question of bloat for existing Android Apps, and how they could be retrofitted to perform better in constrained networks. We introduce the idea of a streamlined mobile app platform for bandwidth-constrained networks, and propose an automated process to reduce code bloat in existing Android apps. It packs commonly used APIs into a single library, allowing for reuse and minimizing per-app package size. This platform could be deployed by mobile phone OS developers like Google or Android app (platform) developers like Tencent. We hope this work leads to continued efforts on code trimming for existing mobile apps, and provides additional support for lightweight development efforts like Building for Billions.

Finally, while the net impact of our proposed code trimming mechanisms can vary across different apps, code trimming is impactful on nearly every app we studied, and often the largest and most popular mobile apps offer the most significant opportunities for code trimming. Our goal is to shed a light on an overlooked approach for reducing code bloat, with the hope that this and follow-up work will increase awareness to mobile developers and encourage them to utilize more efficient code reuse.

2 BACKGROUND AND RELATED WORK

Mini-Programs and Lightweight Apps. We begin by providing some background on the development of mini-programs by WeChat and Alipay. Mini-programs provide an extreme example of what is possible if code size was prioritized over all other concerns.

WeChat and Alipay are two leading Internet apps in China, in both users and influence. WeChat is a ubiquitous messaging platform with a mobile payment component that has become more accepted in China than cash. Alipay is a Chinese financial conglomerate that dominates the mobile payment market. While WeChat might be analogous to a union of Facebook and Venmo, Alipay might be a combination of PayPal and Amazon.

WeChat’s goal for its mini-programs is to introduce users to new apps in real time, often scanning a QR code to instantaneously install and run a mini-program [29]. Thus mini-programs have to be extremely small in size. The current limit is 2 MB for the entire installation package without using subpackages, including code, libraries and resource files. In reality, this development effort elevates WeChat to an app ecosystem capable of competing against Apple’s AppStore or Google Play, and WeChat encourages its users to bypass traditional app stores entirely. Since launching in January 2017, WeChat runs 580,000 mini-programs in 2018, compared with 500,000 mobile apps published by Apple’s AppStore between 2008 and 2012 [32].

The popularity of WeChat mini-programs led to a competing effort from AliPay, which launched their own “mini-app” platform in October 2017 [31]. AliPay’s platform also sets 2 MB as the limit for install package size (up to 4MB if using sub-packages), with an internal limit of 10 MB for storage. Given their similarity, we focus our analysis on WeChat mini-programs, and use mini-programs in the paper to refer to WeChat mini-programs.

Mini-programs and mini-apps also share properties with several other alternative lightweight app platforms. Web apps [33] are accessible via the mobile device’s web browser like Safari or Chrome. The apps run on the server and visual elements are sent to the mobile devices. Another type of apps, called hybrid apps [34], can be regarded as a special type between native mobile apps and web apps. Google announced Android instant apps [35] at Google I/O 2016. Users can tap to try an instant app without installing it first. An instant app only loads portions of the app on an on-demand basis. We note that our goal is to streamline mobile apps that run on the device, which differs from these platforms, since they both rely on network infrastructure [36]. There is a new type of apps, called Lite apps, which is first released by Facebook. Lite apps (e.g., Facebook Lite, Messenger Lite and Line Lite) keep the core functions of their original version and have smaller sizes, which are at the cost of having relatively fewer features and outdated interfaces compared with original version. However, for releasing a Lite version app, the service provider needs to design and develop a specific version, which can not be wide-scale adopted. There are only two apps in the top free 200 list [37] on Google Play have their Lite versions now, i.e., Facebook Lite and Messenger Lite.

Program Debloating. Code redundancy is a common phenomenon in software engineering, which not only leads to a waste of memory space/network traffic, but also increases the risks of being attacked [38], [39]. Therefore, program debloating, as a feasible solution, is being valued by researchers.

Techniques have been proposed for Java and Android applications debloating. RedDroid [40] removes the unused methods and classes in Android applications by statically construct an overapproximate call graph for the analyzed application. However, the authors failed to consider analyzing the library’s redundancy and the work lacked a large-scale verification of apps’ usability. JRed [41] trims unused code from both Java applications and Java Runtime Environment by static analysis.

There has been a lot of research aimed at program debloating in various applications. In response to software bloated in commodity software, RAZOR [42] performs code reduction for deployed binaries. With the booming development of machine learning, it is also starting to be used in program debloating. Heo et al. [43] proposed a system called Chisel to enable programmers to debloat programs effectively. They used a reinforcement learning-based approach to accelerate the search for the reduced program. Modern firmware also contains a mass of unnecessary code and modules. DECAF [44] was proposed for automatically trimming a wide class of commercial UEFI firmware. Azad et al. [45] focused on PHP applications and first analyzed the security benefits of debloating web applications.

Library/API Usage and Dependency. It is often the case that the client code uses an API in an inefficient way [46]. Developers are having difficulty balancing between the predictability of fixed version dependencies and the agility of flexible ones [47]. Several studies are dedicated to improving the use of third-party libraries. Kawrykow et al. [46] developed a tool to automatically detect redundant code and improve API usage in Java projects. Lammel et al. [48] designed an approach to deal with large-scale API-usage analysis of open-source Java projects. This technique helps with designing and defending mapping rules for API migration in terms of relevance and applicability. Wang et al. [49] proposed two qualified metrics for mining API-usage patterns, i.e., succinctness and coverage. They further proposed an approach called UP-Miner, for mining succinct and high-coverage usage patterns of API methods from source code. Huang et al. [50] implemented a method to update third-party libraries with drop-in replacements by their newer versions.

The third-party libraries also increase apps’ attack surface. Backes et al. [51] reported that two long-known security vulnerabilities in popular libraries were still present in the top apps. Derr et al. [52] showed that extensive use of third-party libraries could introduce critical bugs and security vulnerabilities, which put users’ privacy and sensitive data at risk.

## 3 MINI-PROGRAMS VERSUS MOBILE APPS

The proliferation of mini-programs in China demonstrates that for hundreds of thousands of mobile apps, their core functionality could be implemented in an extremely compact form. The question is, what are the tradeoffs necessary to obtain that compact implementation? What accounts for the difference in code size; is it more efficient code, or are there significant losses in functionality? Answering these questions will be a vital help for trimming mobile apps. In this section, we search for an answer by comparing mini-programs with their Android app counterparts, in both app content/features and code structure. We describe two detailed illustrative examples, and then present results of an empirical analysis of 200 of the most popular mini-programs and their Android app counterparts.

### 3.1 Overview

Mobile apps face few design constraints other than size limit. Google Play [53] limits Android apps by 100 MB (Android 2.3+) but allows two expansion files totalling up to 4 GB; iOS allows 4 GB for any submission to its App store [54]. A measurement study in 2017 shows that the average sizes of iOS and Android apps are 38 MB and 15 MB, respectively [12].

