SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

April 2023, Vol. 66 142103:1–142103:19
https://doi.org/10.1007/s11432-021-3528-7

LegoDroid: flexible Android app decomposition and instant installation

Yi LIU[1,2], Yun MA[3], Xusheng XIAO[4], Tao XIE[1,2] & Xuanzhe LIU[1,2*]

[1]*School of Computer Science, Peking University, Beijing* 100871*, China;*
[2]*Key Laboratory of High Confidence Software Technologies* (*Peking University*)*, Ministry of Education, Beijing* 100871*, China;*
[3]*Institute for Artificial Intelligence, Peking University, Beijing* 100871*, China;*
[4]*Department of Computer and Data Sciences, Case Western Reserve University, Cleveland* OH 44106*, USA*

Received 8 March 2021/Revised 18 November 2021/Accepted 14 January 2022/Published online 27 March 2023

Abstract Current mobile applications (apps) have become increasingly complicated with increasing features that are represented on the graphical user interface associated with application programming interfaces (APIs) to access backend functionality and data. Meanwhile, apps suffer from the "software bloat" in volume. Some app features may be redundant, with respect to those features (from other apps) that the users already desirably and frequently use. However, the current app release model forces users to download and install a full-size installation package rather than optionally choosing only their desired features. Large-size apps can not only increase the local resource consumption, such as CPU, memory, and energy, but also inevitably compromise the user experience, such as the slow load time in the app. In this article, we first conduct an empirical study to characterize the app feature usage when users interact with Android apps, and surprisingly find that users access only a very small subset of app features. Based on these findings, we design a new approach named LegoDroid, which automatically decomposes an Android app for flexible loading and installation, while preserving the expected functionality with a fast and instant app load. With such a method, a slimmer bundle will be downloaded and host the target APIs inside the original app to satisfy users' requirements. We implement a system for LegoDroid and evaluate it with 1000 real-world Android apps. Compared to the original full-size apps, apps optimized by LegoDroid can substantially improve the load time by reducing the base bundle and feature bundles by 13.06% and 10.93%, respectively, along with the app-package installation size by 44.17%. In addition, we also demonstrate that LegoDroid is quite practical with evolving versions, as it can produce substantial reusable code to alleviate the developers' efforts when releasing new app versions.

Keywords performance, software bloat, instant installation, mobile applications, program analysis

Citation Liu Y, Ma Y, Xiao X S, et al. LegoDroid: flexible Android app decomposition and instant installation. Sci China Inf Sci, 2023, 66(4): 142103, https://doi.org/10.1007/s11432-021-3528-7

1 Introduction

The burst and great success of mobile applications (a.k.a., apps) has significantly changed our work and life. The current app-release model requires users to download apps' installation packages from app stores, e.g., IPA-format files of iOS apps from the Apple App Store, and APK-format files of Android apps from Google Play, and install them on users' devices. Not surprisingly, with more and more features included, the volume of installation package keeps increasing as well, known as the "software bloat" problem [1][1)]. Although users typically access only certain parts of the features of the downloaded app, the current app-release model enforces the users to download a full-size app.

However, the app-release model can inevitably increase resource consumption and slow down the load time. Therefore, it can potentially compromise user experiences and even reduce the users' enthusiasm to try out new apps or new versions. A recent survey over Google Play[2)] reports that the average APK size

* Corresponding author (email: xzl@pku.edu.cn)
1) Wikipedia. Software Bloat. 2022. https://en.wikipedia.org/wiki/Software_bloat.
2) https://medium.com/googleplaydev/shrinking-apks-growing-installs-5d3fcba23ce2.

has quintupled, and a negative correlation exists between the APK size and installation conversion rate for apps: every 6 MB increase of an APK's size can result in a 1% decrease in the installation conversion rate. In particular, it is worth mentioning that the "actual" storage space for installing an app is much larger than the size of its APK file, as the APK file needs to be unzipped and installed on the device. Indeed, there can be various reasons that users tend to dislike and even uninstall an app [2]. However, the side effects of increasing the package size, such as the additional consumption of local resources (e.g., storage, RAM, or battery), the UI complexity, and the resulting compromised user experiences cannot be simply neglected. Especially, a substantial number of resource-constrained devices, such as low-end smartphones (e.g., in some developing countries) and IoT devices, are still more sensitive to the size of the installation packages.

Although numerous apps are probably "abandoned" due to the increasing size and complexity, whether these apps are "totally" useless to users should be arguable. Some features of these apps can still be useful with respect to some "opportunistic" [3] or "situational" [4] requirements. For example, a tourist may want to search in various e-car-rental apps for renting the same car model and compare the prices. In such a case, she only needs the search feature in these apps, but the current app-release model enforces her to download and install these apps' complete installation packages. Therefore, she is likely to give up downloading these apps, and the developers' opportunity of winning this user is compromised.

In practice, unsurprising, most of the users require only a set of features rather than regularly accessing every single feature of popular apps, especially given that current apps contain increasingly many features [5], which results in the large size of installation packages. For example, one of the most popular apps in China, `WeChat`, now contains more than 1000 "pages", which are technically "activities"[3] along with thousands of features. Moreover, the size of its installation package is more than 160 MB. The size of an app's installation package is proportional to the amount of the required runtime resource for executing the app [6]. Subsequently, increased runtime overhead can potentially prevent the adoption of the app.

To mitigate the problem, one possible option is that the app can be designed in a way where the users can access the app at the page level rather than downloading the whole package, just like browsing web pages. Some attempts [7][4][5][6] have been made. For example, Google Instant apps provide a "lightweight" installation style that allows Android users to click on an in-app hyperlink to access the desired page of an app, rather than downloading the whole app. The `WeChat` app has been equipped with its mini program platform, which works in a similar way. The mini programs are essentially implemented as Web apps with some customized features that are thus accessed via hyperlinks from the app. Recently, Google provided a new app-serving model with the Android app bundle that allows users to install an app with features that the majority of target users use and install dynamic features on demand.

However, existing limitations prevent the wide adoption of these solutions. For example, Instant apps and Android app bundles need substantial development efforts[7] for every single feature and can be accessed only on devices that have installed Google Play Services. `WeChat`'s mini programs suffer from worse user experience than native apps. Meanwhile, refactoring existing apps to support these solutions requires tremendous efforts given the large number of existing apps and their iterative and incremental style in updating the code base from their previous versions.

Addressing the user experience compromised by app bloating is urgent and requires rethinking the current app release and deployment model. In this article, we propose a novel approach, named LegoDroid, to facilitate developers to "debloat" an app by decomposing, re-deploying, and releasing an app. Intuitively, LegoDroid does not deem to replace the current app release model, but it equips an app with the flexible installability so as to help developers gain more potential app usage. We design LegoDroid to meet the following three requirements. (1) Robust decomposition. LegoDroid decomposes an Android app into a core "launch bundle" that assures the correct launch of an app with basic functionalities and a set of "feature bundles" that can be correctly loaded on-demand to access the corresponding services and APIs. (2) User-friendly. LegoDroid should not compromise user experiences in terms of both per-

3) An activity is the basic program unit on the Android OS. Supposing that an app can be analogous to a website, we can consider an activity to be similar to a Web page of this site. If not particularly specified, we use the term "page" and "activity" interchangeably in the rest of this article.

4) Google. Google Instant App. 2022. https://developer.android.com/topic/instant-apps/index.html.

5) WeChat. The mini programs provided by WeChat. 2016. http://tencent.com/en-us/articles/15000551479986174.pdf.

6) Android App Bundle. 2022. https://developer.android.com/platform/technology/app-bundle/.

7) Blog V. Creating an Instant app. 2017. https://medium.com/ vimeo-engineering-blog/vimeo-android-instant-apps-2f8b1e94760c.

formance and interactions. The dynamic loading of features should be agnostic to the users, as if they are interacting with a full-size app. (3) Developer-friendly. LegoDroid should provide an automated way that can correctly detect boundaries of independent features without incurring re-development efforts from the developers.

This article makes the following main contributions.

• We conduct an extensive empirical study and demonstrate that page-level flexible app installation is strongly required.

• We design an approach to properly decomposing an Android app to achieve on-demand and flexible installability while preserving the expected functionality and satisfactory user experiences.

• We evaluate our approach by decomposing 1000 real-world Android apps and testing the performance of the decomposed apps along with on-demand installation. The evaluation results show that our approach can save 44.17% of initial download size, reduce 13.06% and 10.93% of loading time, and reduce 8.83% and 9.14% of memory usage on launching launch bundles and feature bundles in the median case, respectively. In addition, over 80% of the LegoDroid-generated code and resources can be reused when an app evolves. These results indicate that our approach can help improve user experiences compared to the full-size app under the same context, and is also practical in the current app ecosystem.

The rest of the article is organized as follows. Section 2 describes the motivation and Section 3 makes an in-depth analysis of design principles along with challenges. Section 4 presents the approach overview. Section 5 introduces the implementation of our LegoDroid approach. Section 6 details the evaluations of LegoDroid. Section 7 presents discussion about our approach. Section 8 reviews and compares LegoDroid with related work. Section 9 concludes this article.

