Where did my 256 GB go? A Measurement Analysis of Storage Consumption on Smart Mobile Devices

ASHISH BIJLANI*, Georgia Institute of Technology, USA
UMAKISHORE RAMACHANDRAN, Georgia Institute of Technology, USA
ROY CAMPBELL, University of Illinois at Urbana-Champaign, USA

This work presents the first-ever detailed and large-scale measurement analysis of storage consumption behavior of applications (apps) on smart mobile devices. We start by carrying out a five-year longitudinal static analysis of millions of Android apps to study the increase in their sizes over time and identify various sources of app storage consumption. Our study reveals that mobile apps have evolved as large monolithic packages that are packed with features to monetize/engage users and optimized for performance at the cost of redundant storage consumption.

We also carry out a mobile storage usage study with 140 Android participants. We built and deployed a lightweight context-aware storage tracing tool, called COSMOS[1], on each participant's device. Leveraging the traces from our user study, we show that only a small fraction of apps/features are actively used and usage is correlated to user context. Our findings suggest a high degree of app feature bloat and unused functionality, which leads to inefficient use of storage. Furthermore, we found that apps are not constrained by storage quota limits, and developers freely abuse persistent storage by frequently caching data, creating debug logs, user analytics, and downloading advertisements as needed.

Finally, drawing upon our findings, we discuss the need for efficient mobile storage management, and propose an elastic storage design to reclaim storage space when unused. We further identify research challenges and quantify expected storage savings from such a design. We believe our findings will be valuable to the storage research community as well as mobile app developers.

CCS Concepts: • **Information systems → Storage management**; **Mobile information processing systems**; • **Human-centered computing → Empirical studies in ubiquitous and mobile computing**.

Additional Key Words and Phrases: storage management; mobile; smartphones

ACM Reference Format:
Ashish Bijlani, Umakishore Ramachandran, and Roy Campbell. 2021. Where did my 256 GB go? A Measurement Analysis of Storage Consumption on Smart Mobile Devices. *Proc. ACM Meas. Anal. Comput. Syst.* 5, 2, Article 28 (June 2021), 28 pages. https://doi.org/10.1145/3460095

*A part of this work was done when the author was a graduate student at the University of Illinois at Urbana-Champaign.
[1]COSMOS is available publicly at https://github.com/cosmost/cosmost

Authors' addresses: Ashish Bijlani, ashish.bijlani@gatech.edu, Georgia Institute of Technology, 266 Ferst Dr, Atlanta, Georgia, USA, 30313; Umakishore Ramachandran, rama@cc.gatech.edu, Georgia Institute of Technology, 266 Ferst Dr, Atlanta, Georgia, USA, 30313; Roy Campbell, University of Illinois at Urbana-Champaign, 201 North Goodwin Avenue, Urbana, Illinois, USA, 61801-2302, rhc@illinois.edu.

2476-1249/2021/6-ART28 $15.00
https://doi.org/10.1145/3460095

1 INTRODUCTION

Smart mobile devices have evolved as versatile and indispensable tools for everyday personal computing needs. The evolution has led to a surge in mobile apps for entertainment, social networking, health tracking, and home automation. GooglePlay alone now hosts over 2.3 million mobile Android apps [2]. Users often install apps for faster performance and customized experience.

Nevertheless, the storage space on mobile devices is limited. Many of them are not provisioned with an external flash memory [63]. Low-end budget devices place further storage restrictions, and severely limit user experience [54]. As storage demands increase beyond the device capacity, users are forced to uninstall apps and delete data [57, 64]. Alternatively, users purchase cloud storage for data backup [12, 18, 21] or periodically upgrade devices by paying more for higher storage capacity [68, 69].

As such, efficient management of storage on these devices is becoming increasingly important. While anecdotal evidence suggests that user data (e.g., pictures, videos) consume high storage space, modern apps also pose high storage demand. The maximum permissible sizes of GooglePlay Android and iOS apps have only been growing since 2008. Today, a single app can occupy up to 4 GB of storage space. Yet, no systematic study has been carried out on the storage consumption behavior of modern mobile apps.

This work presents the first-ever detailed measurement study of storage consumption behavior of mobile apps. We perform multiple analyses on apps to report their overall behavior.

Large-scale longitudinal analysis. First, we carry out a longitudinal analysis of millions of GooglePlay Android apps to study the increase in app sizes over five years, from 2014 to 2019 (see §3). Our findings suggest that as app stores increase app size limits over time, developers create bigger apps that pose heavy storage demands. The number of apps consuming between 10 MB to 4 GB doubled in five years. Such apps are not limited to games and digital books, but spread across various categories. We also analyze metadata of millions of iOS apps for comparative analysis, and show similar finding.

We then perform static analysis of millions of Android apps to highlight the leading causes of increased sizes over time. Our findings show that as an app gains popularity, developers pack more features in the same app, creating *super apps* to engage users and monetize. The number of features in top apps doubled in five years. These include both free and paid features, where the latter are only unlocked once the user purchases them, but are needlessly always stored on the device.

The number of third-party libraries imported per-app for common tasks such as user authentication and advertisements grew by 10x in five years, adding to app size. However, only a small fraction of those library functions are actually used. Unused code contributes to unnecessary storage consumption.

Furthermore, we found that despite Google's effort to encourage developers to create small, device-optimized apps [22, 26, 27], developers continue to create "build once, run everywhere" universal apps that support multiple hardware (e.g., ARM, x86) and as well as regional languages. Such universal apps reduce engineering effort and offer best performance on every device type, but lead to redundant storage consumption.

Install-time behavior. We also evaluate storage consumption behavior of modern mobile apps during fresh installation (no usage) by analyzing top 30 GooglePlay apps. We found that upon installation, apps further expand to consume anywhere between 1.5x to 5x storage space of their package (see §5). This increase is attributed to additional files created for performance.

User study. Finally, we report runtime storage utilization of Android apps by leveraging usage traces from our user study with over 140 participants for 70 days (see §6). We built CosMos, a novel lightweight context-aware storage tracing tool, and deployed it on each participant's device. The

usage traces we collected revealed that only a small number of apps/features (and files) are actively used and furthermore usage is correlated to user context. These findings suggest a high degree of feature bloat and unused functionality in modern apps, which leads to wastage of storage.

Analyzing the traces, we also found that since apps are not constrained by storage quota, app developers freely use persistent storage by creating auxiliary files and frequently caching/prefetching cloud content for performance. Many apps hoard additional data, such as analytics to track user engagement, crash reports for debugging, and advertisements for monetization. However, unlike cached content, which is automatically deleted when the device runs low on storage space, app data continues to persist on the device, sometimes even after the app is uninstalled. We detected several old video/image advertisements and residual files on user devices.

Finally, guided by our findings, we identify challenges and opportunities to design an efficient mobile storage management system. Specifically, we propose context-sensitive elastic storage for smart mobile devices and identify multiple techniques such as content adaptation, deletion, deduplication, and compression to automatically reclaim storage space occupied by inactive apps/features. Our preliminary evaluation with top 30 Android apps shows that on average about 25 MB (21.34%) of storage savings could be achieved per app, with almost no latency or battery drain using just one of the aforementioned reclamation techniques.

Contributions. In summary, this paper makes the following contributions.

– We carry out a five-year longitudinal static analysis of millions of Android apps and report the increase in their sizes (§3) and highlight various sources of app storage consumption (§4).

– We further perform install-time study of top 30 Android apps to show that apps expand to consume additional storage space upon installation (§5).