Mini-programs must abide by a number of restrictions, in their page depth (max number of hops necessary to reach any page), local storage, installation package size, and API usage, which we summarize in Table 1.² WeChat 6.6.0+ allows the use of subpackages in mini-programs, but still limits the size of a single package to 2 MB and the total sum of all packages to 8 MB [30]. To meet these constraints, developers often simplify app features and user interface (UI) elements in mini-program versions of their mobile app versions. Furthermore, mini-programs use JavaScript rather than Java (used by Android apps). We note that while code size across these two languages can vary (up to 20 percent [55]), any syntactical differences are unlikely to be meaningful. This is because APKs and mini-programs are stored under compression, and compression algorithms are likely to be more efficient on more verbose representations.

Using a common unpack tool [56] to parse mini-programs, we are able to compare the overall code composition of mini-programs and their Android counterparts (see Table 2). Implemented using JavaScript, a mini-program’s main program code resides in app-service.js, which is analogous to the Java Bytecode file in the Android APK (i.e., Android Dex file). There is a common setting file (app-config.json), analogous to AndroidManifest.xml file in the Android APK. Each page appears in the mini-program is registered in the common setting file, and there is a folder for every page to include its CSS configuration. The main page design code (HTML) is packaged in page-frame.html, analogous to Layout files in Android.

### 3.2 Code Package Composition

Next, we perform detailed comparisons between two mini-programs and their Android counterparts. One is a popular Android app whose developers implemented their own mini-program, and the other is an Android app for which we implemented a mini-program that precisely replicated its functionality.

Example 1: YouDao Dictionary. YouDao [57] is a very popular multi-language (7+ languages) dictionary app, which has an official mini-program version. As shown in Fig. 1, the mini-program includes the app logo and an input box

<table>
<thead>
<tr>
<th>Restricted Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page Depth</td>
<td>≤ 5</td>
</tr>
<tr>
<td>Package Size</td>
<td>≤ 2MB (8MB if using subpackage)</td>
</tr>
<tr>
<td>Local Storage</td>
<td>≤ 10MB per WeChat user</td>
</tr>
<tr>
<td>API Usage</td>
<td>WeChat offers certain APIs for development</td>
</tr>
</tbody>
</table>

---

² We omit game-related mini-programs in our table, which are granted 50 MB for local storage.
for translation, while the full app provides a more sophisticated UI and several extra features (camera translator, human translator, audio word search, courses and vocabulary book).

Table 3 lists the per-component code comparison between the mini-program and the Android app (we discuss these components in more detail later in Section 4). We see that the total footprint for the mini-program is 0.2 MB, compared with 47.2 MB for the Android app (219 times smaller). Across each analogous component, the mini-program version is smaller by at least a factor of 100! This is an example where all aspects of the Android app were compacted to generate its matching mini-program. While the core functionality remains, some non-core features were cut and the UI was simplified.

Example 2: Today Weather App. In an app like Youdao Dictionary, the developer made specific tradeoffs in choosing which areas to trim. We wanted to find a more controlled example where full functionality was preserved in the mini-program, so we could better understand the impact of compressing components unrelated to core features. The only way to ensure a true apple-to-apple comparison was to implement a mini-program ourselves, ensuring that the functionality of the Android app was preserved perfectly.

We found a reasonably sized Android app with simple program logic and single function, the Today Weather app, an Android app from the 360 app store with no matching mini-program, which provides city-wise weather conditions in China. Fig. 2 shows the Today Weather app and its mini-program version we developed.

To build a matching mini-program, we first decompiled the Android app from its Dalvik bytecode into Java bytecode using the well-known tool dex2jar [58]. Since the app’s program code, logic and function calls are all accessible, we replicated them completely with one minor exception. We also made sure that resource files like images were also identical to their Android counterparts. We tested our mini-program thoroughly to confirm that it offers the same interfaces, program logic, resources, and function calls.

Table 4 lists the package analysis of the two programs. While providing the same features, program logic, resources and network requests, the mini-program still achieves significant reduction in app size: 82.1 KB versus 527 KB (compressed) or 1452 KB (uncompressed) for the Android app, mapping to a factor of 6× to 18×.

A closer look shows that while the procedure code file in the Android app occupies 1.36 MB (93.80 percent of the package after decompressing the APK), the corresponding code file in the mini-program is only 8.38 KB, which is 160 times smaller. In fact, the procedure code files of the Android app is dominated by the Java library files, which takes up 95.59 percent of the code space. At least in this example, we find that we can significantly trim an Android app while preserving functionality and content (images and features). The key here is streamlining the procedure code, and more specifically, its Java library file.

Summary of Findings. These two examples show that developers can achieve drastic reductions by shrinking all components of mobile apps, including features and content (e.g., images). But even while preserving features and content, we can achieve significant savings by handling libraries more efficiently. We revisit this in more detail in Section 4.

### 3.3 Pair-Wise Package Analysis

Now that we have a high-level sense of potential areas for trimming mobile apps, we extend our comparison of Android apps and mini-programs to 200 of the most popular app pairs. The goal is to get a high level picture of how code, resources (e.g., images) and functionality compare across popular mobile apps, and how much room for trimming each category represents.

**Dataset.** We build a list of popular mini-programs from the monthly list of top-100 mini-programs published by aldwx.com. We include all mini-programs ever to appear on the WeChat mini-programs.

---

5. We omit the configuration file (i.e., AndroidManifest.xml of Android App and app-configuration.json of Mini-program) in the table because it covers a very tiny percentage compared with other files in the Youdao Dictionary example.


7. Due to security restrictions on mini-programs (HTTPS requests only), we had to change HTTP requests in the Android app to HTTPS requests in the mini-program. This minor change should have zero impact on the outcome of our analysis.

8. https://www.aldwx.com/ is a third-party statistics platform for WeChat mini-programs.
For each mini-program and Android app pair, we analyze several key metrics to better understand how the two differ in content, functionality and software package size.

- **Installation package size**: the size of installation package for mini-program (WXAPKG size) and the Android app (APK size).
- **Image size and number of images**: total size of all image files and total number of image files in the respective packages. Applications with richer features tend to have more images.
- **Page count**: a measure of number of features provided by the application. Since mini-programs register individual pages in their common settings file, i.e., app-config.json, we use this to count the pages for each mini-program as a measure of features in the program. Android apps register each activity in their AndroidManifest.xml file, and we use the number of activities as the measure of features for Android apps. In our experience, mini-program pages correlate roughly with Android activities.
- **Composition of installation package**: the proportion of individual components that make up the installation package, including images, procedure code and support files.

We use the reverse engineering tool Apktool [59] to decompress each target Android app and convert the Dalvik bytecode to Smali9 code for our analysis.