2 A motivating study

To understand how the functionalities in an app are used at a fine granularity, i.e., at the app page level rather than at the app level, we first conduct an empirical study to explore the evolving complexity of app functionalities and app usage patterns in Android apps. The empirical study aims to answer the following three research questions.

• RQ1: How does the functionality complexity evolve as apps upgrade? Due to the highly competitive app store ecosystem, developers tend to upgrade and release new app versions quite frequently. Consequently, apps tend to become increasingly complex with more functionalities to attract users. By answering this question, we can understand the evolving trend of apps' complexities in the Android ecosystem.

• RQ2: How many features in an app are used by different users? Although mobile apps tend to provide more features to retain their users, usually only certain features of an app rather than all of them are used, which conforms to the Pareto distribution[8)]. By answering this question, we can understand whether all provided features of an app are really necessary for users.

• RQ3: Do different users focus on using different features in an app? Developers satisfy the requirements of different users by providing more features in a single app, but different users may have their own preferences. By answering this question, we can understand the diversity of app usage patterns.

2.1 Increasing app complexity

To answer RQ1, i.e., how does the functionality complexity evolve as apps upgrade, we choose the top-50 most downloaded apps in a leading Android app store in China, namely Wandoujia[9)]. For simplicity, we download each app's APK file for the first version that can be publicly found and the latest version. We use the size of the APK files and the number of activities as two metrics to measure an app's complexity for the following reasons. (1) Larger APK sizes usually imply more features. (2) More activities, also referred to as pages, indicate more unique interactive functionalities.

Figure 1 shows the distribution of the APK size (a.k.a the download size) and the number of activities for these 50 apps (including the first and the latest versions). Comparing the first and the latest versions, the APK size and the number of activities have dramatically increased. With the evolution of versions, apps have increased by 3.96X to 70.85X in APK sizes and 2.53X to 156X in the number of activities.

8) Pareto V. Pareto Principle. 1896. https://en.wikipedia.org/wiki/Pareto_principle.

9) As of Feb. 2022, Wandoujia has more than 500 million users and 3.2 million apps.

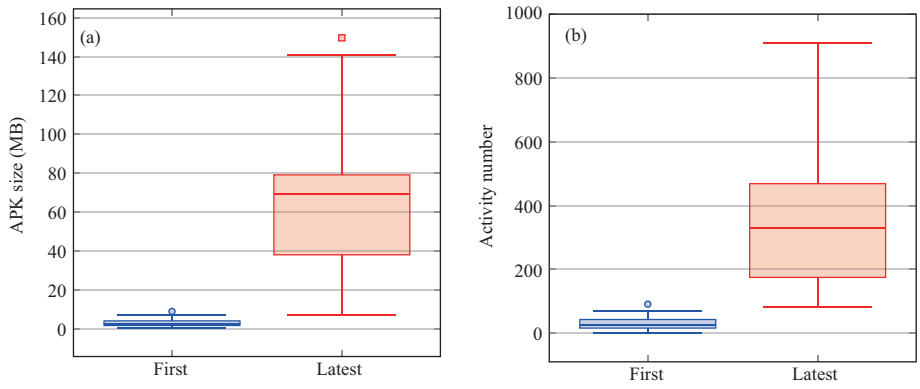

Figure 1 (Color online) Change in (a) the APK size and (b) #activities of the first version and the latest version from top 50 apps in the Wandoujia market.

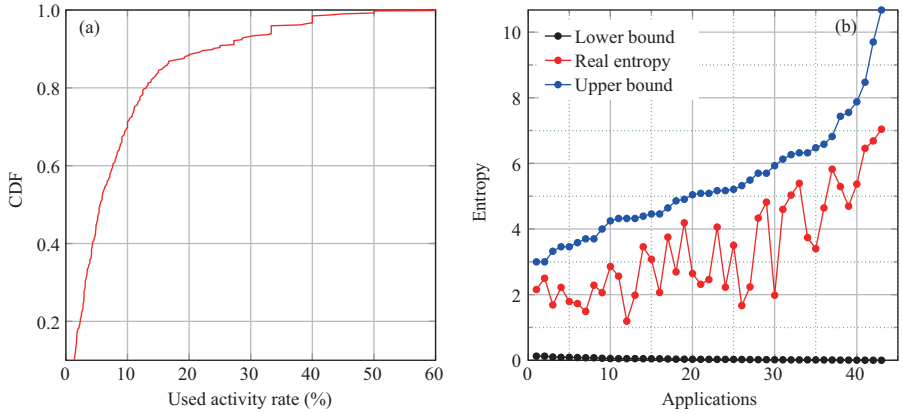

Figure 2 (Color online) Empirical results of how users interact with apps. (a) Users visit only a part of features in mobile app; (b) the entropy of visited pages for each app.

The median values for the increase of APK sizes and the number of activities are 15.45X and 16.24X, respectively.

Although the median package size of these top apps is only 69.3 MB, the actual storage space required for installing such top app (known as "install size"[10]) is several times larger. In the median case, 187 MB of local storage can be occupied after installing an app for the top apps. Thus, we also measure the increased rates of the installation size and APK size for the top apps. The increases in the installation size range from 1.09X to 7.10X than the APK size, and the installation size is 3.22X larger than the APK size in the median case. A recent survey indicates that users averagely install 80 apps per device[11], occupying a non-negligible part of available local storage. These results indicate apps should include more and more features to meet users' increasing needs in later versions, resulting in the significant increases of apps' complexities.

2.2 Sparse page access inside apps

To answer RQ2, i.e., how many features in an app are used by different users, we analyze a dataset of activity-level user behaviors released by a recent study [8]. The dataset contains 894542 visit-activity records collected from 64 college students for three months, covering 3527 activities from 389 apps. We filter out the records related to system apps and self-developed apps that cannot be downloaded from Android markets, and finally acquire a dataset consisting of 240 apps.

For each app in the dataset, we count the number of all activities visited by the users, and divide the number by the total number of activities, to compute the feature-usage ratio of the app. Figure 2(a) shows the distribution of the feature-usage ratio among all these 240 apps. The results show that users visit only less than 20% of activities for most apps (approximately 87%). This result indicates that users

10) App's download size and install size. https://support.google.com/googleplay/android-developer/answer/9302563?hl=en.
11) App Annie. https://www.appannie.com/en/insights/market-data/apps-used-2017/.

use only a small subset of features in mobile apps but they have to download the full-size APK files. From the users' perspective, installing unnecessary features on their devices is annoying, resulting in high consumption of the local storage, memory, and other resources. Thus, an infrequently used app of a huge size is very likely to be removed by the users in a short time.

2.3 Diverse usage among users

We answer RQ3, i.e., do different users focus on using different features in an app by using the same dataset as the one used for RQ2. We choose apps that have more than 5 users, and obtain 43 apps in total. We use the entropy[12)] to measure the diversity of activity usage among users. We count the number of users and calculate the probability of being visited for each visited activity in the dataset. For every single app, we compute the entropy as shown in the following equation:

$$E = -\sum_{i=1}^{n} p_i \ln p_i,$$

where n is the number of activities that have been visited by users, and p_i is the probability of being visited for the activity i.

The entropy is an unpredictability metric of which activity is visited. If all activities are visited equally, then all p_i values are equal to $1/n$, and the entropy hence takes the value $\ln(n)$ (upper bound). If the users' visits focus on one activity, and other activities are rarely visited, the entropy gets close to zero (lower bound). Figure 2(b) shows the distribution of the entropy among these 43 apps. Each app has an entropy between its upper bound and lower bound, indicating that its activities have diverse probabilities of being visited. This result indicates that different users not only focus on using the same subset of activities, but also may occasionally visit other activities.

2.4 Findings and implications

The preceding study results indicate that Android apps indeed become increasingly complex, containing more activities and resulting in larger sizes. However, only a limited subset of activities are frequently accessed, and some activities may even never be visited. This finding motivates us to equip apps with the capability to flexibly customize and remove those infrequently visited activities to reduce the APK size.

Furthermore, different users focus on different features in an app. This observation implies that these users need to use an infrequently accessed activity quite occasionally and opportunistically. Therefore, we cannot simply abandon those infrequently visited activities, as doing so can cause not to allow users' later visitations to them unless the users download the full-size app.

We need to devise a mechanism that allows the users to not only keep the basic and frequently used features when the apps are downloaded, but also can visit the infrequently used features whenever necessary (i.e., on demand).

3 Requirements and key challenges

Motivated by the findings of the empirical study, we propose an approach, named LegoDroid, that aims to enable the on-demand installability of apps without compromising user experiences. It is advocated that Android-app development adopts the component-based development paradigm[13)], and developers tend to design their apps to be modularized. Such modularity of Android apps brings opportunities for decomposing an app into a set of loosely-coupled bundles, each of which contains parts of the app's code and resources for realizing a specific set of features.

For each app, users just need to download and install a launch bundle that can correctly launch the main page of the app. In this way, the users keep only those code and resources in the launch bundle on their device, so that the device's storage can be saved and the required runtime resources can also be reduced. When the users need to access the features not included in the launch bundle, the related

12) Wikipedia. Shannon Entropy (Diversity Index). 2022. https://en.wikipedia.org/wiki/Diversity_index.
13) Google. Android Developer. 2022. https://developer.android.com.