– Leveraging the storage traces from 140 Android users, we also present our findings on the runtime storage utilization behavior of modern mobile apps (§6). To the best of our knowledge, this is the first-ever large-scale and detailed measurement study on the storage consumption behavior of modern mobile apps.

– Drawing upon our findings, we propose an elastic storage design for smart mobile devices, identify research challenges, and quantify expected storage savings from such a design (§7.1). We believe our findings will be valuable to not only the storage research community, but also mobile app developers.

2 BACKGROUND

In this section, we cover the basics of Android app anatomy as well as present a background on different storage areas available to mobile app developers on Android.

App structure. Android supports two types of development environments, namely Java and native C/C++. A Java source file is first compiled into a bytecode `.class` file, which is then linked with third-party `.jar` libraries to produce a `classes.dex` file. Each DEX file can contain up to 65,000 methods. Therefore, large apps typically consist of many `.class` files and split functionality across multiple DEX files. In contrast, C/C++ sources are compiled into dynamic `.so` libraries that contain native (e.g., ARM, x86) machine code. All DEX files, native libraries, and auxiliary *asset* files (e.g., icons) are packed into a single App Package (APK) zip file.

In addition to the APK, developers can optionally provide up to two monolithic *expansion files* [31], known as Opaque Binary Blobs (OBB), to add additional auxiliary multimedia resources as needed (e.g., sound, animations). OBB files allow developers to supplement the app APK and pack more functionality without bloating the APK.

Installation process. During app installation, the app APK is downloaded from GooglePlay. All app executable files (DEX, libs) are extracted from the APK for faster runtime access, and stored

Partition	Area	FS
/	Internal	RootFS
/system	Internal	EXT4
/data	Internal	EXT4
/data/media	*Prm Ext*	FUSE

App Storage Location	Dir Type
/data/app/app/base.apk	APK file
/data/dalvik-cache/app	OAT file
/data/data/app	Priv dir*
/*PExt*/data/Android/app	Pub dir*

App Dir*	Type
app/files/	Java Code, Data
app/libs/	Native Code
app/databases/	State (SQLite)
app/shared_prefs/	State (XML)
app/cache/	Cached Data

Table 1. *Storage paritions on Android devices. A large part of* internal */data parition is exposed as* primary external */data/media partition. Each app is assigned private (under internal) as well as public (under primary external) storage areas (dirs) to host its code and data. App files residing in public app storage area can be accessed by all other installed apps with read/write permissions. Typically app code (java classes and native libraries) and structured data (e.g., XML, databases) are stored in private app dirs and large unstructured files (e.g., OBB files) are stored in public app dirs. App APK and perf-optimized OAT files are stored in internal data partition.*

on the device under app-specific directories. At this time, any OBB files are also downloaded and stored on the external partition under an app-private directory (see Table 1.)

Unlike native code, DEX bytecode is agnostic to the native machine architecture; thus it requires compilation into native machine code to be able to run on the device. Older Android versions used the Dalvik Virtual Machine (VM) runtime that performed Just-in-Time (JIT) compilation of DEX files. Version 5 introduced Ahead-of-Time (AOT) compilation of DEX files into optimized `OAT` files during installation. AOT compilation of Java bytecode is configurable. It ranges from compiling "everything" to "interpret only". While the former improves app runtime performance, the latter provides performance similar to Dalvik VM. By default, most methods in an OAT file are precompiled to maximize runtime performance. Consequently, OAT files consume significant storage space and incur a longer installation time.

Once an app is installed, its compressed APK file is also stored on the device at all times for the app to directly access its asset files as well as for the system to perform delta updates to the app. This technique results in two copies of every executable file in an app: 1) the original (compressed) APK, and 2) an optimized OAT that trades in more storage for performance.

Storage areas and partitions. Android offers two types of storage areas, namely `internal` and `external`. The internal storage area is a built-in non-volatile flash memory containing critical partitions such as `boot` (for bootloader and kernel), `system` (for system software), and `data` (for app and user data). The internal storage, however, is not directly accessible to the user. In contrast, the external storage area contains only of the data partition and is directly accessible to the user (e.g., over USB). Table 1 shows various Android storage areas and partitions.

Furthermore, an Android device can contain two types of external storage areas: built-in non-removable (primary) and removable (secondary) external areas. An example of the latter would be `/sdcard` that is mounted when an external removable flash memory card is plugged into the device. However, many smart mobile devices today are not equipped with an external flash memory slot. Therefore, in this work, we only focus on the consumption of internal and primary external storage areas since they are fixed in size (non-removable) and shared by all apps with no per-app storage quota limits.

Storage abstractions. Android provides various storage abstractions, such as files, databases, and xml schemas. App developers can choose to store raw bytes in the form of traditional files or make use of high-level storage abstractions such as databases and xml schemas to store formatted data.

App storage directories. Each user-installed app is assigned two data directories: a default internal (under `/data/data/`) and an external (under `/data/media/Android/data`). These directories are named after the `app` package name. Table 1 shows various designated app data directories on

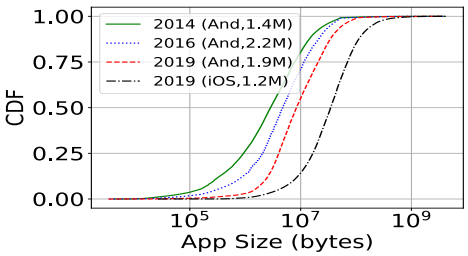
Android. Directories in internal storage are private and exclusively owned by respective apps. Whereas, directories in external storage are public and can be accessed by apps with permissions.

Being private to apps, internal directories are primarily intended for hosting proprietary code (e.g., java classes, libraries) and private data (e.g., user preferences/account). In contrast, being public, external directories are typically used for storing app auxiliary data (e.g., icons) and any public user content such as music, videos, photos, and documents that can be accessed by other apps as well.

Based on the type of content (raw bytes or formatted data), developers can pick the appropriate app directory to store it. For example, it is advisable to store temporary data such as web content and temporary files under `cache` because when the device runs low on storage space, files under `cache` are automatically deleted by the system to reclaim space. In contrast, `databases` and `shared_prefs` directories contain stateful persistent data (e.g., user preferences). Due to lack of per-app storage quotas on Android, there are no limits on the amount of data an app can host in its storage directories.

3 MODERN MOBILE APPS

To understand how app sizes have evolved over time, we performed a longitudinal study of millions of GooglePlay Android apps from 2014 to 2019. For comparative analysis, we also performed a study of millions of modern iOS. In this section, we report our study methodology and findings.



(a) Growth over the years.

Max Size	Android Apps (%) '14 (1.4M)	Android Apps (%) '19 (1.9M)	iOS (%) '19 (1.2M)
50MB	99.224	94.139	65.33
100MB	0.3221	5.0365	20.81
1 GB	0.4395	0.7973	13.63
2 GB	0.0124	0.0243	0.193
3 GB	0.0008	0.0016	0.028
4 GB	0.0001	0.0006	0.009

(b) Change in five years.