**Key Observations.** We plot key results from these metrics in Fig. 3. All graphs are sorted by descending order of the size of the Android app, ranked 1 (largest) to 200 (smallest). Thus all graphs are consistent on the x-axis. From these results, we make three main observations.

- **Mini-programs and their equivalent Android counterparts differ significantly in the size of installation package.** Mini-programs are 5-50 times smaller than their Android counterparts. Surprisingly, there seems to be little or no correlation between package sizes (Fig. 3a); several large Android apps (tens of MBs in size) have been translated to some of the smallest mini-programs (< 100 KB).
- **Not surprisingly, Android apps contain much more images and larger image sizes than their mini-program counterparts** (Fig. 3b). Bigger, more feature-rich Android apps lost more features in the translation to their mini-program counterparts.
- **For most Android apps, program code (procedure code and C++ library) dominates the installation package,** often taking 60-70 percent of the total footprint. Images only occupy 10-20 percent. Not surprisingly, images often dominate the much smaller installation packages of mini-programs.

Considered as a whole, our analysis of popular pairs of mini-programs / Android apps shows that current Android apps have used a variety of techniques to generate compact mini-program counterparts, and apps vary widely in how much potential code/resources are available for trimming.

### 4 Libraries and Bloat in Mobile Apps

Our results in the previous section identified Java Libraries as a potential culprit for the rapid growth of install packages in Android apps. Here, we take a closer look at how resource files, libraries and code make up the components of an Android app, by examining the code structure of a large range of popular Android apps.

**Android App Dataset.** We build a large set of popular Android apps and use it for our code analysis and code trimming experiments. We start with a ranking of top free Android apps from the popular app analytics platform App Annie.10 We choose 32 Android app categories11 from Google Play, and download the top ranked 100 apps in each category, forming a total dataset of 3,200 apps. Popular apps in our dataset include...
Duolingo, Khan Academy, Walmart, Uber and McDonald’s. It should be pointed out that the 32 categories we choose do not include game-related apps due to two reasons. First, as mentioned above, game-related mini-programs and regular mini-programs have different development restrictions. It will lead to an unfair comparison of Android apps and mini-programs, because there are different package size restrictions of Android game-related apps and Android regular apps. Second, the installation package format of most Android game-related apps is XAPK, which are different from normal apps (APK).

**TABLE 4**

| Package Analysis of Mini-program and Android App Versions of Today Weather |
|-----------------------------|-----------------------------|-----------------------------|-----------------------------|
| **Android App**             | **Mini-program**            |
| **Total installation package size** | 527 KB                      | 82.1 KB                     |
| **APK size**                | 1452.01 KB                  | WXAPKG size                 |
| **Size after decompressing** | 65.63 KB                     | 83.17 KB                     |
| **Images**                  | (4.52%)                     | (36.88%)                    |
| **Layout file**             | 6.80 KB                     | HTML/CSS                    |
| **Size after parsing**      | (0.47%)                     | 42.82 KB                     |
| **Images**                  | 30.17 KB                     | (52.32%)                    |
| **Assets Resource**         | 0 KB                        | **Lib**                     |
| **Assets**                  | (0%)                        | 0 KB                        |
| **C++ Library**             | (0%)                        | (0%)                        |
| **Procedure Code**          | **Android Dex file**        | **app-service.js**          |
| **Android Dex file**        | 1361.92 KB                   | 8.38 KB                     |
| **(93.80%)**                | (10.23%)                    |
| **Configuration File**      | **AndroidManifest.xml**     | **app-config.json**         |
| **AndroidManifest.xml**     | 2.42 KB                      | 0.47 KB                     |
| **(0.17%)**                 | (0.57%)                     |

Fig. 3. Detailed comparison between 200 pairs of mini-programs and their Android app counterparts. The pairs are ranked in descending order by the Android APK package size.

---

Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on December 12,2022 at 02:46:13 UTC from IEEE Xplore. Restrictions apply.
The Android Dalvik Executable specification limits the total number of methods that can be referenced within a single DEX file to 65,536. When the total number of methods exceed 65,536, there will be multiple DEX files of the application.

Fig. 5 quantifies the code sizes of Java libraries from different perspectives. Fig. 5a plots, for each of our 3,200 apps sorted by procedure code size, the absolute size of total procedure code (blue dot) and the absolute size of Java Libraries (red dot). For both, the code sizes are the sizes of the Smali files obtained after decompilation. Our key observation here is that for the overwhelming majority (96.7 percent) of apps, procedure code is dominated by Java libraries. This is further confirmed in Fig. 5b, which plots the CDF of the ratio between Java library code size and total procedure code size. For more than 55 percent of apps, Java library code accounts for more than 70 percent of total code size. Non-Java library code makes up the majority of procedural code in only 17 percent of apps. Fig. 5c shows that this dominance by Java libraries is consistent across app categories.

4.3 Library Management in Android versus Mini-Programs

While procedure code in Android apps is dominated by Java libraries, code in mini-programs is not dominated by their libraries. This can be directly attributed to how libraries are managed by WeChat mini-programs and their Android counterparts. Fig. 6 illustrates the two library management mechanisms, which we describe next.

Android Apps. Each Android APK includes both library code and app-specific code (code written by app developers). For example, Android.support.v4 is a library package and com.example is a custom code package. When generating an APK, all library code and custom code are packed into the same APK.

WeChat Mini-programs. After applying decompilation to WeChat, we find the library file used by WeChat mini-programs is the libappbrandcommon.so file in the lib directory of the WeChat app. When generating a mini-program, only custom code is packed into the mini-program, and not library files. In other words, a mini-program does not include library files in its installation package, but uses the library file included in the WeChat app.

This key difference likely arises from the lack of consistency in application libraries across Android devices. Apps packing their own library code improve robustness and increase the likelihood of the app running on different devices and Android versions. The price for this robustness is redundant library code packed into the APK files of each Android app. In contrast, the consistency offered by WeChat’s own mini-program platform means mini-programs can make stronger assumptions about library versioning, and a single common library can be shared across all mini-programs. This dramatically reduces the duplication of library code across apps or mini-programs.

5 TRIMMING CODE AND RESOURCES ON ANDROID APPS

Our early analysis of the Today Weather mini-program in Section 3 showed that some apps included large (and likely unused) Java libraries in their install packages. The sizes of these Java libraries could account for significant code size discrepancy between Android apps and mini-programs. Our additional hypothesis is that most Android apps only use a small subset of modules in Java libraries, but...
developers often import the entire library because manually identifying the code snippets for the target modules is labor-intensive. As a result, significant portions of the package code is actually unused by the app, i.e., “code bloat,” and should be trimmed.