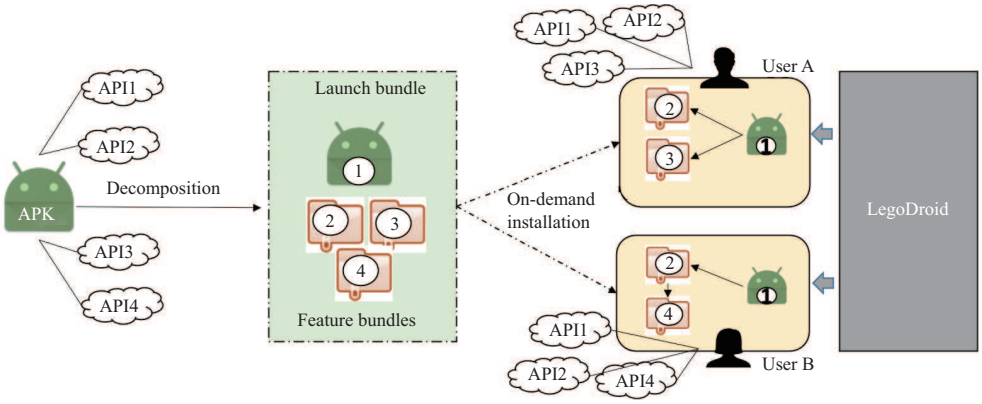

Figure 3 (Color online) Overview of our LegoDroid approach.

feature bundles are allowed to be dynamically downloaded and installed on the fly, achieving on-demand installation of bundles.

Considering the current Android ecosystem, our proposed LegoDroid approach should satisfy the following requirements.

Decomposable. In order to enable the on-demand visitations of apps, the LegoDroid approach should decompose an app into multiple bundles and support dynamic loading of them. Thus, users can access necessary features with a few of bundles instead of a full-size app.

User-friendly. The users can easily and regularly download and install the decomposed apps on their devices with our LegoDroid-enabled system compared to the current app-delivery mechanism. Our approach should be agnostic to users, and should not compromise user experiences or change the way of user interactions with apps.

Developer-friendly. The LegoDroid approach should work for legacy and on-the-shelf apps so that developers do not have to manually re-develop their apps to adopt our approach.

To achieve the preceding requirements, we still need to address two key challenges.

First, there exist complex dependencies among code and resources in an app, and it is difficult to obtain the complete and accurate dependencies using only static analysis [9]. If required classes or resources are not included in the bundles, the decomposed app can suffer from crashes.

Second, we need to identify the exact call sites to launch those activities in feature bundles as well as dynamically loading code and resources in advance. Only those activities started by explicit intent[14)] can be precisely determined through static analysis. Without deterministically resolving these activities' names at runtime, it is hardly possible to dynamically load feature bundles beforehand.

4 The LegoDroid approach

This section presents the overview of LegoDroid and describes the detailed design of LegoDroid's components.

4.1 Approach overview

Figure 3 shows the overview of LegoDroid. Given an Android app, LegoDroid automatically decomposes the app into decoupled bundles, and enables the users to visit these bundles on demand.

In an Android app, an activity represents a single user interface for users to interact with, and developers tend to use activities to separate different sets of features. Naturally, LegoDroid performs the decomposition at the activity level; i.e., each activity is regarded as an atomic feature that may be desired by the users. Considering the robustness and reliability, we apply the class-level decomposition rather than the method-level decomposition. For example, we cannot directly remove those non-executed methods since later dynamically loaded bundles may use them. After decomposition, LegoDroid generates two types of bundles.

14) Google. Intents and Intent Filters. 2022. https://developer.android.com/guide/components/intents-filters.html.

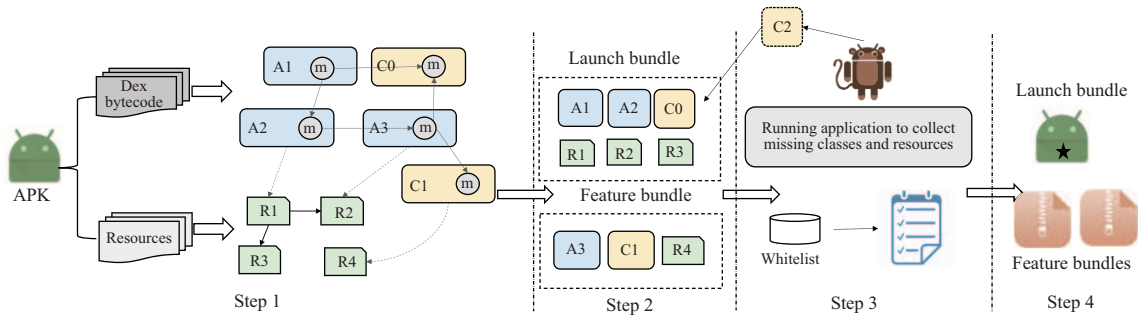

Figure 4 (Corlor online) The workflow of app decomposition.

Table 1 Summary of notations

Notation	Description	Notation	Description
Σ	Class dependency	$C/R/A$	The set of classes/resources/activities
Γ	Resource dependency	$\mathbb{C}$	The activity-related classes
Ω	Class-resource dependency	$\mathbb{R}$	The class-related resources
DG	The dependency graph	$\hat{\kappa}/\tilde{\kappa}$	The code of launch/feature bundle
$\Pi/\tilde{\Pi}$	A launch/feature bundle	$\hat{\gamma}/\tilde{\gamma}$	Resources of launch/feature bundle

• A launch bundle, which contains the code and resources related to the app launching process. It is the entry of the decomposed app. In addition, a launch bundle also includes other resources that cannot be decomposed and should be mandatorily included in the APK file such as native libraries, manifest file, and assets, which are vital to make the app work correctly.

• A set of feature bundles, each of which contains the code and resources related to a specific activity (named feature activity) that is not in the launch bundle. A feature bundle is requested and dynamically loaded when the users navigate to a new activity that is not included in the launch bundle. By default, none of feature bundles are installed on the device.

In order to support on-demand installation of bundles, LegoDroid places hooks into the related system services to detect which activity users want to visit and downloads the target feature bundle if it has not been visited before. Afterwards, LegoDroid dynamically loads the target feature bundle, and then launches the target activity. In Subsections 4.2 and 4.3, we describe the details of app decomposition and on-demand installation, respectively.

4.2 App decomposition

For each Android app, the decomposition process focuses on the code and resources that can be decoupled and loaded dynamically. Other files, such as native libraries and configure files (e.g., AndroidManifest.xml and resources.arsc), are required for launching and executing the app correctly, and thus are packed into the final package of the launch bundle directly. Figure 4 illustrates the workflow of app decomposition in LegoDroid. We describe the details of each step as follows. Table 1 shows the all notations we used in the following formalization.

4.2.1 *Dependency analysis*

To identify the boundaries of launch bundles and other bundles for decomposition, LegoDroid first analyzes the dependencies among the classes in the code, the dependencies among the resources, as well as the dependencies between the code and the resources. We next formally define the three types of dependencies.

Definition 1 (Class dependency)**.** The class dependency Σ is a relation between classes C: $\Sigma \subset C \times C$. Given two classes c_1 and c_2, $\langle c_1, c_2 \rangle \in \Sigma \Leftrightarrow \exists$ methods $m_1 \in c_1, m_2 \in c_2,$ and m_1 invokes m_2.

Since Android apps are developed in Java, which is an object-oriented language, LegoDroid decomposes the code at the class level. Give an Android app, we construct its call graph $\mathrm{CG} = (V, E)$ as follows. Each vertex $v_m \in V$ represents a method m. If method m_1 invokes method m_2, then CG contains an edge $v_{m_1} \to v_{m_2} \in E$. However, Android developers can use intent (i.e., inter-component communication, ICC) to interact with a component. The intent provides a core set of callback methods that can be

invoked by the Android system to enter a new state. For example, an activity may start another activity through an intent, which is similar to a method invocation of the target activity's callback method. There is no direct invocation between the caller and callback method.

To address this problem, we conduct the ICC analysis [10], which can infer the implicit invocations to the callback methods of a component. Then, we build an extended call graph ECG, which is an expansion of CG, produced as follows. For each method m, if it initiates a request to start a component p, then ECG $= (V, E)$ contains an ICC edge $v_m \rightsquigarrow v_b \in E$, for each callback method b in component p.

Definition 2 (Resource dependency)**.** The resource dependency Γ is a relation between resources R: $\Gamma \subset R \times R$. Given two resources r_1 and r_2, $\langle r_1, r_2\rangle \in \Gamma \Leftrightarrow \exists r_1, r_2 \in R, r_1$ includes r_2.

Besides class dependencies, we also need to consider the resource dependencies of an app. There are two kinds of resources in an Android app. Some reside in the res folder, and can be referenced with identifiers such as R.type.resName in the code and @type/resName in the resource files. Others reside in the assets folder, and can be referenced with the AssetManager object and the loadUrl method. The former reference can be detected explicitly, while the latter reference cannot be detected explicitly. For example, developers may load a local Web page in the assets folder with the loadUrl method in a WebView component. However, the loading of other resources such as CSS files, JS files, and images in the assets folder is not explicitly specified.