Fig. 1. *App installation sizes of GooglePlay Android and AppleStore iOS apps from 2014 to 2019.*

Questions. Our study answers the following questions about mobile apps:

- *How much storage is consumed by app installation files? How has this changed over time?*
- *Does device type (tablet vs. phone, high- vs low-resolution display) affect app installation size?*
- *What effect does app category have on its size? What categories are likely to be large?*
- *What effect does app installation size have on its popularity (downloads) and vice-versa?*

Methodology. Popular mobile app stores such as GooglePlay and AppleStore provide APIs to fetch the most current metadata on all apps, such as app size, category, and number of downloads. We wrote scripts to download and analyze such metadata. It is worth noting that GooglePlay does not provide a precise number of downloads (or installs) per app; instead, it only provides download range approximations. AppleStore, on the other hand, provides no statistics on app downloads.

In this study, we only report app sizes: storage space consumed by all app components downloaded from the official store during installation. An Android app consists of an APK file and up to two OBB files. Therefore, its size is the sum of all such files downloaded during installation. In contrast, an iOS app is downloaded as a single zipped archive `.ipa` file, which constitutes 100% of its size.

Dataset. Our dataset consists of only metadata (name, size, category) of millions (M) of GooglePlay Android and iOS apps. Specifically, the dataset includes the metadata of,

– 1.4M Android apps collected in October, 2014 by PlayDrone [66].
– 2.2M Android apps collected in December, 2016 by OSSPolice [19].
– 1.1M and 1.9M Android apps we collected in July, 2018 and August, 2019, respectively.
– 1.2M iOS apps we collected in August, 2019 from AppleStore using iTunes Preview [3].

Our GooglePlay crawler to discover and download apps is based on PlayDrone [66].

App sizes. Figure 1a shows our findings on app sizes. We found a sharp increase in both the number of available GooglePlay Android apps (1.4 vs. 2.2M) and the average app size (7.89 vs. 17.16 MB) in five years. In 2014, only 20% of apps were more than 10 MB in size. That number grew by 50% in 2016 and doubled in 2019; that is, over 40% of the apps from 2019 were over 10 MB in size. We also found a small (<1%) increase in the number of apps over 1 GB.

Furthermore, over 5% apps from 2019 consume 50-100 MB in size, compared to only 0.3% such apps in 2014 (see Figure 1b). We believe this increase is due to the change in GooglePlay app submission policies in 2015 that allowed developers to publish an app with APK size of up to 100 MB (i.e., 2x the prior limit of 50 MB [32]). Figure 3 shows change in maximum permissible app sizes over time.

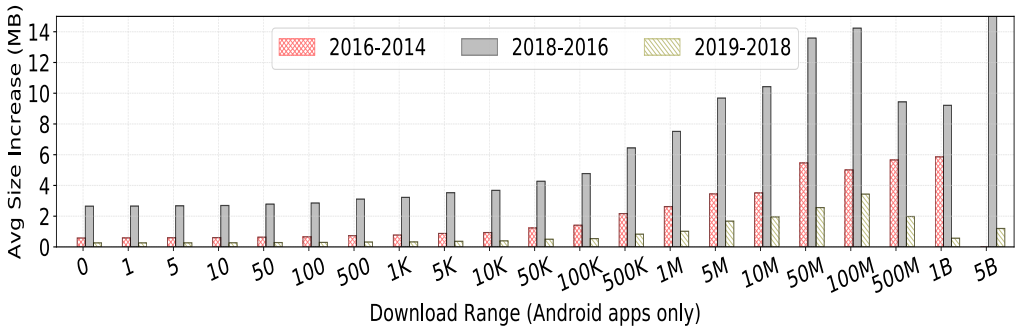


Fig. 2. *Analysis of app sizes of common GooglePlay Android apps from 2014 to 2019.*

Common apps across years. We compared the common set of apps over the years to understand their evolution over time. Our findings suggest that popular apps grew more in size on average, compared to other apps. This behavior is captured by Figure 2. Of the 931.9K common apps between 2014 and 2016, 1,270 (0.13%) received 10-100M downloads and grew by 3 MB on average. Whereas, 18 received 1-5B downloads and grew by 6 MB on average. Similarly, of the 356.2K common apps between 2016 and 2018, 16 (0.45%) received 500M-1B downloads and grew by 10 MB on average. The aforementioned pattern suggests that once an app becomes popular (i.e., receives more downloads), more features are added in the same app to engage and monetize users, which results in higher storage consumption. We further discuss this behavior in §4.

iOS apps. Apple has also been increasing the Over-The-Air (OTA) download limits since 2008 [6]. As a result, developers create bigger apps. We found that 34.67% of 1.2 million iOS apps consume more than 50 MB. 20.81% of apps consumed 100 MB–1 GB. Almost 11% were 100–250 MB in size. Whereas, 13.63% of apps consumed 1–2 GB.

An astute reader may notice that the iOS apps are much bigger in size than Android apps. However, the app size numbers are not directly comparable. Unlike Android app sizes that depict the sizes of compressed app APK files, iOS app sizes depict app install sizes and not the sizes of compressed IPA files downloaded from AppleStore. iOS apps contain native machine executable code. Whereas, Java code in Android apps is compiled into the native machine code during installation, resulting in higher install app size. We further discuss this phenomenon in §5.

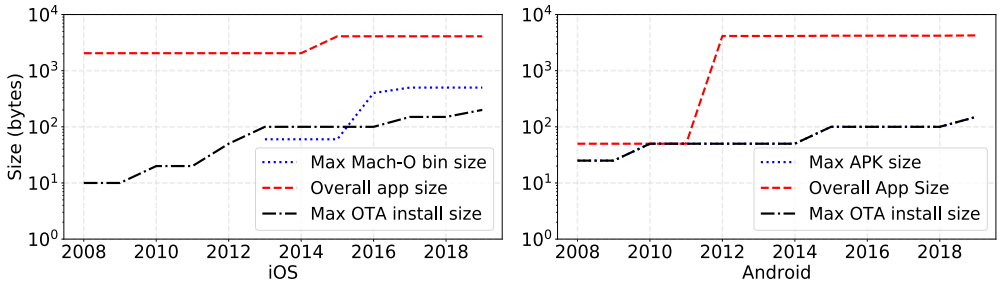


Fig. 3. *Change in maximum permissible app sizes and Over-The-Air (OTA) install sizes under GooglePlay Android and AppleStore iOS apps policies over time [6, 29, 30]. GooglePlay imposes a limit of 100 MB on Android APKs [32]. However, developers can create bigger apps by supplementing the APK with two monolithic OBB files, each up to 2 GB in size (see §2). Similarly, iOS apps have a 60 MB limit on the main Mach-O executable file to enforce clear separation between required binary assets and media files [6].*

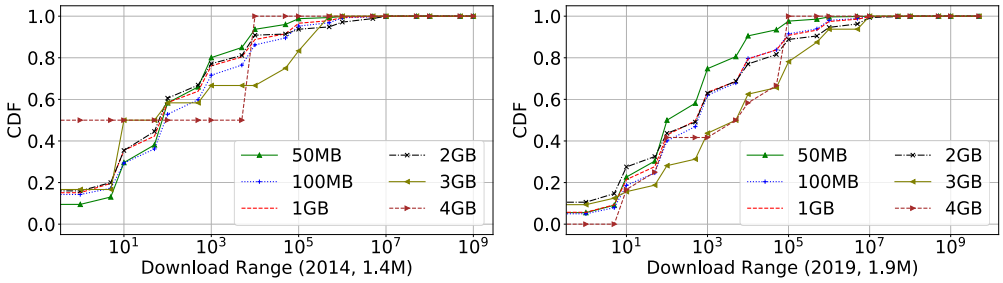
It is also important to note that the iOS app sizes used in this study are *universal sizes* provided by iTunes Preview [3], and not device-specific sizes available from the AppleStore app on an iOS device. Apple uses App Thinning [4] technology that detects the target device type and downloads only device-specific code and auxiliary components when installing an app. As such, device-specific iOS apps are typically smaller in size than universal iOS apps. For instance, the size of Facebook and Uber apps we collected in 2019 for iPhone8 Plus devices were 231.5 MB and 277.3 MB, respectively. Whereas, the sizes of their universal iOS apps from the same year were 426 MB (1.84x) and 384 MB (1.38x), respectively.