In this section, we propose a systematic framework to trim Android apps, including identifying code bloat, removing it, and then repacking the app. Meanwhile, we also test the usability of apps after being trimmed. We can perform similar operations to identify resource bloat (e.g., unused images) and remove them from the app. Here, we describe our proposed process for trimming program code (Section 5.1) and resources (Section 5.2), and a process that integrates both. Later in Section 6, we evaluate the effectiveness of our proposed app trimming techniques and usability of the trimmed apps.

5.1 Trimming Code Bloat

Our trimming framework consists of four sequential steps: preprocessing, code decompilation, code bloat detection, and app repacking and validation. It takes an Android installation package as input, and outputs a repacked app with detected code bloat removed. More specifically, we first preprocess the input app, then unpack it using the dex2jar tool where Dalvik bytecode gets transformed to Java bytecode. We then leverage ProGuard [62], a Java class file shrinker, optimizer and obfuscating tool to identify and trim code bloat. Finally, we repack the app and validate that its functionality has not been disrupted by the trimming process. We now describe these steps in detail.

Preprocessing. The goal of preprocessing is to identify Android apps that cannot be decompiled and repacked due to built-in security mechanisms (e.g., encryption or code signatures) that prevent code modification or decompilation [60] (more discussion in Section 7). The preprocessing step tests Android apps by first re-signing the app (as a different developer from the original) and check if it can still run properly, and if successful, then decompiles the app using bytecode transformation, repacks the app, and then re-signs the app. If the re-signed app passes both tests, it is suitable for code trimming. Note that this limitation only applies because we are an untrusted third party. Google or an authorized third party could use authenticated tools to bypass an app’s protection mechanisms and enable code trimming.

Identifying and Removing Code Bloat. To identify code bloat, we first apply the dex2jar tool to convert the app’s Dalvik bytecode to Java bytecode. Here the conversion supports both apps with single-dex and multi-dex. For apps with multi-dex, we merge the Java bytecode per DEX file by their file paths and use a map file to record the file path that will later be used by the re-pack step.

Next we use the ProGuard to explore the app execution space, recursively searching for any class and class members that the app will actually use. Those not found in recursively search are treated as code bloat and removed from the app package. The search of the app execution space requires a seed or entry point. For this we use MainActivity, the actual entry point of the target Android app, accessible directly from the global configuration file (AndroidManifest.xml). To be conservative, we also do not trim any subclass of the Application, Activity, Service, BroadcastReceiver and ContentProvider classes, and instead use them as extra entry points. Furthermore, because ProGuard only targets Java, we do not trim the Enum class, Java Native Interface and construction methods as well as widgets used by the XML files. Finally, our current search implementation does not consider Java reflection and dynamically loaded code instantiated by the Java class loader, because ProGuard does not recognize them [63]. This means we could accidentally trim useful code, but we can identify any such mistake during the app validation step.

Fig. 5. Comparing the Java library code size with the total procedure code size. (a) absolute code size of the total procedure code (blue) and that of the Java libraries (red); (b) distribution of (Java library size/Procedure code size); and (c) average value of (Java Library size/Procedure size) across 32 app categories.
public class MainActivity extends AppCompatActivity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        int a = 1, b = 2, c = 3, d = sum(a, b);
        int sum(int num1, int num2){ return num1 + num2; }
        int sub(int num1, int num2){ return num1 - num2; }
    }
}

Listing. 1. The original MainActivity.

public class MainActivity extends AppCompatActivity {
    public MainActivity() {} 
    protected void onCreate(Bundle var1) {
        super.onCreate(var1);
        this.setContentView(2131296283);
        int var2 = this.sum(1, 2);
        int sum(int var1, int var2){ return var1 + var2; }
        int sub(int num1, int num2){ return num1 - num2; }
    }
}

Listing. 2. MainActivity bytecode after decompilation.

An Illustrative Example. Here is an example of how to identify and remove code bloat from an app. Listing 1 shows the Java code of the `MainActivity` class in a sample Android app, where “`onCreate()`” sets the layout file of the main page. Since `onCreate()` is the entrance to the program, the `sub` function will not be used after the program is executed, and should be trimmed. Listing 2 shows the `MainActivity` Java bytecode after decompilation, where variable names are replaced by their values, and any unused variables (i.e., `c`) are removed. Listing 3 shows the result after code trim, where the unused function `sub` is removed.

Re-Packing and App Validation. After re-packing the trimmed app, we need to validate if it still functions correctly. For this we follow previous works [60], [64] for app validation. In [60], the authors ran a Monkey [65] script on the tested apps to validate the effectiveness after they anonymized sensitive information in Android apps. Monkey tool is a means of automated testing for the Android platform provided by Google. In their work, Monkey performed a random exploration on the App UI for 2 minutes per app. In [64], their work included the process of binary rewriting and re-packaging apps, and they used PUMA [66] Android UI-automation framework to run each re-packed app for 3 minutes or until all UI states were explored. PUMA is a programmable UI automation framework for conducting dynamic analyses of mobile apps at scale, which incorporates a generic Monkey script and exposes an event driven programming abstraction.

Therefore, we run an automatic UI traversal script for 3 minutes based on the Appium [67] and Monkey scripts. Appium is an open source test automation framework for mobile apps. The script performs UI traversal as well as random exploration. This script will validate the functionality of the trimmed apps.

It is worth noting that another option for app validation is the proposed PUMA framework [66], which is also used by [64] for validation. Unfortunately, PUMA only supports up to Android 4.3, and a significant portion of apps (roughly one third of apps tested) fail to be installed on the Android 4.3 emulator due to SDK (Software Development Kit) limitations, which makes PUMA unsuitable for our final app validation process. 538 apps in our dataset passed the preprocessing test and can run on Android 4.3, therefore we also apply PUMA to these 538 apps for dual validation. Finally, there are 486 out of 538 apps (90.33 percent) function properly after being trimmed, which is consistent with the results of our main validation experiment in Section 6.

5.2 Trimming Resource Bloat

We also seek to detect and remove unnecessary bloat in resource files, i.e., both Res resources and Assets. For this we use static code analysis to identify unused resources in the app, from images to XML files. Specifically, we first use Apktool to decompile the target Android app for static code analysis, which converts the Dalvik bytecode to Smali code and parses the resource files. Parsing the resource files allows us to identify unused resource files.

Identifying Bloat in Res Resources. The `res` directory contains different file types like drawable, string, color, etc. We only identify bloat in drawable resources like images and XML files, because trimming other resource types requires modifying the XML file, and can potentially disrupt the decompilation and re-packaging process.

First, parsing resource files will produce a `public.xml` file in the folder `res/values`, which records every Res resource’s ID, name and type (drawable, string, attr, color, array, etc). As we mentioned in Section 4, after they are compiled, any Res resource is accessed through its resource ID. To identify all resources used by the app, we can just search for them in each Smali file.