In this article, we just focus on the decomposition of resources in the res folder. In future work, we plan to conduct the resource decomposition on all resources. For resources in the res folder, one resource could include others with specific attributes and resource identifiers. For example, a layout file r_1 can include another layout file r_2 with the "include" tag whose attribute specifies the name of r_2. After parsing all resources files and traversing relevant attributes, we can get the dependencies among different resources.

Definition 3 (Class-resource dependency)**.** The class-resource dependency Ω is a relation between classes C and resources R: $\Omega \subset C \times R \cup R \times C$. Given a class c and a resource r, $\langle c, r\rangle \in \Omega \Leftrightarrow$ c references r in the code, $\langle r, c\rangle \in \Omega \Leftrightarrow r$ references c in the resource file.

Additionally, we deal with the dependencies between the code and the resources. In Android, one class could reference other resources to construct user interface based on a hexadecimal ID or the resource name and resource type. Meanwhile, developers can also include customized UI components (i.e., classes inherit from system-provided UI components) in a resource file.

Definition 4 (Dependency graph)**.** A dependency graph DG is a five-tuple DG $= \langle C, R, \Sigma, \Gamma, \Omega\rangle$, where C is the set of classes, R is the set of resources, $\Sigma \subset C \times C$ is the dependency relation between classes, $\Gamma \subset R \times R$ is the dependency relation between resources, and $\Omega \subset C \times R$ is the dependency relation between classes and resources.

To better capture the relationships among these dependencies, we define a dependency graph to represent the code and resource dependencies in Android apps.

Figure 4 shows an example dependency graph. There are five classes, where A1 depends on A2 and C0, A2 depends on A3, and A3 depends on C0 and C1. There are three resources, where R1 includes R2 and R3. For class and resource dependencies, A2 references R1, A3 references R2, and C1 references R4.

4.2.2 *Constructing launch bundle and feature bundles*

In the second step, LegoDroid constructs a launch bundle and a set of feature bundles. The launch bundle consists of the classes and resources related to launching-related activities, and each feature bundle consists of the classes and resources related to each remaining activity. Therefore, we should analyze the classes and resources related to each activity to determine the boundaries of each bundle for decomposition.

Definition 5 (Activity-related classes)**.** We consider that a class c is required by an activity a if there exists a dependency path from a to c in the dependency graph DG. Formally, given an activity $a \in A$ where A is whole set of activities, the activity-related classes of a is

$$\mathbb{C}_a = \{c \in C \setminus A \mid \langle a, c\rangle \in \Sigma \ \text{ or } \ \exists c_0, c_1, \ldots, c_k, \langle a, c_0\rangle, \langle c_0, c_1\rangle, \ldots, \langle c_k, c\rangle \in \Sigma\}.$$

Definition 6 (Class-related resources)**.** We consider that a resource r is required by a class c if c references r, or if there exists a resource r', c references r' and there exists a dependency path from r'

and r. Formally, given a class c, the class-related resource of c is

$$\mathbb{R}_c = \{r \in R \mid \langle c, r\rangle \in \Omega\} \cup \{r \in R \mid \exists r_0, r_1, r_2, \ldots, r_k, \langle c, r_0\rangle \in \Omega, \langle r_0, r_1\rangle, \langle r_1, r_2\rangle, \ldots, \langle r_k, r\rangle \in \Gamma\}.$$

Based on the preceding definitions, we give the formal definitions of the launch bundle and the feature bundle.

Definition 7 (Launch bundle and feature bundle)**.** Given a set of activities $\Lambda = \{a_1, a_2, \ldots, a_n\}$, the launch bundle is

$$\Pi = \langle \hat{\kappa}, \hat{\gamma}\rangle,$$

where

$$\hat{\kappa} = \Lambda \cup \left(\bigcup_{i=1}^{n} \mathbb{C}_{a_i}\right), \ \ \hat{\gamma} = \bigcup_{c_i \in \hat{\kappa}} \mathbb{R}_{c_i}.$$

For each launching-related activity u, the corresponding feature bundle is

$$\tilde{\Pi}_u = \langle \tilde{\kappa}_u, \tilde{\gamma}_u\rangle,$$

where

$$\tilde{\kappa}_u = (\mathbb{C}_u \cup u) \setminus \hat{\kappa},$$

$$\tilde{\gamma}_u = \left(\bigcup_{c_i \in \tilde{\kappa}_u} \mathbb{R}_{c_i}\right) \setminus \hat{\gamma}.$$

A launch bundle contains activities Λ that are required in successfully launching the original app. For each activity a, we compute its set of activity-related classes $\mathbb{C}_a$. The classes $\hat{\kappa}$ added to the launch bundle are defined as the union of the launching-related activities Λ and the corresponding activity-related classes $\mathbb{C}_a$ for each activity a in Λ. The resources $\hat{\gamma}$ added to the launch bundle are defined as the union of class-related resources $\mathbb{R}_{c_i}$ for each c_i in $\hat{\kappa}$.

For each activity u that is not included in the launch bundle, we pack its classes $\tilde{\kappa}$ and resources $\tilde{\gamma}$ as a feature bundle $\tilde{\Pi}$. $\tilde{\kappa}$ denotes the set of related classes in $\mathbb{C}_u$ except those classes in $\hat{\kappa}$. $\tilde{\gamma}$ denotes the set of referenced resources that are in the union of $\mathbb{R}_{c_i}$ for each class c_i in $\tilde{\kappa}$ but not in the referenced resources $\hat{\gamma}$. In order to guarantee the runtime correctness, we need to keep all dependent classes and resources of each transition path to the target activity. Meanwhile, we need to keep all dependent classes and resources when interacting with the target activity excluding new activities and their successive dependencies. We then use the activity transition graph (ATG) [11] to further reduce redundancies between different feature bundles, in which a successor activity can reuse the code and resources of its predecessor activity. In such way, the feature bundle will only contain its exclusive classes and resources. For an activity supporting deep linking [12], developers can even generate a "slim" app with only a base bundle and a feature bundle that contains the target activity.

For example, as shown in Figure 4, let us assume that A1 and A2 are launching-related activities to pack into the launch bundle. Then, the activity-related classes ($\mathbb{C}_{\mathrm{A1}}$ and $\mathbb{C}_{\mathrm{A2}}$) and their related resources ($\mathbb{R}_{\mathrm{A1}}$ and $\mathbb{R}_{\mathrm{A2}}$) are computed based on their dependencies. Based on these results, class C0 and resources R1, R2, R3 are found to be required by activities A1 and A2, and are packed into the launch bundle. Since C0 and R2 have been packed into the launch bundle, we do not pack them into the feature bundle that contains activity A3. Hence, A3's feature bundle consists of C1 and R4.

4.2.3 *Iterative and back-complementary recovery*

To capture all the dependencies, LegoDroid requires a precise and complete call graph, which is very challenging to produce due to the event-driven nature and reflection calls in Android apps [13]. To alleviate these issues, we devise an iterative and back-complementary recovery mechanism to supplement missing code and resources for decomposed bundles.

When the decomposed app without certain required classes and resources is run, a system exception (indicating that the required classes or resources cannot be found) is thrown. We collect and analyze such exception logs to infer the missing classes and resources that cannot be detected by static analysis.

LegoDroid provides a back-complementary recovery tool to automatically supplement the missing code and related resources, repack bundles, and launch the target bundle in an iterative fashion. The back-complementary recovery tool iteratively launches the target bundle, runs dynamic tests on the target

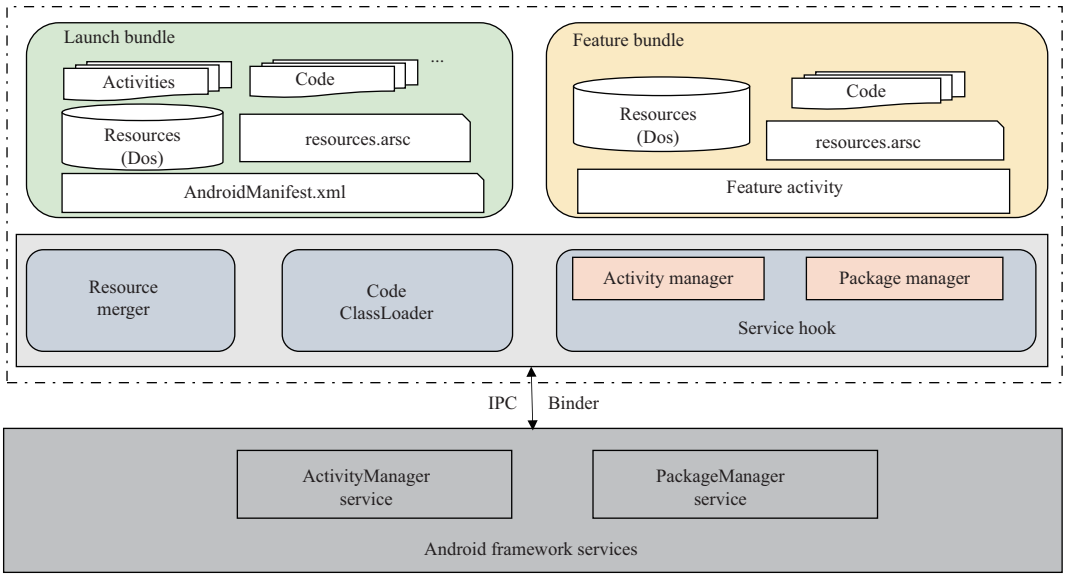

Figure 5 (Color online) System architecture of LegoDroid.

bundle (e.g., clicking a button), and adds the detected missing classes and resources back to the target bundle until no exceptions are detected.