App size vs. category. We compared sizes of apps from 2014 and 2019 across different categories, and found that average app size in each category grew at least by 50% in five years. Table 2 shows our findings. Average size of media/videos apps increased the most: 2.8x from 7.02 MB in 2014 to 19.70 MB in 2019; followed by gaming apps that increased 2x in size (from 13.73 MB to 33.72 MB), on average. Gaming apps were the highest in number as well each year, although we saw a decline in their overall number from 17.26% in 2014 to 14.90% in 2019. In contrast, the average size of digital books increased by almost 50%, from 8.44 MB in 2014 to 12.77 MB in 2019.

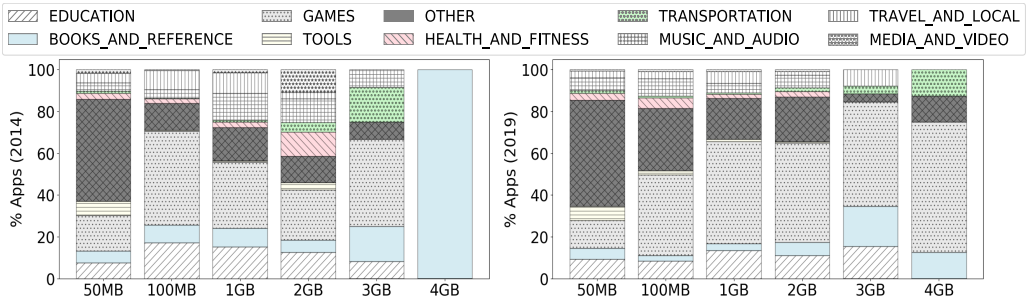
Figure 4b further shows the change in distribution of app sizes across various categories. We found that large (>= 1 GB) apps are not limited to graphics-rich games or digital books, but span across multiple categories, including fitness, education, and music/video. While limited to digital books in 2014, modern games, music/video, and travel apps can also occupy up to 4 GB.

App size vs. downloads. To understand the effect of app size on its download count, we first group apps according to their sizes as small (<= 50 MB), medium (>50 MB and < 1GB), and large (>=1GB). We then compared app downloads across these groups to obtain a percentage cumulative distribution of apps downloads against their sizes, shown in Figure 4a. We found that a fewer number of large apps receive as many downloads as small or medium apps. In fact, the number reduces even further for apps over 3 GB in size. This behavior was consistent across apps we analyzed from 2014 as well as 2019, which suggests that the size of an app may affect its overall download count. Specifically, users may shun downloading large apps, given the storage space constraints on mobile devices.

However, the percentage of popular apps (with >= 1 million downloads) remains almost the same across groups. For example, 6.10% of medium and 6.79% of large apps were downloaded at least one million times in 2019, indicating that popular apps are downloaded regardless of their size. Note that we use app download range as a proxy for app popularity.



(a) *# App downloads vs. sizes. Fewer large apps (>=1GB) receive as many installs as small (<=50MB).*



(b) *App sizes vs. categories. Large (>=1GB) apps span across multiple categories.*

Fig. 4. *Analysis of millions of GooglePlay Android apps collected in 2014 [66] and 2019.*

App Category	And ('14,1.4M) %	And ('14,1.4M) MB	And ('19,1.9M) %	And ('19,1.9M) MB	iOS ('19,1.2M) %	iOS ('19,1.2M) MB
GAMES	17.26	13.73	14.90	33.72	16.80	103.86
EDUCATION	7.76	10.29	9.34	16.74	10.12	71.19
TRNSPT/MAPS	1.23	5.16	1.24	14.06	1.27	62.59
TRAVL/LOCAL	4.66	9.33	3.22	19.27	4.44	67.49
BOOKS/REFS	5.56	8.44	5.02	12.77	2.94	70.75
MUSIC/AUDIO	3.83	10.58	6.20	15.76	2.91	51.32
MEDIA/VIDEO	1.68	7.02	0.64	19.70	2.37	48.80
HLTH/FITNESS	2.85	7.12	3.40	18.26	4.29	55.20
TOOLS/UTILS	6.61	2.30	6.44	8.15	6.58	35.64
OTHER	48.56	4.92	49.60	13.22	48.28	45.34
OVERALL	100	7.89	100	17.16	100	61.22

Table 2. Change in the number and average sizes of Android and iOS apps across categories.

App size vs. device. Android supports many devices, each one is different in multiple aspects, such as form factor (tablet vs. phone), hardware architecture (x86 vs. ARM), or screen size and resolution. Historically, developers created apps that were compatible by default on most devices by packing multiple redundant resources in a single APK (e.g., multiple resolution images for different screen sizes, multiple native libraries for different CPU architectures). During app usage, Android system selects appropriate device-specific resources. However, as mobile hardware evolved and devices became more fragmented, this design resulted in bloated apps that waste storage space.

For the past few years, Google has been encouraging developers to publish multiple smaller APKs for the same app [26], each optimized for a particular device configuration. App Bundles [22] were also introduced in 2018 to allow developers to submit a single bundle (max 150 MB) of code/resources instead of submitting multiple APKs to target different devices. During installation, GooglePlay then generates appropriate APKs on the fly from app bundles based on the device configuration so that only absolutely necessary code/resources are downloaded on the device.

To determine the number of apps supporting optimized APKs, we wrote a Soup [52] scraper that collects generic metadata from GooglePlay app webpages. Apps which support optimized per-device APKs do not show fixed installation sizes in their description on GooglePlay webpages; instead their size varies with each device [26]. We analyzed the metadata for app installation sizes and found that only 58.3K (3.4%) of 1.7M apps from 2019 contain "varies with device" in their size description (i.e., support device-specific APKs). 4.3K of them received at least 1M downloads. Whereas, the remaining 96.6% apps were *universal*; that is, developers create a single app to target multiple devices, resulting in apps that consume unnecessary storage. 22.6K such apps were downloaded at least 1M times.

To summarize:

- Both the number of mobile apps and the maximum permissible app size, have been increasing over the years: installation of a single app can consume up to 4 GB of space.
- Developers capitalize on increasing app size limits to create feature-rich and immersive apps. As an app gains popularity, more features are packed in the same app to engage and monetize users, resulting in storage-heavy super apps.
- Apps over 1 GB are not limited to games or e-books, but spread across multiple categories.

4 STATIC CODE ANALYSIS

We also performed longitudinal static analysis of millions of free GooglePlay Android apps to study their evolution over time and identify various causes of app size increase. We present our study methodology and findings here.

Questions. We sought answers to the following questions.

- *What causes mobile apps to continue to grow in size? How has this behavior changed over time?*
- *What third-party SDKs (libraries) are used by apps? How much do they contribute to app size?*
- *What kind of assets and resources are used by apps? What effect do they have on app size?*

Methodology. We used state-of-the-art static analysis tools (e.g., apktool, dex2jar) to decompile the app APKs, and manually inspected the content and size of DEX files, native libraries, and any asset files present (e.g., icons). We further analyzed APK Java executables using LibScout [7] to determine third-party Java libraries being used and detected API calls. Similarly, we analyzed native libraries used.