Identifying Bloat in Assets. Assets usually store static resources like database files and videos, which are neither code nor configuration files. Thus resources in assets are not compiled when packed into an APK. Since asset resources are accessible by their absolute path in the code, we can identify them by traversing the absolute path of each asset resource in each Smali file. Resources not identified by this search are trimmed.

5.3 Putting Everything Together

Finally, we can integrate the code trimming process with the resource trimming process to build a fully automated app trimming framework for Android apps. The overall framework is shown in Fig. 7. It trims the assets, the res resources, and finally the procedure code in sequence.

Obfuscated Code. It is worth noting that the code of many commercial applications would be obfuscated when the apps are released. Code obfuscation is a commonly used method to prevent applications from being reverse-engineered, which usually makes the decompiled code unreadable. For the cases

13. Here we need to exclude any Res resources found in the R class Smali files, since those files include all Res resources.
where the code is obfuscated, we explain as follows that how our framework identifies specific resources, functions and variables.

- **Resources.** Taking List 2 as the example, it shows the `MainActivity` bytecode after decompilation. After being decompiled, the resources will be represented by different IDs, e.g., `R.layout.activity_main` in Listing 213129683, where `R.layout.activity_main` is a layout file to set the `Main Activity` interface. Actually, resources are represented by unique IDs during compilation. Thus, after decompiling the app, we can still find a resource by a unique ID, even though the resource name is not readable. Resources usually include layout files, images and XML files. Therefore, our resource trimming process can still locate resources by IDs even when code is obfuscated.

- **Functions and variables.** For the name of variables and functions, obfuscated code is often decompiled with some unreadable letters, e.g., `a`, `b` and `c`. Meanwhile, there will be lots of variables and functions with the same names in all the byte code. For example, there is a `sum` function in file `file_name1`, and a `multiple` function in file `file_name2`. `file_name1` is in the path `/dir1/` and `file_name2` is in the path `/dir2/`. After the app being obfuscated, we present one possible scenario as follows: `sum`→`a`, `multiple`→`a`, `file_name1`→`a`, `file_name2`→`a`, `dir1`→`a`, `dir2`→`b`. Therefore, we could use `/a` to locate the file `file_name1`, use `/b` to locate the file `file_name2`, use `/a` to locate the function `sum`, and use `/b` to locate the function `multiple`. Please note that the same file name will not appear in the same path and the same function or variable will not appear in the same file when code is obfuscated. Therefore, our code trimming process can still locate functions, variables and files by paths plus file names plus variable/function names.

Thus, even if the code is obfuscated and unreadable, we can still recursively search for any class and class member which apps actually use.

### 6 Evaluation

In this section, we evaluate the performance of our proposed app trimming framework. We consider two key metrics: effectiveness as measured by reduction in mobile app size, and correctness in terms of whether the trimmed app still functions properly.

**Experimental Configuration.** Our evaluation considers the 3,200 top Android apps described in Section 4. To experiment with this wide range of apps [68], we installed these 3,200 apps on an Android emulator (Samsung Galaxy S7, Android 8.0), and ran the emulator on two identical Ubuntu 16.04 machines with 6-core 3.60 GHz CPU and 100 GB memory. 76 out of the 3,200 apps fail to be installed on the emulator, while 204 apps fail to run properly after installation. We removed them from our experiments. In the end, our experiments used the 3,020 apps which successfully ran on the emulator.
remaining 2,920 apps to test the effectiveness and correctness of our trimming framework.

6.1 Effectiveness of App Trimming

Fig. 8 plots the CDF of the absolute app size reduction, the normalized app size reduction and the per component reduction normalized by the app size. From Figs. 8a and 8b, we see that for 40 percent of the apps, trimming the app can reduce the app size by at least 10 MB, or at least 52 percent. Here are some specific examples: Duolingo reduces from 19.87 MB to 12.07 MB, Khan Academy reduces from 21.94 MB to 16.48 MB, Uber reduces from 60.64 MB to 31 MB, and McDonald’s reduces from 42.5 MB to 15 MB. Fig. 8c further shows that trimming res resources (images) is highly effective, followed by trimming procedure code (Java library files).

These results confirm that our design can effectively and significantly reduce the sizes of Android apps by trimming code and resource bloat. Our trimming process is fully automated, allowing third-parties to easily generate lightweight mobile apps for developing regions without sacrificing basic functionality. For app developers, our framework helps to identify potential code and resource bloat for performance optimization.

6.2 Correctness of App Trimming

Large-Scale Evaluation. After re-packing the trimmed app, it is necessary to validate if it still functions correctly. We give a detailed introduction to the validation process in Section 5.1. Because of the vast number of applications that need to be tested, we use the above-mentioned UI-automation tools Monkey and PUMA for validation.

Among the 3,200 Android apps we tested, 2,920 apps passed our preprocessing steps and were deemed suitable for automated trimming. Of these, 2,617 apps (89.62 percent) passed validation and operated properly after being trimmed, which indicates the robustness of our framework. Meanwhile, as mentioned above, there are 538 apps in our dataset that can run on Android 4.3 and passed the preprocessing test, which can be validated by PUMA. Among the 538 apps, 486 apps (90.33 percent) function properly after being trimmed and re-packed. The dual authentication of PUMA experiments further confirms the robustness of our framework.

We observe that most apps have significant drops in their Java code sizes, with reductions ranging from 60-80 percent of their original sizes. More than 70 percent of apps saw a drop in redundant code of more than 5 MB, and a small number of apps saw a size reduction of more than 20 MB after being automatically trimmed. The average size of the 2,617 original apps is 22.49 MB, and after being trimmed, the average size drops down to 10.73 MB, which is about half of the original size. In other words, considering the average value, users are able to install twice as many applications on the mobile devices compared with the original situation. Similarly, the cost of network traffic is half of the original amount.

User Study. In addition to adopting above-mentioned PUMA, the widely used Android UI-automation framework, and Monkey, the Android standard UI automator for large-scale correctness validation, we further conducted a user study to demonstrate the correctness of our trimming process.

Referring to the user study performed in [69], we randomly selected 4 apps (Tips Imo beta call video chat, iFunny, ABC7 News, and Max Hurricane Tracker) from 4 different categories, i.e., education, entertainment, news & magazines, and weather. The apps were selected from the 2,617 apps which passed validation process in our large-scale evaluation. We recruited 11 people, including graduate students in Computer Science from a university and software engineers from a leading IT company in China, to participate in the user study. Each participant received a $10 bonus for the participant fee. We installed the 4 apps on an Android emulator (Samsung Galaxy S7, Android 8.0). Each participant was told to complete two tasks with each given app: (1) manually explore as many functionalities of the original app as possible in 5 minutes (longer than 71.56 seconds, the typical average app session [70], and 3 minutes, the automated test time), (2) manually explore as many functionalities of the trimmed app as possible in 5 minutes. After the tasks, participants were asked to rate their satisfactoriness of the trimmed apps.