The back-complementary recovery tool applies the dynamic analysis to test the functionalities in the target bundle and collects logs involved with missing classes and resources. For example, we can employ the record-and-replay technique [14] to record the actions performed during launching and visiting a target activity in the original app, and then iteratively replay these actions to test the corresponding bundle after decomposition. In practice, developers always need to conduct comprehensive testing before releasing apps. Our approach can reuse these existing test scripts and tools to perform the dynamic analysis, and perform the back-complementary recovery for decomposed bundles, effectively improving the accuracy and efficiency.

Moreover, there can be many common third-party libraries that are used by different apps. Hence, we can reuse the analysis results of these third-party libraries to speed up the back-complementary recovery tool, especially when LegoDroid fails to compute the accurate dependencies of such libraries. To this end, we maintain a cache of third-party libraries and their classes along with resources that have been successfully analyzed by the back-complementary recovery tool. Based on the existing efforts [15], given an app to be analyzed, if any third-party library used by the app is found in the cache, we can directly add the corresponding classes and resources into the target bundle without re-analyzing this library.

4.3 On-demand instant installation

In order to support on-demand instant installation, we need to modify the runtime environment of Android apps. On one hand, our system should be aware of the activity to be launched when users trigger activity transition, and then detect whether the target activity has already resided in the local device. On the other hand, after the feature bundle is downloaded, the system should on demand load the code and resources in the feature bundle at runtime. Figure 5 shows the system architecture of our approach. We design three key components to satisfy the requirements.

Service hook. In order to support on-demand installation, LegoDroid places hooks into the system services to detect which activity the users want to visit and download the corresponding feature bundle if the activity has not been visited before. The Android system manages and runs apps with system services via the Binder[15)], which is an Android-specific inter-process communication (IPC) mechanism. It adopts a client-server model, where each client has a proxy object to communicate with the server. For example, ActivityManagerService (AMS) is responsible for managing the lifecycles of activities, and PackageManagerService (PMS) is responsible for package installation and information management. For a running app, its process holds clients of system services (e.g., the activity manager, abbreviated as

15) Google. Binder: the inter-process communication mechanism in Android system. 2022. https://developer.android.com/reference/android/os/Binder.html/.

AM, is the client of AMS). We place hooks into these clients, which intercept the invocation of the target activity and detect whether the activity has been installed in the local device. If not, these hooks initiate a request to download the corresponding feature bundle.

Code ClassLoader. The code ClassLoader is responsible for dynamically loading the code in a feature bundle. It extracts and moves the code to the private source-code folder of the app so that the app can dynamically load such piece of code with dynamic code loading (DCL) based on DexClassLoader. The usage of dynamic code loading may introduce some vulnerabilities (e.g., remote code injection) [16], which are out of the scope in this article. We plan to check the integrity of each bundle before dynamically loading the code in the bundle [17].

Resource merger. The Android system uses the hexadecimal ID to reference a resource in the res folder, and these IDs are stored into the code in a hard-coded way during compilation. When running an app, the system queries the exact resource path and content from a resource table[16] according to the hard-coded ID. By default, each app accesses only resources according to a resource path that directs to its APK file. The resource merger extends the single resource path to a resource list, adds the path of the feature bundles' resources to the resource list, intercepts resource requests, and retrieves resources along with the resource list until finding required resources.

We implement all three components running within the app's process so that they do not affect other apps. With both code and resources dynamically loaded, the decomposed app can run as the original app does. Meanwhile, we do not change any functionality of the original app. As such, the decomposed app can still work even if the original app has applied other dynamic loading technologies[17]. The memory occupied by a dynamically loaded feature bundle will be automatically taken over by the Android system, and we do not need to manually manage the unloading process of the feature bundle. Each requested feature bundle is downloaded and loaded into the memory only once, and subsequent access to the same activity can load code from the memory or the local device directly, reducing delays and improving user experiences.

5 Implementation

According to the design in Section 4, we implement a system for LegoDroid.

Dependency graph. We build a static analysis tool based on FlowDroid [18] and Soot[18] to generate the call graph. Then, we apply the IC3 tool [10] to infer the indirect invocations of callbacks of Android components and combine them with the call graph to generate the extended call graph. We use string matching to obtain dependent resources referenced by hexadecimal IDs, and extract exact parameters (e.g., resource type and resource name) to obtain dependent resources referenced by method invocations (e.g., getIdentifier). We also parse XML-format resource files to resolve the resource dependencies according to the official document.

Bundle generation. We extract classes from the complete bytecode to generate new dex-format files, and extract resources that are referenced by these classes for each bundle. With other files in the original package (e.g., AndroidManifest.xml, native libraries), we generate the launch bundle as a normal APK-format package. In order to support the dynamic resource loading, we extract IDs of these extracted resources from the resources.arsc file, and generate a simplified resources.arsc file. Finally, we pack all these needed files to generate a feature bundle as a zip-format file according to users' access paths. In order to better discover and identify the resources, we apply the digital object architecture (DOA) [19] to manage all the resources on the remote cloud, and each resource is wrapped as a unique digital object (DO).

Iterative and back-complementary recovery tool. We implement our iterative and back-complementary recovery tool upon the Android debug bridge (ADB) tool and the popular MonkeyRunner[19]. Our tool leverages the ADB tool to search apps' logs for entries that contain ClassNotFoundException and Resources$NotFoundException, so that our approach can detect the missing classes and resources. We implement a record-and-replay module based on the MonkeyRunner, and the developers

16) The Android system resolves the resources.arsc file to get the resource table that maintains the mappings between resources and IDs when running an app.

17) Tencent. A hot-fix solution library for Android, it supports Dex, library and resources update without reinstall apk. 2022. https://github.com/Tencent/tinker.

18) Soot. A framework for analyzing and transforming Java and Android applications. 2022. https://sable.github.io/soot/.

19) Google. MonkeyRunner. 2022. https://developer.android.com/studio/test/monkeyrunner/index.html.

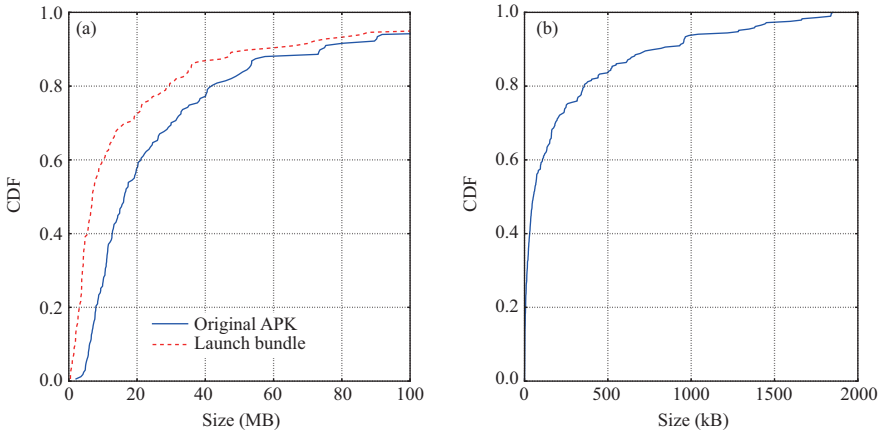

Figure 6 (Color online) Distribution of the size of decomposed bundles. (a) Launch bundle; (b) feature bundle.

can record a sequence of actions that can be used to launch the target bundle and do random testing with Monkey[20] to test functionalities included in the target bundle.

System integration. Although there exist various kinds of techniques to place hooks into system services, such as Xposed[21], VirtualApp[22], and Reptor [20], these techniques introduce extra overhead and may affect the apps' performance. There exists a trade-off between desirable performance and deployability. Actually, we can also implement LegoDroid based on VirtualApp to apply an app-level solution. We choose to directly modify the source code of the Android Open Source Project (AOSP) to build LegoDroid. We implement LegoDroid on every single version from Android 4.0 to Android 8.0 with only a few lines of modification on the Android framework, and apps generated by our approach can correctly run on these modified systems. Due to the space limit, we report only the evaluation results on a modified system based on AOSP version 7.1.1.

6 Evaluations

To evaluate the effectiveness of LegoDroid, we conduct comprehensive evaluations from four aspects. First, we apply LegoDroid on an extensive set of apps to demonstrate its effectiveness in saving the initial download size. Second, we evaluate the efficiency and robustness of the iterative back-complementary recovery tool on a set of open-source apps. Third, we evaluate the runtime performance of the decomposed apps on the LegoDroid-enabled system. Finally, we select a set of popular apps that have stable updates, and demonstrate how LegoDroid can work with version upgrades by preserving a substantial percentage of reusable code. We need to address that we use the cumulative distribution function (CDF) to show the distribution of each metric when evaluating LegoDroid.