Dataset. Our dataset consists of APK and OBB files of millions of free GooglePlay Android apps.

- 1.1M free apps collected in Oct'14 by PlayDrone [66].
- 1.6M free apps collected in Dec'16 by OSSPolice [19].
- 1.7M free apps that we collected in Aug'19.

Table 4 summarizes various sources of app APK storage consumption as per our findings.

App composition. We found that modern apps are large and complex installation packages, consisting not only of executable binaries and libraries, but also various auxiliary resources, such as icons, images, animations, sounds, videos, text and xml files needed by the app to provide rich user experience. For instance, files under app resource dir `res/mipmap/` contains various icons needed by the app. Similarly, `res/values/strings.xml` file holds app strings. On Android, these auxiliary

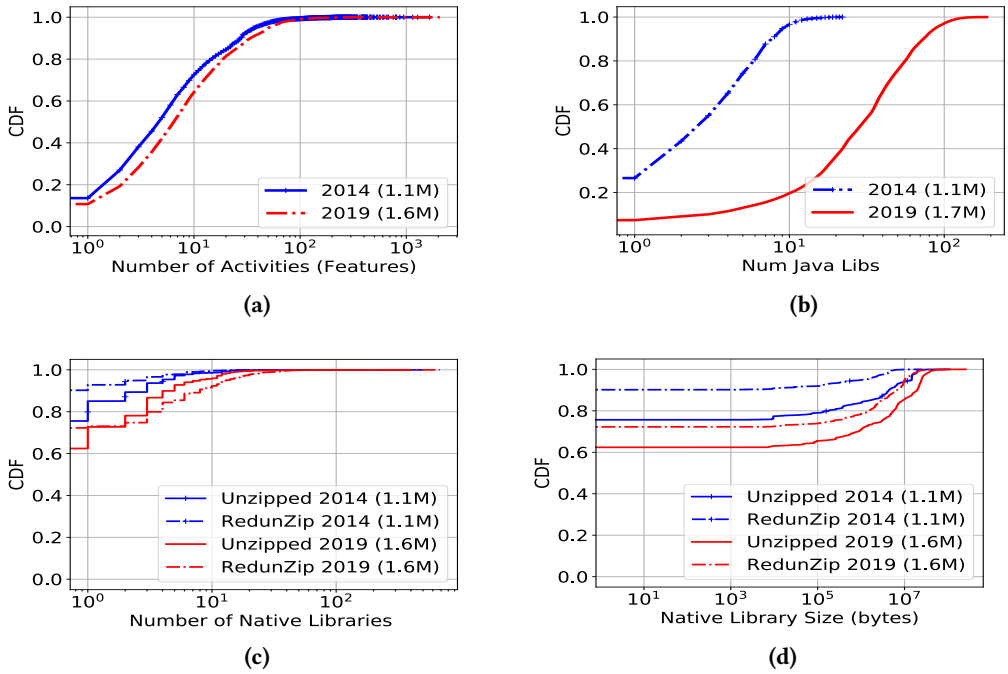


Fig. 5. *Static analysis of GooglePlay Android APKs collected in 2014 [66], and 2019, respectively.*

files are part of the app APK, and are not extracted on device during installation; instead, apps read them directly from the zipped APK. We found a 27.23% and 32.71% increase in the number and average size of such auxiliary files, respectively from 2014 to 2019.

App features. A typical Android app has multiple *Activities*. Each Activity implementation is placed in a separate Java (or C/C++) file, and offers a particular feature (e.g., login, take photo). Since users interact only with Activities, we use them as a proxy for app features. Our analysis reveals that modern apps contain more Activities. In particular, we found an average of 14.14 Activities per app across app from 2019, compared to only 10.18 in 2014. Figure 5 shows our findings.

Compared to an average increase of 4 Activities per app across all apps, we found an average increase of 100 Activities in top 30 GooglePlay apps, each with at least 5 million downloads. Table 6 reports the numbers. These results corroborates our findings on the increase in sizes of top apps from §3. Specifically, as apps become popular (e.g., reach 100K installations), developers pack more features within the same *super app* to offer one-stop experience to their users and create an ecosystem for more monetization. For instance, Tencent added payments and ride sharing, among other services. Similarly, Uber added e-bike rentals and food delivery services. Consequently, such super apps grow substantially more in size on average over the years compared to other apps. This behavior is further captured by Figure 2 and discussed in §3.

Java libraries. To quickly bring apps to market, app developers often focus on unique app features, and rely on third-party libraries or Software Development Kits (SDKs) to import common features.

We analyzed apps in our dataset with LibScout [7] to list various Java libraries being used. LibScout is resilient against common bytecode (e.g., identifier renaming) as well as code-based (e.g., API hiding) obfuscation techniques.

We first leverage pre-compiled Java JAR files from Maven [61] and JCenter [11] to generate a database of unique library profiles. The database is then fed to LibScout, which decompiles Java

SDK Type	Popular Examples	# Apps (%) 2014	2019
Advertising	FBAudience, AdCol, Chartboost	41.14	82.70
Analytics	Flurry, Google, HockeyApp	3.90	21.31
SocialMedia	Facebook, Twitter, WeChat	29.78	74.28
Cloud	Amazon, Dropbox, Twilio	1.56	3.99
Utilities	JodaTime, OkHttp, Crashlytics	74.78	87.83

Table 3. Distribution of various types of SDKs across Android apps in five years, from 2014 to 2019.

App APK (2019)	Avg (%)
ARM Native Lib	18.3
- Dup ARM Libs	63.69
Non-ARM Native Lib	28.31
Java Classes/SDKs	29.39
Misc (res, assets)	24

Table 4. Various sources of app APK storage consumption.

`classes.dex` files in apps, compares them against the database, and produces a list of matching libraries found in apps.

LibScout detected 5.77M and 55.30M Java libraries (full matches); whereas, 3.4M and 20M were partially matched or unmatched in 1.1M (from 2014) and 1.7M (from 2019) free Android apps, respectively. Overall, we found that the average number of Java libraries per app increased from 7.72 in 2014 to 62.38 (800% increase) in 2019, which adds to the storage pressure. Also, the average number of classes per app increased from 1.76K in 2014 to 5.16K in 2019, which signifies 3x more code in modern apps. These classes include all Java code: app-specific proprietary code as well as generic third-party Java libraries.

The detected libraries suggest that apps rely on third-party SDKs for a host of functionality, such as tracking analytics, advertisements, and various utilities (e.g., debugging). Table 3 summarizes popular SDK types we found. The number of apps containing in-app advertisements and social media plugins increased by 2x and 2.5x, respectively in five years.

SDK Code Bloat. Third-party SDKs implement generic functionality. For example, a cryptographic library may implement multiple encryption algorithms. However, only a subset of SDK *methods* is used by developers, even though the entire third-party SDK is imported and distributed with the app, resulting in code bloat. Our LibScout analysis reveals that no method was used for 13.28% Java libraries in 1.7M apps from 2019. Furthermore, less than 40 methods were used for 40% of SDKs.

Developers also add multiple third-party SDKs for the same functionality. For example, many apps integrate both Google and Facebook social media SDKs to offer easy account login functionality. However, the user may only use their favorite, rendering others unused.