Fig. 9 shows the absolute size of each component’s redundant content, i.e., procedure code, assets and res resources. The original sizes of apps vary from 2.90M to 22.32M, and the blue bars represent the absolute app sizes after being trimmed. The results show that there are different redundant components for different applications, e.g., a lot of resource redundancy in the app #1, and similar size of resource and code redundancy in the app #3. In addition, we tested the cases of single-dex (one app) and multi-dex (three apps, 2, 2, and 3 DEX files, respectively) at the same time. The multi-dex cases were able to help us validate the correctness of our DEX file merging and mapping processes. In our user study, compared with the original apps, the trimmed apps behaved in the same way it was designed. It is also worth mentioning that network errors and blank pages did occur in some tests, and we verified that it also occurred when testing the original apps. According to the participants’ feedbacks, the average satisfactoriness of the trimmed apps is 4.27 (the satisfactoriness is from 1 - 5). Also, they felt that there is no significant difference in performance, such as startup speed, and page switch. This may be due to that we tested the regular applications. As mentioned above,
because of the different development mechanisms compared with normal apps, we excluded game-related apps in both mini-programs and Android applications, which have larger sizes, are more complex, and have higher power consumption.

6.3 Validation of Mobile Sensors

Most mobile phone devices do not have external sensors, i.e., sensors outside the phone. Smart watches and smart bracelets which have come into fashion in recent years usually include special sensors like ECG monitors and temperature sensors, which are usually not built into mobile phones. However, apps related to smart watches and smart bracelets have different development processes and APIs compared with Android apps for phones, which called Android Wear apps and Android BLE (bluetooth low energy) apps. Therefore, external sensors and wearable devices are not considered in our work now.

Meanwhile, we want to verify whether apps can still call internal sensors of mobile phones correctly after using our framework. We choose three highly ranked sensor test apps in Google Play, i.e., Sensors Toolbox [71], Sensors Multitool [72] and Sensor Box for Android [73], which have all been downloaded more than 5 million times. Few Android devices have every type of sensors (especially barometers or thermometers) [74]. Thus we choose the sensors that most mobile phones have for testing. All three apps have the functions of testing Accelerometer Sensor, Light Sensor, Pressure Sensor and Proximity Sensor, and Sensors Toolbox can test GPS location. Therefore, we test Accelerometer Sensor, Light Sensor, Pressure Sensor, Proximity Sensor and GPS after using our framework. We also install apps on a Samsung Galaxy S7 emulator with Android 8.0. All three applications can be successfully installed and passed our preprocessing step. After our trimming and re-packing processes, all the sensors tested are working correctly, which verifies that our framework is effective for internal sensor-related applications.

7 DISCUSSION

Reducing Redundancy During App Development. Results of our study show that significant code and resource redundancy are widely present in today’s mobile apps. Removing or limiting them during app development is quite feasible using existing tools.

To reduce code redundancy, one potential tool is ProGuard [62], which has already been integrated into Android Studio [75], the official development environment for Android. Developers can easily edit ProGuard configurations to remove redundant code in their projects. However, our results show that code redundancy is still extensively existed in Android apps. Developers prefer to use ProGuard to intentionally obfuscate their code for preventing attacks rather than use ProGuard to remove redundant code. Moreover, even for obfuscation, developers report difficulties applying ProGuard for their own apps [76]. It is needed to be pointed out that ProGuard can only be used at the processes of development, and it is hard for users to trim the released applications. One other possible reason is that developers updating their apps over time might opt to save code belonging to deprecated features rather than removing them fully, since code removal might introduce troubles for the future revision, which require more effort to locate and fix.

Tools also exist for removing resource redundancy during app development. Android Lint is a code scanning tool provided by Android SDK and has been integrated into Android Studio. It helps developers identify and correct issues like unused resources during development. Similar to ProGuard, Android Lint can also only be used in the processes of application development. Our observations of high levels of resource redundancy likely indicates that few developers are using Lint. In roughly half of Android apps, more than 50 percent of asset resources are redundant, and it is even worse for Res resources: in roughly half of Android apps, more than 80 percent of Res resources are redundant. Since removing unused resources is less likely to produce complex failure modes, developers looking to trim bloat should start with resources.

The above discussion shows that there is a gap between developers and users, especially users with limited network resources, who anticipate mobile apps with less redundancy. However, although there are already existing tools for removing or limiting redundancy of apps during development, developers tend to keep the redundancy with the consideration of avoiding potential troubles and reducing workload. Meanwhile, the existing tools such as ProGuard and Lint, can only be used directly at the processes of development. There is no way for normal users to trim the applications because of the difficulty of decompilation and the fragileness of modifying released apps. Our framework can effectively and automatically trim existing released Android apps by trimming both redundant code and resources.

Devices Used in Developing Countries. Related research [1] shows that while mobile data has become more affordable across all regions, device affordability remains a significant barrier to mobile Internet access in developing regions, particularly for the poorest 20 percent of the population. Taking Sub-Saharan Africa as an example, according to consumers, a lack of digital skills and literacy followed by affordability are the two largest barriers to mobile Internet adoption [77].

Although in the developing regions, affordability is still a huge problem. However, we find that even for users from developing countries, there are quite a number of 4G phone users (17.9 percent in Indonesia and 22.5 percent in the Philippines) [15]. Still, for people who can afford 4G phones, the support of local networks is necessary.

In developing regions, service providers usually offer low-cost devices to meet the market demand. Low-cost smart phones used in developing regions will also have a high-resolution touch screen display, Wi-Fi connectivity, web browsing capabilities, and the ability to accept sophisticated applications. For example, Tecno Mobile has established a franchise retail network to provide low cost smart phones since 2010, and has been successful in Ghana, Cameroon, Nigeria, and other countries in Africa [78]. In 2016,
Orange and Google jointly launched an affordable digital communication package 3G Orange Rise 31 Special Edition, offering a high-quality smart phone at low-price for 14 African markets and Jordan with Android 6.0 [78]. Most Android apps can run on such low-cost devices but app installations are limited by the memory sizes of devices, e.g., above-mentioned 8GB memory.