6.1 Instant installation over real-world apps

In order to evaluate the practicality of our approach on real-world apps, we first analyze the top-1000 most downloaded apps without repackage protection from Google Play, as the subjects that we can apply our static analysis to measure the size distribution of launch bundles and feature bundles. These apps involve most app categories of Google Play.

The size of launch bundles. Figure 6(a) shows the size distribution of launch bundles and their original apps. The results show that the median value for the saving of the initial download size is 44.17%, resulting in a substantially less download time and local storage consumption. In the median case, the size of launch bundle is 4.6 MB, which is close to the 4 MB size limit for Instant apps.

For some apps, the launch bundle does not save much storage compared to the original apps. The reason is that developers may place most functionalities in the launching-related activities of their apps, and thus most code and resources have been packed into the launch bundle. Meanwhile, our current approach just decomposes resources that reside in the res folder, and other resources are packed into the

20) Google. UI/Application exerciser monkey. 2022. https://developer.android.com/studio/test/monkey.html.
21) Xposed. A framework to hook Android platform APIs. 2022. http://repo.xposed.info/.
22) Lody. An open source implementation of MultiAccount. 2022. https://github.com/asLody/VirtualApp/.

launch bundle directly. Some apps place almost all of resources in the assets folder (e.g., apps of the mobile game category), so the savings of storage for these apps can be quite marginal.

The size of feature bundles. As shown in Figure 6(b), users need to download only a feature bundle whose size is less than 500 kB for 84.23% of cases. To make a comparison, a recent study of httparchive.org[23] reports that the average web page size in 2021 was 2225 kB. In other words, a feature bundle is much smaller than a common Web page, and thus users do not need to wait for a long time to download the feature bundle. The reason is that some feature bundles may just contain a small set of classes, and their dependent resources have been packed into the launch bundle. Therefore, their sizes are relatively small.

These results indicate that LegoDroid can lower the barrier for accessing a new app due to the reduced sizes of the decomposed apps. In addition, to visit a new activity that has not been installed in the devices, users just need to download a small-size feature bundle. Most of the delays are even shorter than visiting a new web page in an app. On average, the overall size of all generated bundles of an app increases by 12.68% due to allowable redundancy to assure runtime correctness. Fortunately, users often visit a small part of bundles as shown in our empirical study, and the overall size of visited bundles can be reduced as well.

6.2 Robustness of decomposed apps

To evaluate the robustness of the mechanism of iterative and back-complementary recovery, we conduct an evaluation on a set of open-source apps. Although we can decompose legacy apps directly, developers may have used some tools to protect their apps from being reverse-engineered. Therefore, we could fail to run the decomposed apps. To avoid such issues, we verify the correctness of our decomposition process using 100 open-source apps with high downloads and ratings from F-Droid[24] and GitHub.

It is impossible to exhaustively explore all activities of all tested apps, so we just evaluate those important activities with small depth on the activity transition graph (ATG) [21]. For each app, we choose the home activity and those splash activities involved in the launching process to generate the launch bundle, and choose activities that are successor nodes of the home activity on the ATG to generate feature bundles. At runtime, LegoDroid simply ignores transitions of activities that are not included in either the launch bundle or feature bundles.

Efficiency of supplementing missing code and resources. We use the iterative back-complementary recovery mechanism to supplement the missing code and resources so that decomposed apps can run correctly. We count the required number of iterations to supplement the launch bundle and the feature bundles for these 100 apps. Figure 7 shows that we can successfully supplement the missing code and resources after 8 iterations in the median case, and no more than 10 iterations for 80% of the apps. Notice that every iteration needs to add only one class. Compared to the number of classes (ranging from 1160 to 6327, with 3647 in the median case) in an app, the additional cost of iterations is quite marginal.

Robustness verification. We further verify the decomposition robustness of LegoDroid on the 100 open-source apps. We first use the Monkey tool to generate random streams of user events, such as clicks, touches, and gestures to run the original app for one minute. Indeed, a longer time of executions may reach higher coverage. However, as reported by Li et al. [9], running automated UI testing for only 30 s can produce enough informative logs. During the executions, we use the MonkeyRunner to record these actions. The recorded actions are then used to be replayed on the decomposed apps running in the LegoDroid-enabled system for a fair comparison. For each app, we run both the original app and the decomposed app for 10 times to collect the logs. We successfully run our decomposed apps in the LegoDroid-enabled system without crashes and any extra exception. Within affordable efforts, we also randomly select 10 apps and perform manual inspection to mitigate the issues that random testing may not expose behaviors to cause crashes. According to collected logs, our decomposed apps do not throw unexpected exceptions or cause errors during the executions, demonstrating the robustness of LegoDroid.

6.3 User-perceived load time and memory consumption

We evaluate the runtime performance of decomposed apps using the 100 apps as described in Subsection 6.2. We use a Nexus 6 (3 GB RAM, 32 GB ROM, Quad-core 2.7 GHz) running Android 7.1.1 as

23) Archive H. The HTTP archive tracks how the web is built. 2022. http://httparchive.org/.

24) F-Droid is an installable catalog of Free and Open Source Software (FOSS) apps for the Android platform (https://www.f-droid.org/).

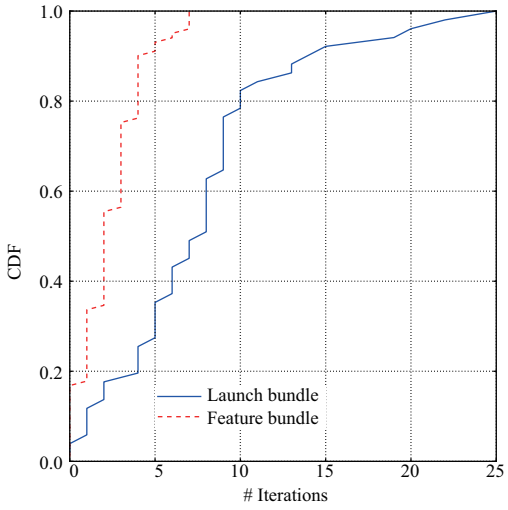

Figure 7 (Color online) Distribution of #iterations for bundles.

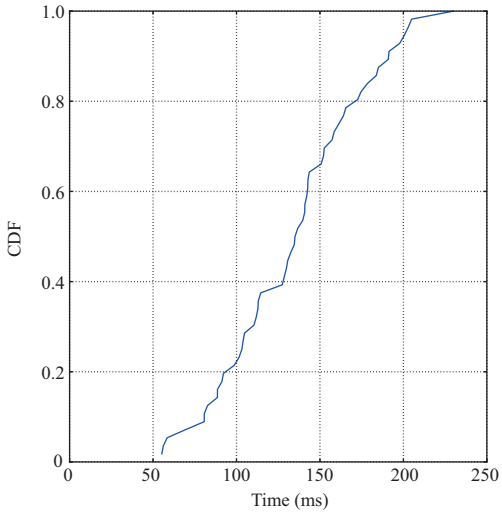

Figure 8 (Color online) Distribution of the time spent on merging feature bundles.

our test device. We measure the overhead when loading the launch bundles with the LegoDroid-enabled system. For each app, we also choose activities that are successor nodes of the home activity on the ATG as feature activities for testing, and load them in the LegoDroid-enabled system. For fair comparisons, we then launch the same activity in the original app on a clean system. We compare the loading time and memory consumption, respectively.

6.3.1 *Loading time comparison*

Figure 8 shows the distribution of time cost on merging resources in feature bundles. We do not include the download time of feature bundles since it depends on the network condition. In the median case, the merging process costs only 135.85 ms. Figure 9(a) shows that loading the launch bundle performs even better than directly loading the original app. The launch bundle contains only a part of code extracted from the original app, and thus the launching process is more efficient. Launching a decomposed app in the LegoDroid-enabled system can save 13.06% of the loading time than loading the original app on the clean system in the median case. We can achieve maximum 32.26% reduction in loading time in our dataset. Figure 9(b) shows that loading a feature activity in the LegoDroid-enabled system costs less time for the warm start compared to launching the same activity in the original app on the clean system. In the median case, we can save 10.93% of the loading time. Meanwhile, we can achieve 26.93% improvement at best.

6.3.2 *Memory consumption*

We also measure the memory usage of our approach as shown in Figure 9(c). We find that loading the launch bundle requires less memory compared to loading the original app directly. In the median case, loading a decomposed app with LegoDroid can save 8.83% of the memory usage. We can save 28.9% of the memory usage at best. Figure 9(d) shows the distribution of the memory usage when loading the same activity in both the decomposed app and the original app, respectively. In the median case, loading a feature activity in the decomposed app can save 9.14% of the memory usage. We can save 16.30% of the memory usage at best. With our approach, we effectively reduce the code size of an Android app, and it costs less memory to load the `dex` file (bytecode of Android apps) and reduce the time spent on looking for classes that are randomly distributed in the `dex` file. Most of the irreducible memory consumption is from mandatory functionalities, e.g., loading images and initiating network requests, which may mitigate our overall efficiency. Fortunately, our approach can save the memory usage and reduce the launching time without changing any app logic.