Native libraries. Besides Java libraries, Android apps also contain C/C++ native libraries compiled to machine code (see §2). We found that both the number of apps containing native libraries and the average number of native libraries per app increased by 10% in five years. Figure 5b shows the distribution. Specifically, 290K (24.42%) of 1.1 million free Android apps contain at least one native library in 2014. Average number of native libraries across those 290K apps was 3.2. In contrast, of 1.77 million free apps from 2019, 669K (37.7%) contain native libs. In 2016, 640.7K (32.52%) of 1.97M free apps contained at least one native lib, with an average of 6.3 native libraries per app.

Universal Apps. We also found cross-architecture `x86/x86_64`, `ARMv5/v8`, `MIPS/MIPS64` native libs, consuming an average of 10 MB per app that will never be accessed on `ARMv7` devices. The number of such libraries increased significantly in five years: 2.4M and 4M were found across 640K and 669K apps with native libraries in 2016 and 2019, respectively. The average number of `ARMv7` libraries across 669K apps from 2019 was 4 per app; whereas, redundant libraries grew to 6. Table 5 provides the breakdown.

This trend suggests that despite Google's guidelines to create smaller and device-specific optimized apps [22, 26, 27], developers create universal apps. It is important to note that ARM devices are backward compatible; that is, a native library compiled for `ARMv5` or `ARMv7` architecture is compatible with an ARMv8 Android device. Similarly, x86 Android devices can emulate ARM

instructions using a binary translator [46] at runtime, trading performance and battery. However, developers include individual CPU-specific libraries to provide improved performance and experience.

Unstripped native libs. By default, the Android development environment (Android Studio) removes (strips) all debug information that may be embedded in native libraries by compiler. App developers may disable stripping during app development phase for easier debugging at the cost of generating bigger libraries, and re-enable for a lean release. However, 25% of the apps with native libraries we analyzed from 2014, contain at least one *unstripped* library, adding to the storage pressure. In 2019, the number of unstripped libraries fell to 13.2%.

In addition to debug info, native libraries also contain symbols for the linker to resolve references at runtime. By default, the compiler adds symbols for all *exported* functions that may potentially be invoked at runtime. However, it needlessly increases the library size, particularly if the app only invokes a small fraction of such functions at runtime. We found that about 88% of native libraries have more than 50 exported functions that further add to the library size.

CPU	# Apps (%)		# Libs (%)	
	'14	'19	'14	'19
ARMv5	63.08	44.10	32.40	8.86
ARMv7	74.75	94.35	53.16	38.41
ARMv8	0.04	42.39	0.02	12.95
MIPS	9.38	23.46	2.93	3.92
MIPS64	0.04	9.65	0.02	1.42
X86	24.39	67.94	11.16	24.85
X86_64	0.05	36.26	0.03	9.58
OTHER	0.35	0.08	0.28	0.02

Table 5. Change in num of Android apps supporting multiple CPU types over years.

App	# Activtis		Size (MB)		# Libs	
	'14	'19	'14	'19	'14	'19
AmznKdle	65	123	30.83	37.29	2	54
AgryBrds	12	25	47.26	99.00	3	8
Dropbox	55	130	28.24	50.62	4	12
LinkedIn	53	104	28.07	29.44	2	9
Facebook	208	563	25.20	51.58	44	140
Tencent	489	900	27.58	105.70	58	212
Twitter	98	193	14.29	20.77	20	48
Yelp	138	263	14.60	24.30	10	80
Fitbit	87	250	16.63	57.44	0	208
Starbucks	6	56	7.80	42.75	12	92

Table 6. Change in sizes, num of Activities and native libs across top 10 (>1B installs) Android apps over years.

Duplicate code. As mentioned above, apps use third-party SDKs to implement common features. However, such common SDKs lead to code duplication. For example, 28% to 90% of apps that we analyzed from 2019 contain the same version of at least one library package from `com.android.support`. 6.6% of apps contain the same version of OkHttp library. Duplication can also occur when multiple apps from the same vendor are installed. For instance, Facebook and Instagram apps contain the same version of `libvideo.so` and `libfb.so`.

To find duplicate native libraries across all apps from 2019, we extracted libraries from app APKs using `apktool`, and calculated `md5sum` for each library. We only analyzed `ARMv7` libraries as most (94.35% of 669K) apps contain such native libraries (see Table 5). We found 2.1 million (91.84%) `ARMv7` native libraries across all apps contain at least one duplicate library, occupying a total of 4.48 TB of redundant space. Therefore, a simple file-level deduplication scheme could save 4.48 TB across 2.1 million apps (i.e., 2.13 MB per app on average).

In-app products. Another source of increase in app sizes is the use of paid in-app products for monetization. Developers allow users to purchase additional features or related products from within their apps. For instance, Disney Pet Palace book allows users to purchase more pets. Similarly, games allow users to purchase advanced levels. However, such features will only be accessed once paid for and unlocked. We found that about 118K apps from 2019 support in-app purchases, consuming unnecessary storage space.

Opaque Binary Blobs. Developers can supplement the app APK with two monolithic OBB files, each up to 2 GB (see §2). We inspected OBB files from 150 large (> 100MB) GooglePlay Android apps across all categories with at least 1M downloads to understand how developers use OBB files.

We found that OBBs are zipped archives of multiple small files. Games and digital books developers leverage OBBs to hide proprietary implementation (e.g. 3D textures) [24]. For example, 1497 MB OBB from Asphalt 8 Airborne racing game contains 11K distinct objects — binary (e.g. shaders, textures), text/xml, and multimedia objects (e.g. images) — ranging from 100 bytes to 13.47 MB.

Developers also abuse OBB files to build universal apps, bundling auxiliary assets catering to more than one demographic region (e.g. text/audio in various languages) and device type (e.g. ARM vs. x86, phone vs. tablet). For example, OBB archive from Nightmares from the Deep game contains 445 icons and images containing non-English content (totaling 10 MB).

To summarize:

- Modern apps pack more features and third-party SDKs/libraries. Number of features in top apps doubled in five years, resulting in bigger apps.
- Most apps are universal, containing cross-architecture native libraries for improved performance on every device type at the cost of more storage.
- Apps also contain paid features that are only unlocked once the user purchases them. Nevertheless, such features consume storage space regardless.
- Installation of apps that use common SDKs or are from the same vendor result in duplicate SDKs and libraries, consuming redundant storage space.

5 INSTALL-TIME BEHAVIOR.

Apart from APK and OBB files that are downloaded from GooglePlay, an Android app may download or create additional files during the installation process, leading to additional storage consumption. To understand such additional install-time storage requirements, we now present results from our analysis of fresh installation (no usage) of apps.

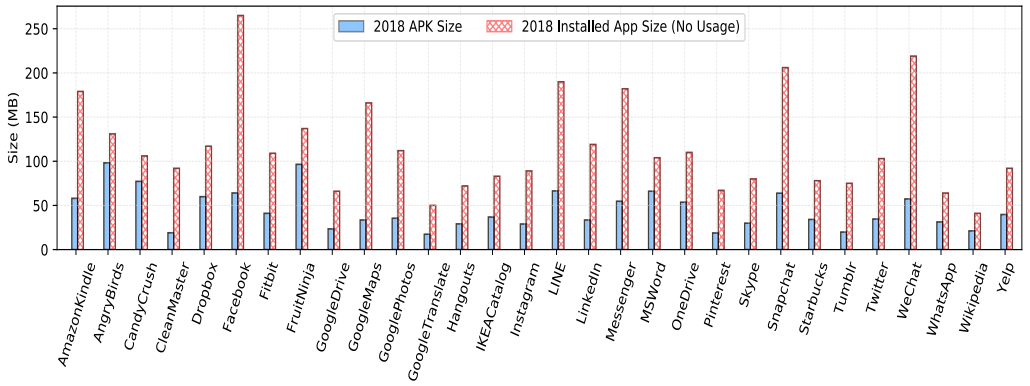


Fig. 6. *Install-time storage consumption analysis of top (> 10M downloads) Android apps collected in Aug, 2018. Upon installation, apps expand to further consume additional storage, primarily due to creation of pre-JITed OAT files and decompressing libraries to offer better performance.*

Questions. During this study, we sought answers to the following questions.