To get a clearer picture of the mobile devices that are primarily used by users in the developing regions, we choose three main e-commerce sites in Africa, Kilimall [16] from Kenya, Jumia [17] from Nigeria and Takealot [18] from South Africa for study. We counted the top 10 mobile phones sold on the three websites, and the results are shown in Table 5. First, we find refurbished older iPhones are very popular on Kilimall (5 out of 10, from iPhone 4 to iPhone 6). Refurbished devices have a lower price than new devices and there is no significant difference in use, which is in line with the lack of affordability in developing regions. Second, besides refurbished iPhones, most popular devices use Android system. The price advantage of Android phones leads to the popularity in developing regions. Meanwhile, our work is for the Android system and it is very suitable for developing regions. Third, there is only one feature phone (non-smart phone) Nokia 105 (2019) in the top lists, which demonstrates that most popular devices used in developing regions are smart phones and they are able to run mobile applications. Besides Nokia 105 (2019), the smallest memory of other most popular devices in developing regions is 8GB and the smallest RAM is 512 MB, which is not a very low-end configuration, and we have tested that apps are trimmed by our framework can operate properly on the emulator with above configuration. Fourth, the memory sizes of most popular devices are less than 64 GB (24 out of 30), which indicates that the memory of popular mobile phones used in developing regions is still limited by low affordability. Besides, we use Samsung Galaxy S7 as the emulator in our experiments and Samsung Galaxy is also found to be on the top lists of the survey. Therefore, the analysis results indicate that our work can not only help users reduce network traffic, but also help users save phone memory.

In conclusion, our work can effectively trim mobile applications in the market and reduce the sizes of installation packages, helping users in developing countries better use mobile apps, and is suitable for most popular devices used in developing regions.

The Application Update Issue. The majority of mobile apps are designed for bandwidth-rich regions, and developers move to release updates more frequently. In order to reduce the data that needs to be transferred for app updates, Google Play has used a smart update strategy for Android apps updates since 2012 [79]. The idea of the smart update is basically to consider changes (also called deltas) to APK files are downloaded and merged with the existing files, which

<table>
<thead>
<tr>
<th>Sites Name</th>
<th>No.</th>
<th>Phone Name</th>
<th>Brand</th>
<th>Memory</th>
<th>System</th>
<th>Network Support</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jumia (Nigeria)</td>
<td>1</td>
<td>iPhone 11 Pro Max</td>
<td>Apple</td>
<td>6/16/64GB</td>
<td>iOS</td>
<td>4G/3G/4G LTE</td>
<td>4GB</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>iPhone X</td>
<td>Apple</td>
<td>6/16/64GB</td>
<td>iOS</td>
<td>4G/3G/4G LTE</td>
<td>3GB</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>Samsung Galaxy A20s</td>
<td>Samsung</td>
<td>32GB</td>
<td>Android</td>
<td>2G/3G/4G LTE</td>
<td>4GB</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Samsung Galaxy A71</td>
<td>Samsung</td>
<td>128GB</td>
<td>Android</td>
<td>4G LTE</td>
<td>8GB</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>Samsung Galaxy A31</td>
<td>Samsung</td>
<td>128GB</td>
<td>Android</td>
<td>4G LTE</td>
<td>4GB</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>Samsung Galaxy A10s</td>
<td>Samsung</td>
<td>32GB</td>
<td>Android</td>
<td>4G LTE</td>
<td>2GB</td>
</tr>
</tbody>
</table>

| Kilimall (Kenya) | 1 | Infinix ZERO 3 X552 | Infinix | 16GB | Android | 2G/3G/4G LTE | 3GB |
| | 2 | Refurbished iPhone 4S | Apple | 8/16GB | iOS | 2G/3G/4G LTE | 1GB |
| | 3 | Refurbished iPhone 4 | Apple | 8/16/32GB | iOS | 2G/3G/4G LTE | 512MB |
| | 4 | Refurbished iPhone 5 | Apple | 16GB | iOS | 2G/3G/4G LTE | 1GB |
| | 5 | TecnO F1 | TecnO | 8GB | Android | 2G/3G | 1GB |
| | 6 | Refurbished Samsung Galaxy S7 edge | Samsung | 32/64GB | Android | 2G/3G/4G LTE | 4GB |
| | 7 | Refurbished iPhone 5S | Apple | 16GB | iOS | 2G/3G/4G LTE | 1GB |
| | 8 | Samsung Galaxy A10s | Samsung | 32GB | Android | 2G/3G/4G LTE | 2GB |
| | 9 | Refurbished iPhone 6 | Apple | 64GB | iOS | 2G/3G/4G LTE | 1GB |
| | 10 | Global Google Version Refurbished Huawei P8 | Huawei | 16GB | Android | 4G LTE | 2GB |

| Takealot (South Africa) | 1 | Hisense U962 2019 | Hisense | 8GB | Android | 3G | 1GB |
| | 2 | Samsung Galaxy A30s | Samsung | 128GB | Android | 4G LTE | 4GB |
| | 3 | Xiaomi Redmi Note 8 | Xiaomi | 64GB | Android | 2G/3G/4G LTE | 4GB |
| | 4 | Huawei Y5 Lite | Huawei | 16GB | Android | 2G/3G/4G LTE | 1GB |
| | 5 | Samsung Galaxy A2 Core | Samsung | 8GB | Android | 2G/3G/4G LTE | 1GB |
| | 6 | Samsung Galaxy A10s | Samsung | 32GB | Android | 2G/3G/4G LTE | 2GB |
| | 7 | Nokia 105 (2019) Feature Phone | Nokia | 4MB | Symbian | 2G | 4MB |
| | 8 | Xiaomi Redmi 8A | Xiaomi | 32GB | Android | 2G/3G/4G | 2GB |
| | 9 | Samsung Galaxy A51 | Samsung | 128GB | Android | 2G/3G/4G LTE | 4GB |
| | 10 | Samsung Galaxy A50 | Samsung | 128GB | Android | 2G/3G/4G LTE | 4GB |
reduces the sizes of updates. In 2016, Google announced a new additional delta algorithm bsdiff [80] for reducing the size of app updates.

At present, research on program debloating rarely indeed concerned about update issues [39], [40], [42], [45]. Here, we offer insights into how our study works with app update issues. First, it is worth mentioning that normal apps do not update as often as expected. It has been observed that top apps update more frequently than normal apps (apps were randomly chosen from Google Play) [81]. There are only 5 percent of normal apps had four or more updates while 45 percent of top apps had four or more updates within a period of two years. Therefore, our framework is suitable for most apps. Second, apps are becoming more vulnerable and getting more permission hungry over updates. The majority of API calls related to dangerous permissions might be added to the code in the new versions [81], [82], [83], [84]. In those circumstances, we suggest that users should not necessarily always update, especially with constrained networks and outdated devices. When the necessary updates are needed, we are also able to provide new trimmed versions for downloading.

Third, besides bsdiff, there are other incremental (delta) update methods that were proposed [85], [86]. Most app stores in China also support delta update now, such as Mi App Store, 360 App Store and Wandoujia. It is entirely feasible in our framework that comparing the new trimmed version of the APK with the old trimmed version of the APK to generate the patch package for delta update. This step can be put on the server so that users are also able to use delta updates. We take delta update as future work to further help users save network traffic.