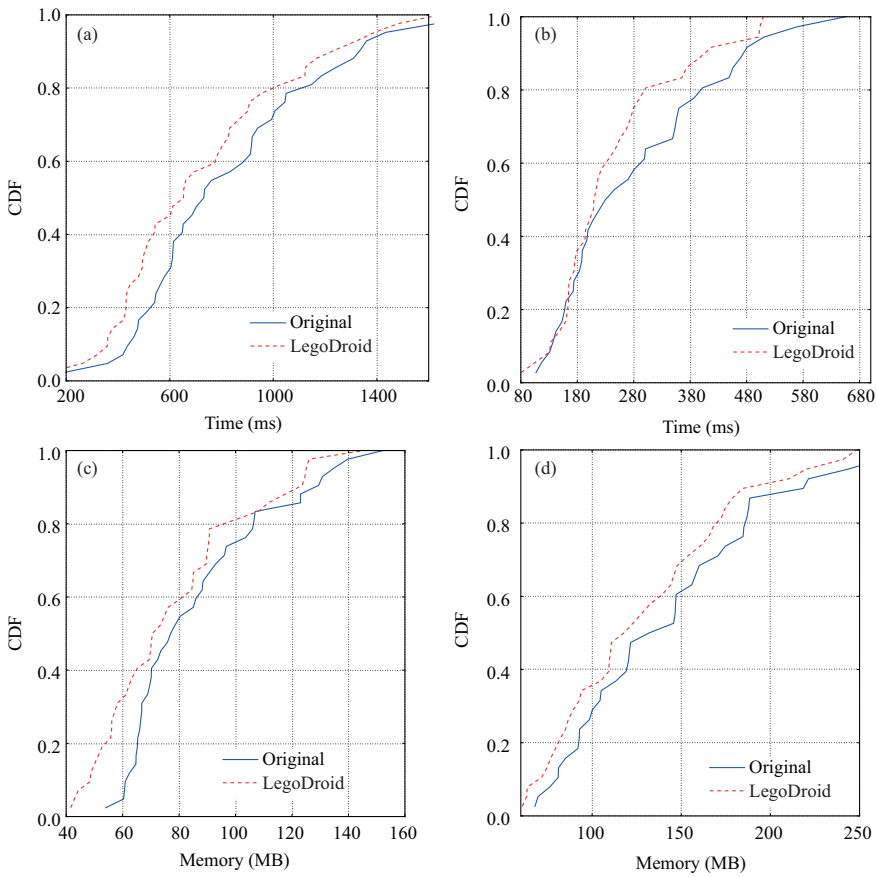

Figure 9 (Color online) Runtime performance comparison. (a) Loading time of launch bundles; (b) loading time of feature bundles in the warm start; (c) memory usage of loading launch bundles; (d) memory usage of loading feature bundles.

6.3.3 *Comparison with Google Instant apps*

To further demonstrate the effectiveness of our approach, we choose to compare the performance of LegoDroid-enabled apps and Google Instant apps[25] when requesting the same functionality. The results show that instant apps perform worse than their original apps in terms of both loading time and memory usage. Instant apps run based on app-level virtualization environment with the Google Play service, which introduces non-negligible overhead.

6.4 Reusability with version evolution

Usually, an app is frequently updated, e.g., weekly or even every two or three days. Intuitively, the developers can apply LegoDroid to a new version and release the newly generated launch bundle. However, it was reported that when updating mobile apps, users' major concerns are the storage and network traffic [22]. Hence, it is important to deliver newly generated bundles as small as possible. We then explore how LegoDroid can help reuse the code and resources from an older version to the latest one.

Since LegoDroid needs to support on-demand loading, only classes and resources with consistent name and content across different versions can be directly reused. However, the developers may use obfuscation tools to produce anonymized class names and resource names, which can affect the reusability. Therefore, we manually select those open-source apps that have released versions on Google Play without obfuscation and have stable update frequencies.

As shown in Table 2, we select 12 apps that have at least three released versions (except for the Piebald, which has only two released versions) to analyze the reusability of LegoDroid. For the two adjacent versions V_{t-1} and V_t of an app, we decompose both of them to obtain the launch bundles Π_{t-1} and Π_t. We define the coverage of V_{t-1} to V_t as $\frac{\Pi_{t-1} \cap \Pi_t}{\Pi_t}$. We observe that some launch bundles of

25) We get Instant apps from Google Play. https://play.google.com/store/apps/collection/promotion_3002d0f_instantapps_featuredapps. We do not show detailed results here due to space limit.

Table 2 The effect of version update on apps

App name	Description	Coverage
Douban Movie	Movie info client	82.31%, 86.04%, 95.44%
Drinks	Help make cocktails	81.56%, 100%, 99.78%
Easy xkcd	Comics client	99.34%, 100%, 99.87%
Iven News Reader	Light feed reader	93.28%, 99.39%, 93.54%
K9 mail	Mobile e-mail client	100%, 92.38%, 100%
Phonograph Music Player	Music player client	100%, 100%, 100%
Piebald	Photo client	50%, 50%
PocketHub	Client for github	100%, 66.07%, 100%
RedReader	Client for reddit	100%, 100%, 99.51%
Gallery	Photo client	99.40%, 100%, 99.72%
Timber Music Player	Music player client	99.56%, 97.86%, 73.71%
Yahnac for Hacker News	Client for hacker news	84.04%, 94.25%, 100%

older versions even can be applied directly to the newer version. Meanwhile, the newer version may add new features or introduce new third-party libraries, which can result in low reusability. Fortunately, we find that the coverage of most apps can reach above 80%. For those apps published with obfuscation technologies, developers can apply incremental obfuscation[26)] to keep the consistent names of code and resources with a mapping file. Hence, apps with obfuscation can also benefit from the incremental update of LegoDroid.

7 Discussion

In this section, we describe some issues that may potentially affect the generalization and effectiveness of LegoDroid, and discuss how to alleviate them.

Missing code and resources. To alleviate the limitations of state-of-the-art static analysis techniques [9], LegoDroid employs the record-and-replay technique to collect missing code and resources. However, it is difficult to exhaustively explore all the behaviors in an activity. As a result, some missing code and resources may exist when unexplored behaviors are triggered. To address this issue, a runtime complementary mechanism could be introduced with LegoDroid to retrieve the missing code and resources.

Unreliable network connection. LegoDroid dynamically requests and downloads feature bundles. Therefore, under some poor or even unavailable network conditions, users may fail to access a feature bundle that has not been downloaded before. In such a case, LegoDroid could display an "unavailable" page. However, given the increasingly pervasive deployment of Wi-Fi and stations, especially under the recently emerging "edge" environment, the network access is expected to become highly stable to avoid this issue. In addition, we need to address that once a feature bundle is downloaded by LegoDroid, subsequent visits will directly load bundles from the local device. Moreover, alternative solutions can consider sending a reduced feature bundle (e.g., low-quality images), or prefetching feature bundles by predicting users' behaviors.

Flexible app removal. Once users want to uninstall a decomposed app, they just need to uninstall the launch bundle as a normal app. LegoDroid will monitor the uninstall requests, and automatically remove the downloaded feature bundles and other temporary files. In addition, LegoDroid can enable fine-grained app management that allows users to remove only those infrequently visited feature bundles to release the local storage instead of uninstalling a whole package [23].

Additional cost. We have demonstrated that LegoDroid can save local storage, launching time, and memory usage. Intuitively, we can assume that the energy drain can also be reduced. LegoDroid may introduce other additional costs. Requesting feature bundles requires the data transfer, and the cost can vary according to the network connection and bandwidth. Hence, we cannot simply guarantee that LegoDroid can always gain benefits for all apps. However, given that feature bundles are usually non-frequently visited pages, LegoDroid still provides its values in practice.

Impacts of software analysis. Indeed, the LegoDroid's effectiveness relies on software analysis. Our current implementation uses some existing software analysis techniques/tools such as MonkeyRunner and

26) Proguard. The open source optimizer for Java bytecode. 2022. https://www.guardsquare.com/en/proguard.

Monkey to generate bundles. Moreover, evaluations indicate that our prototype system can work correctly and efficiently. For further improvement, we can reuse existing test scripts to test a generated bundle more effectively.

Security. LegoDroid loads feature bundles through dynamic code loading based on the DexClassLoader method provided by the Android system. However, Poeplau et al. [16] identified some vulnerabilities (e.g., remote code injection) related to the incorrect usage of dynamic code loading. To address this issue, we plan to check the integrity of each bundle before being dynamically loaded [17].

Generalizability. Currently we focus only on the open Android platform. Hence, the approach over close platforms, such as iOS, cannot be directly evaluated. However, we believe that the ideas of our approach are still generalizable, with the adaptation efforts according to the characteristics of the system and the software analysis for generating bundles.

8 Related work

On-demand visitation. Google Instant apps and Tencent mini programs provide on-demand visitation, but they also require extra developer efforts. Bhardwaj et al. [24] proposed ephemeral apps that dynamically load code and resources from a remote server. They further proposed AppSlicer [6], which creates app slices based on classes and resources loaded at runtime when launching an activity. However, AppSlicer requires a constant network connection and downloads required classes and resources in real time. Our LegoDroid approach downloads only feature bundles at the first visitation, but no more downloads are required for subsequent usages.