– *How much storage do apps consume immediately upon fresh installation (no usage)?*

– *Do app installations consume additional storage beyond app sizes? If so, how much and what causes this additional storage consumption?*

Dataset. We analyzed top 30 GooglePlay apps from 2018, each with at least 5 million downloads.
Methodology. We performed fresh manual installation of each of the apps in our dataset on a LG Nexus 5 device, and used Android Debug Bridge (ADB) shell scripts to inspect sizes of files in app storage areas (see §2) without using launching the app.
Findings. Figure 6 shows our findings. We found that upon installation, 27 apps expand to further consume at least 1.5x storage space of their respective package. Facebook app, in particular, expands from a 64 MB package to consume sizable 265 MB (over 4x increase) when installed.
OAT creation. The primary reason for additional storage consumption during installation is creation of additional files. During the installation process, each `classes.dex` in app APK is parsed by the ART/Dalvik JVM. The resulting pre-compiled `classes.dex` bytecode, called OAT, is stored on the device so that any future app execution occurs without the need for further JIT compilation, resulting in better performance and user experience at runtime. Post installation, the compressed APK file is also stored on the device, resulting in two copies of each app `classes.dex` file–one in the APK and another in pre-compiled OAT file.
Unzipped libraries. During installation, native libraries are also extracted on the device for faster I/O at the cost of more storage consumption. For example, Facebook APK extracts over 120 native libraries. Once installed, the app extracts additional 70 native libraries and Java DEX files from zipped asset resources, namely from `libs.xzs` archive, consuming additional 50 MB.
Additional persistent files. We found that apps also create `databases` and xml files under `shared_prefs` upon installation to host app settings. Apps that need auxiliary OBB files as a part of their functionality, download such files upon first activation (usage) of the app, not during installation. A few apps also download or create additional resources separately as needed. For instance, Facebook app preallocates storage space by creating a large 20MB file.

> **To summarize:** Upon installation, Android apps create optimized code and additional files to provide high performance at the cost of more storage consumption.

6 USER STUDY

To gain insight into the runtime storage consumption behavior of today's mobile apps, we carried out a study with Android users in the wild. The purpose of the study was to answer the following:

– **Storage consumption:** *How many apps do users install on average? How much storage do apps consume after weeks/months of usage? How much is consumed by user data (e.g., photos, videos)?*
– **Storage usage:** *How many apps (and features) do users interact with daily on average? How many installed executables (e.g., libs) and auxiliary files are actually used actively?*

6.1 Study methodology

Participants. Our study was conducted with participants recruited via PhoneLab [45], a mobile testbed at the University of Buffalo. PhoneLab allows researchers to deploy and test Android changes with 100–300 participants, who are university affiliates (i.e., faculty, staff, students) across diverse groups of age, gender, and occupation [55]. We provided our Android changes to PhoneLab administrators, who then made our study available to their participant pool to volunteer.

Our initial sample consisted of 231 PhoneLab participants. However, data from 91 participants were removed due to substantial missing data. The final sample used in this study consisted of 140 participants, who provided complete data across 70 days.

OS and device. PhoneLab provided a 16 GB LG Nexus5 device to each participant, which they use as their primary smartphone for the study duration, allowing us to collect representative usage data. The device ran Android Lollipop version 5.0.1, customized to collect data from the Android `Logcat` memory buffer [28]. Data collected was stored on the device and securely uploaded to the PhoneLab servers over HTTPS every night when connected to WiFi.
Privacy. To ensure the privacy of participants, PhoneLab anonymized all device identifiers when collecting data. Approval from the Institutional Review Board was obtained prior to the study.
Tracing tool. Smartphones are highly personal consumer devices. As such, each user persona type (e.g., gamer vs. non-gamer) is likely to have a completely different device usage patterns.

Therefore, we built a novel context-aware storage tracing tool, called CosMos, that not only collects low-level file system events, but also logs apps being used, along with various device contextual attributes (e.g., location). It consists of a daemon process and a native library that is transparently loaded into each Android app using `LD_PRELOAD` when the app is launched. Upon initialization, the library hooks relevant file system APIs (e.g., open, read) in `bionic` C library by registering wrappers functions that intercept and log file system requests. Table 7 summarizes APIs we hook and data we collect. This design requires no change to individual apps or the Android framework. We deployed CosMos on each participant's device as a background system service that posts data to the Android Logcat buffer, which is collected by the PhoneLab framework.

We have designed CosMos particularly for lightweight continuous data collection on mobile devices. For instance, we avoid periodic polling for device context in order to minimize the runtime overhead; instead, our daemon process subscribes to various Android services to collect contextual attributes. Our wrapper library allocates a large user-space memory buffer in each app to allow bulk posting of file system events and minimize IO overhead. Figure 7 shows its performance.

In contrast, existing file system monitoring tools such as inotify [44] pose high memory and performance overhead due to recursive watchpoints on each dir [14]. User-space file system tracing based on FUSE [59] (e.g., loggedfs [20]) incur up to 4x performance overhead [65]. In-kernel tracing frameworks (e.g., ftrace) log system-wide low-level file system requests, and thus require large kernel memory buffers. Also, requests from Android system services must be filtered to only capture storage accesses from apps.

Nevertheless, CosMos fails to log memory-mapped I/O traffic as no file system API (read/write) calls are made. Nevertheless, eBPF [38] kernel extensions could be leveraged to selectively capture such low-level requests from user space [9, 10]. We leave this as future work.
Platform changes and data collection. We collect the following data for all the participants.

(1) Detailed low-level file system activity traces, and relevant parameters (e.g., file path, size, offset).
(2) Detailed device and app usage statistics (e.g., app `Activity` invoked by the user). We made a small change to the core Android framework to track app Activity usage.
(3) Stateful device updates, such as network connectivity changes, and storage reclamation events.
(4) Multiple spatio-temporal contextual attributes such as device location, time of the day, and physical activity of the user (e.g., on foot, in vehicle, etc.). Table 9 lists them.

The data we collected can be accessed publicly from PhoneLab [49]. We focused only on the storage consumption of the `/data` partition because it is fixed in size, shared among all apps, and also hosts user data such as videos, pictures, and documents (see §2 for details).