Lightweight App Platforms (for Developing Regions). We show that WeChat mini-programs exhibit significant size savings when compared with Android apps. Part of this comes from mini-programs’ shared library access, as discussed in Section 4. This suggests that a platform for lightweight (mobile) applications might benefit mobile users in developing regions, using WeChat’s platform as a reference. Placing commonly used code into a shared, consistent app library would greatly reduce code redundancy for mobile apps.

Limitations. Finally, we discuss the limitations of our study. First, app decompilation and repacking are known to be fragile [60]. Errors can creep into the system during the use of reverse-engineering tools like dex2jar, Apktool and enjarify [87]. For instance, a unknown opcode exception was reported when we used dex2jar to translate Dalvik bytecode to Java bytecode. This is because that most reverse-engineering tools read bytecode linearly and the parse process fails when encountering an invalid bytecode. Thus developers can prevent third-party code trimming by intentionally or accidentally inserting invalid bytecode into the Android DEX file. Similarly, steps like parsing procedure codes can be disrupted or slowed using unexpected inputs in procedure code. Finally, developers can always use encryption or code signatures to prevent or detect alterations to their code.

8 Conclusion and Future Work

In this paper, we take an empirical approach to analyze sources of bloat in today’s mobile applications. Using WeChat mini-programs as a basis for comparison, we were able to identify a number of potential causes for the rapid growth in mobile app package sizes. This, in turn, allowed us to identify techniques to significantly reduce sizes of existing Android applications by modifying and trimming unnecessary code and resources. Our framework can quickly and automatically convert regular mobile apps to lightweight apps, which can be applied at large-scale.

Developers no longer need to design and implement specific lightweight versions of original applications.

While our techniques have demonstrated significant success in our tests, we believe they represent only initial steps by which developers can support mobile users in developing regions. For example, our work helps to address the challenge of downloading and updating apps in bandwidth-constrained networks. But many mobile apps today make strong assumptions about the availability of network bandwidth, and either fail to operate fully under constrained conditions, or aggressively consume bandwidth to the detriment (and high-costs) of their users. We hope our work and others will lead to treatment of bandwidth-constrained networks as a first class consideration, along with development of tools and platforms that more easily integrate support for low-bandwidth networks into a wide-range of mobile applications.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grants 62072115, 71731004, 61602122, and 61971145 and in part by China Postdoctoral Science Foundation under Grant 2021M690667.

References


Authorized licensed use limited to: FUDAN UNIVERSITY. Downloaded on December 12,2022 at 02:46:13 UTC from IEEE Xplore. Restrictions apply.
et al.


IEEE Trans. Mobile Comput.,

Proc. Annu.


interests include network security, user behavior analysis, and computational social systems.

Qingyuan Gong received the PhD degree in computer science from Fudan University in 2020. She was a visiting student with the University of Göttingen in 2015 and 2019, respectively, and the University of Chicago in 2018. She is currently a postdoctoral at Fudan University. She has authored or coauthored referred papers in the IEEE Communications Magazine, ACM Transactions on the Web, IEEE Transactions on Computational Social Systems, Springer World Wide Web Journal, ACM CIKM, and ICPP. Her research interests include mobile systems, machine learning, and data mining.

Xinlei He received the BS degree from the School of Computer Science, Fudan University, in 2017. He is currently working toward the graduate degree at the School of Computer Science, Fudan University. He was a visiting student with the University of Göttingen and the Southern University of Science and Technology in 2018. Since 2014, he has been a research assistant with Mobile Systems and Networking Group. His research interests include mobile systems, machine learning, data mining, and user behavior analysis and modeling.

Yang Chen (Senior Member, IEEE) received the BS and PhD degrees from the Department of Electronic Engineering, Tsinghua University, in 2004 and 2009, respectively. He is currently an associate professor at the School of Computer Science, Fudan University, where he leads Mobile Systems and Networking Group, Fudan. From April 2011 to September 2014, he was a postdoctoral associate with the Department of Computer Science, Duke University, USA, where he was a research assistant and the deputy head of Computer Networks Group, Institute of Computer Science, University of Göttingen, Germany. He was a visiting student with Stanford University in 2007 and with Microsoft Research Asia from 2006 to 2008. He was a Nokia visiting professor with Aalto University in 2019. His research interests include online social networks, Internet architecture, and mobile computing. He is currently the associate editor-in-chief of Journal of Social Computing, an associate editor for IEEE Access, and an editorial board member of the Transactions on Emerging Telecommunications Technologies. He is an OC or TPC member for many international conferences, including the SOSP SIGCOMM, WWW, IJCAI, AAAI, ECAI, DASFAA, IWOoS, ICCCN, GLOBECOM, and ICC.
Xin Wang (Member, IEEE) received the BS degree in information theory from Xidian University, China, and the MS degree in communication and electronic systems from Xidian University in 1994 and 1997, respectively, and the PhD degree in computer science from Shizuoka University, Japan, in 2002. He is currently a professor at Fudan University, Shanghai, China. His research interests include quality of network service, next-generation network architecture, mobile Internet, and network coding. He is a distinguished member of the CCF.

Haitao Zheng (Fellow, IEEE) received the PhD degree from the University of Maryland, College Park, in 1999. After spending six years as a researcher with industry labs (Bell-Labs, Crawford Hill, NJ, and Microsoft Research Asia), she joined UC Santa Barbara faculty in 2005 and became an associate and a full professor in 2009 and 2013, respectively. In July 2017, she joined the University of Chicago as Neubauer professor in computer science. She has been actively working on security and privacy issues for both machine learning models and mobile computing systems. Her research work has been frequently featured by media outlets, such as New York Times, Boston Globe, LA Times, MIT Technology Review, and Computer World. Her research work has been frequently featured by media outlets, such as New York Times, Boston Globe, LA Times, MIT Technology Review, and Computer World. She has been actively working on security and privacy issues for both machine learning models and mobile computing systems. Her research work has been frequently featured by media outlets, such as New York Times, Boston Globe, LA Times, MIT Technology Review, and Computer World.

Ben Y. Zhao (Member, IEEE) received the BS degree from Yale in 1997 and the PhD degree from Berkeley in 2004. He is currently the Neubauer professor of computer science at the University of Chicago and an ACM distinguished scientist. His work has been covered by media outlets, such as Scientific American, New York Times, Boston Globe, LA Times, MIT Technology Review, and Slashdot. He has authored or coauthored more than 160 publications in his research interests. His research interests include security and privacy, networked systems, wireless networks, data-mining, and HCI (H-index 70). He was the TPC co-chair for the World Wide Web Conference (WWW 2016) and the ACM Internet Measurement Conference (IMC 2018), and is currently the general co-chair for the ACM HotNets 2020. He was the recipient of NSF CAREER Award, MIT Technology Review’s TR-35 Award (Young Innovators Under 35), Computer World Magazine’s Top 40 Tech Innovators Award, Google Faculty Award, and IEEE ITC Early Career Award.

For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.