Decomposition. Gui et al. [25] and Liu et al. [26] statically analyzed apps to find the usage of ad libraries and rewrite bytecode for privilege de-escalation or better user experiences. Rubinov et al. [27] and Zhang et al. [28] focused on extracting and offloading code to remote servers or trusted environments. Chandrashekhara et al. [29] proposed the BlueMountain to decompose an app's data management logic and assign users more controls to load different data-management code portions on demand. LegoDroid decomposes an app at the granularity of the activity to support dynamic visitation without affecting user experiences.

Software debloating and refactoring. Jiang et al. [30] and Xie et al. [31] automated code and resource trimming for Android apps by static analysis. Huang et al. [32] proposed a static analysis technique to remove code elements that are relevant to user-specified unwanted UI elements in Android apps. Qian et al. [33] proposed a framework that leverages multiple control-flow heuristics to customize the app binary based on users' specifications. Our LegoDroid approach combines static and dynamic analyses to detect the boundaries of activities and their dependent resources, and can support on-demand visitation as well as eliminating the app bloat. Mahouachi [34] and Bavota et al. [35] applied software remodularization technologies to improve the maintainability by dividing the software into independent and loosely-coupled modules. Mourad et al. [36] explored the code clone of software and can significantly reduce the code size through clone refactoring. In contrast, LegoDroid analyzes the dependencies among both classes and resources, and divides an installation package into multiple bundles, without modifying the logical structure of the app. LegoDroid can benefit from the remodularization and other refactoring techniques to further reduce the coupling and size of code, and make it easier to distinguish the boundaries of activities.

9 Conclusion

In this article, we present LegoDroid to decompose Android apps and support flexible installability. LegoDroid allows developers to decompose an app into various bundles, and allows users to download and access infrequently used features on the fly. Our evaluations show that LegoDroid can save 44.17% of the initial download size in the median case, and achieve better runtime performance. In the median case, LegoDroid can reduce 13.06% and 10.94% of the launching time for launch bundles and feature bundles, respectively. Meanwhile, LegoDroid can efficiently reduce the network traffic for app updating by reusing code and resources. In future work, we plan to extend our decomposition to other code and resources and improve the accuracy of static analysis.

Acknowledgements This work was supported by National Key Research and Development Program of China (Grant No. 2020YFB2104100), National Natural Science Foundation of China (Grant Nos. 61725201, 62161146003), Beijing Outstanding

Young Scientist Program (Grant No. BJJWZYJH01201910001004), Beijing Nova Program (Grant No. Z211100002121159), and PKU-Baidu Fund Project (Grant No. 2020BD007).

References

1 Quach A, Erinfolami R, Demicco D, et al. A multi-OS cross-layer study of bloating in user programs, kernel and managed execution environments. In: Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, Dallas, 2017. 65–70

2 Li H, Ai W, Liu X, et al. Voting with their feet: inferring user preferences from app management activities. In: Proceedings of the 25th International Conference on World Wide Web, Montreal, 2016. 1351–1362

3 Ncube C, Oberndorf P, Kark A W. Opportunistic software systems development: making systems from what's available. IEEE Softw, 2008, 25: 38–41

4 Balasubramaniam S, Lewis G A, Simanta S, et al. Situated software: concepts, motivation, technology, and the future. IEEE Softw, 2008, 25: 50–55

5 Guo Y, Li Y, Yang Z, et al. What's inside my app?: understanding feature redundancy in mobile apps. In: Proceedings of the 26th Conference on Program Comprehension, Gothenburg, 2018. 266–276

6 Bhardwaj K, Saunders M, Juneja N, et al. Serving mobile apps: a slice at a time. In: Proceedings of the 14th European Conference on Computer Systems, Dresden, 2019. 1–15

7 Liu Y, Xu E, Ma Y, et al. A first look at instant service consumption with quick apps on mobile devices. In: Proceedings of the 26th International Conference on Web Services, Milan, 2019. 328–335

8 Ma Y, Hu Z, Gu D, et al. Roaming through the castle tunnels: an empirical analysis of inter-app navigation of Android apps. ACM Trans Web, 2020, 14: 1–24

9 Li L, Bissyandé T F, Papadakis M, et al. Static analysis of Android apps: a systematic literature review. Inf Software Tech, 2017, 88: 67–95

10 Octeau D, Luchaup D, Dering M, et al. Composite constant propagation: application to Android inter-component communication analysis. In: Proceedings of the 37th International Conference on Software Engineering, Florence, 2015. 77–88

11 Zhang Y, Sui Y, Xue J. Launch-mode-aware context-sensitive activity transition analysis. In: Proceedings of the 40th International Conference on Software Engineering, Gothenburg, 2018. 598–608

12 Ma Y, Hu Z, Liu Y, et al. Aladdin: automating release of deep-link APIs on Android. In: Proceedings of the 27th World Wide Web Conference, Lyon, 2018. 1469–1478

13 Wang Y, Zhang H, Rountev A. On the unsoundness of static analysis for Android GUIs. In: Proceedings of the 5th SIGPLAN International Workshop on State Of the Art in Program Analysis, 2016. 18–23

14 Gomez L, Neamtiu I, Azim T, et al. RERAN: timing- and touch-sensitive record and replay for Android. In: Proceedings of the 35th International Conference on Software Engineering, San Francisco, 2013. 72–81

15 Ma Z, Wang H, Guo Y, et al. LibRadar: fast and accurate detection of third-party libraries in Android apps. In: Proceedings of the 38th International Conference on Software Engineering, Austin, 2016. 653–656

16 Poeplau S, Fratantonio Y, Bianchi A, et al. Execute this! Analyzing unsafe and malicious dynamic code loading in Android applications. In: Proceedings of the 21st Annual Network and Distributed System Security Symposium, San Diego, 2014

17 Falsina L, Fratantonio Y, Zanero S, et al. Grab'n run: secure and practical dynamic code loading for Android applications. In: Proceedings of the 31st Annual Computer Security Applications Conference, Los Angeles, 2015. 201–210

18 Arzt S, Rasthofer S, Fritz C, et al. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In: Proceedings of the 35th SIGPLAN Conference on Programming Language Design and Implementation, Edinburgh, 2014. 259–269

19 Kahn R, Wilensky R. A framework for distributed digital object services. Int J Digit Libr, 2006, 6: 115–123

20 Ki T, Simeonov A, Jain B P, et al. Reptor: enabling API virtualization on Android for platform openness. In: Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, Niagara Falls, 2017. 399–412

21 Azim T, Neamtiu I. Targeted and depth-first exploration for systematic testing of Android apps. In: Proceedings of the 2013 SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, Indianapolis, 2013. 641–660

22 Nayebi M, Adams B, Ruhe G. Release practices for mobile apps—what do users and developers think? In: Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering, Suita, 2016. 552–562

23 Singh I, Krishnamurthy S V, Madhyastha H V, et al. ZapDroid: managing infrequently used applications on smartphones. IEEE Trans Mobile Comput, 2017, 16: 1475–1489

24 Bhardwaj K, Gavrilovska A, Schwan K. Ephemeral apps. In: Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications, St. Augustine, 2016. 81–86

25 Gui J, Mcilroy S, Nagappan M, et al. Truth in advertising: the hidden cost of mobile ads for software developers. In: Proceedings of the 37th International Conference on Software Engineering, Florence, 2015. 100–110

26 Liu B, Liu B, Jin H, et al. Efficient privilege de-escalation for ad libraries in mobile apps. In: Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services, Florence, 2015. 89–103

27 Rubinov K, Rosculete L, Mitra T, et al. Automated partitioning of Android applications for trusted execution environments. In: Proceedings of the 38th International Conference on Software Engineering, Austin, 2016. 923–934

28 Zhang Y, Huang G, Liu X, et al. Refactoring Android Java code for on-demand computation offloading. In: Proceedings of the 27th Annual SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, Tucson, 2012. 233–248

29 Chandrashekhara S, Ki T, Jeon K, et al. BlueMountain: an architecture for customized data management on mobile systems. In: Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking, Snowbird, 2017. 396–408

30 Jiang Y, Bao Q, Wang S, et al. RedDroid: Android application redundancy customization based on static analysis. In: Proceedings of the 29th International Symposium on Software Reliability Engineering, Memphis, 2018. 189–199

31 Xie Q, Gong Q, He X, et al. Trimming mobile applications for bandwidth-challenged networks in developing regions. 2019. ArXiv:1912.01328

32 Huang J, Aafer Y, Perry D M, et al. UI driven Android application reduction. In: Proceedings of the 32nd International Conference on Automated Software Engineering, Urbana, 2017. 286–296

33 Qian C, Hu H, Alharthi M, et al. RAZOR: a framework for post-deployment software debloating. In: Proceedings of the 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, 2019. 1733–1750

34 Mahouachi R. Search-based cost-effective software remodularization. J Comput Sci Technol, 2018, 33: 1320–1336

35 Bavota G, Lucia A D, Marcus A, et al. Software re-modularization based on structural and semantic metrics. In: Proceedings of the 17th Working Conference on Reverse Engineering, Beverly, 2010. 195–204

36 Mourad B, Badri L, Hachemane O, et al. Exploring the impact of clone refactoring on test code size in object-oriented software. In: Proceedings of the 16th International Conference on Machine Learning and Applications, Cancun, 2017. 586–592