6.2 Findings.

Here we present our findings by analyzing the data collected from our user study.
App storage consumption. We found that on average a user has 122 apps installed. 91 of those were pre-installed system apps (i.e., apps installed in the `system` partition). Such pre-installed

API	Input Parameters
Read	File descr, Offset, Length
Write	File descr, Offset, Length
Open	File descr, File path
Mmap	File descr, Access perms

Table 7. *Bionic C APIs we hook for collecting file system traces.*

Device Event	Android Service	Description
Bluetooth	BluetoothAdapter	Pairing Updates
Battery	BatteryManager	Battery Status
Cellular	ConnectivityMngr	Cellular Info
Wireless	WifiManager	Wifi SSID
Storage	StorageServer	Storage reclaim

Table 8. *Android system services we subscribe to for collecting contextual attributes.*

Attribute	Android Service	Context
Timestamp	System Time	Day, Time
Location	LocationServices	Location
UserActvty	ActvtyRecognition	Walk, Drive

Table 9. *Summary of Contextual data collected.*

Vendor	Example apps
AOSP	Clock, Email, Phone, Settings
Google	Maps, Docs, Photos, GMS, Play
LG	Update, SprintHiddenMenu
Qualcomm	redbend.vdmc, qcrilmsgtunnel

Table 10. *Preinstalled system apps.*

User	Apps Installed			Apps Used		# Features Used (%)	
	Total	System	User	Avg Daily	Overall	Avg Daily	Overall
A	135	97	38	17	45	1.36	7.24
B	113	95	18	11	32	1.31	5.53
C	116	93	23	15	34	1.51	9.93
D	123	95	28	8	23	1.41	5.56
E	121	97	24	10	37	0.84	4.94
F	130	97	33	16	36	1.14	4.19
G	107	96	11	15	31	1.34	6.05
H	115	91	24	18	37	0.71	3.82
I	166	102	64	13	51	0.80	5.68
J	124	92	32	17	39	1.15	5.32
K	122	90	32	12	34	0.99	3.94
L	129	96	33	12	34	1.20	7.38
M	132	98	34	19	59	0.99	5.90
N	145	98	47	11	35	0.45	2.59
O	146	99	47	15	41	1.05	4.41
P	122	96	26	12	35	1.20	8.50
Q	131	91	40	18	30	0.79	5.33
R	145	99	46	8	41	0.43	3.16
S	121	89	32	14	50	1.09	6.80
T	117	92	27	10	25	0.26	4.26
U	123	94	29	16	45	1.26	6.40

Table 11. *Snapshot of device storage consumption across 21 users. On average, each user has 122 apps installed. 91 of those were pre-installed system apps. However, only a small fraction of apps, features, and native libraries are actively used daily.*

apps are installed by phone vendors, manufacturers, or carrier providers. We list a few examples in Table 10.

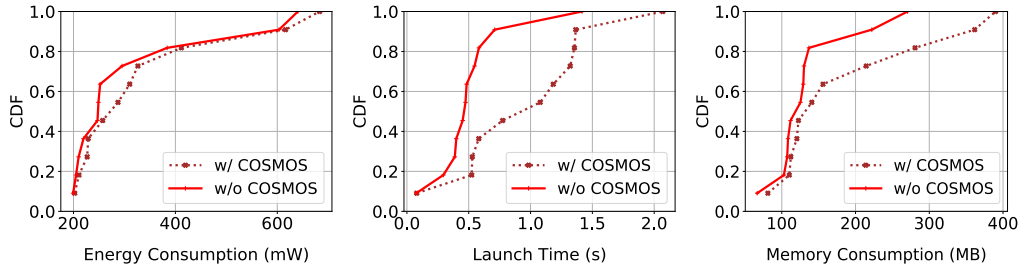


Fig. 7. *Performance of COSMOS across top 12 apps (see §5) as measured by app launch latency and battery use. Evaluation setup and methodology are the same as in §7.1.*

Figure 8 shows the breakdown of storage consumption of the `/data` partition for 20 different users chosen at random. The space consumption of an app is calculated as the sum of sizes of its individual files found in various designated internal as well as external app storage locations (see Table 1). Space consumed by the OS is calculated based on the files present in the internal `/data` partition that do not belong to any app. User data such as documents, pictures, and videos are identified by their file type and location in primary external partition (or `/sdcard`). For instance, on Android devices, pictures taken using the digital camera on the device are stored under `/data/media/DCIM`, whereas apps save picture messages in `/data/media/Pictures`. Similarly, audio files are stored in `/data/media/audio`.

As seen in the figure, apps consume almost 50% of the total storage across users, on average. Apart from user data (e.g., pictures/videos) that is commonly known to consume significant storage space due to high-quality multimedia content, we found that over time modern apps also can occupy a significant storage space. For instance, app consumption exceeds 60% for user-H and user-N, even though none of the apps are over 1 GB, suggesting data accumulation over time.

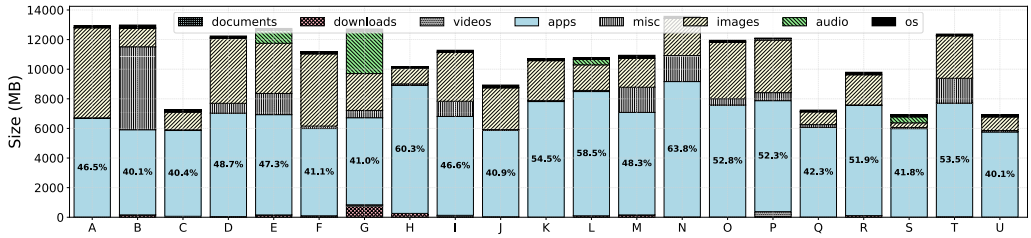


Fig. 8. *Breakdown of `/data` storage partition snapshot of randomly selected 21 users. The space consumption of an app is calculated from the sizes of its various individual files found in various internal as well as external storage areas (see Table 1). `misc` includes files in public `/sdcard` dirs.*

Storage abuse. To get more insights into what causes apps to accumulate data over time, we further studied the storage consumption of apps as per various designated app storage locations on Android devices (see Table 1). Figure 9 shows the breakdown of storage consumption. We found that on average 20% and 11.86% of the available storage space is consumed by app caches (C) and files (F), respectively across 21 users.

This suggests that apps abuse persistent storage by creating files as needed and frequently caching or prefetching content from the Cloud to improve app runtime performance and offer a streamline user experience. As a result, through the daily use of apps, mobile devices accumulate large amounts of data and quickly exhaust storage space.

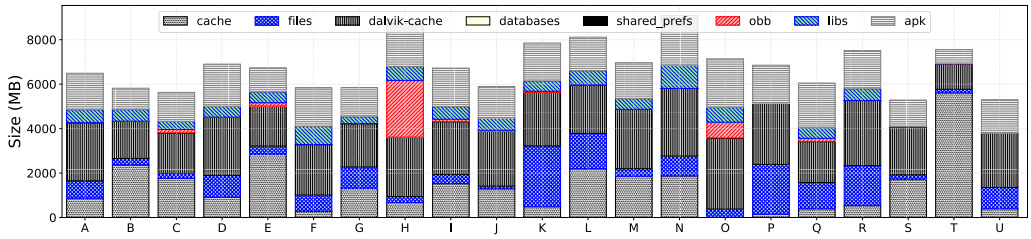


Fig. 9. *Further breakdown of storage consumption of apps in Figure 8 as per the app storage areas (see Table 1), namely files (F), cache (C), databases (D), shared preferences (S), native libraries (L), OBB files (O), APK files (A), and Dalvik cache (V) shows that apps frequently persist data on the device (e.g., caches, files, db, libs) to provide low latency and streamline experience, particularly under fickle network conditions. Cache dir space is blindly reclaimed when the device runs low on storage regardless of device/apps usage pattern. In contrast, the space occupied by other app persistent storage areas (e.g., files, DBs) is not reclaimed until the app is manually deleted.*

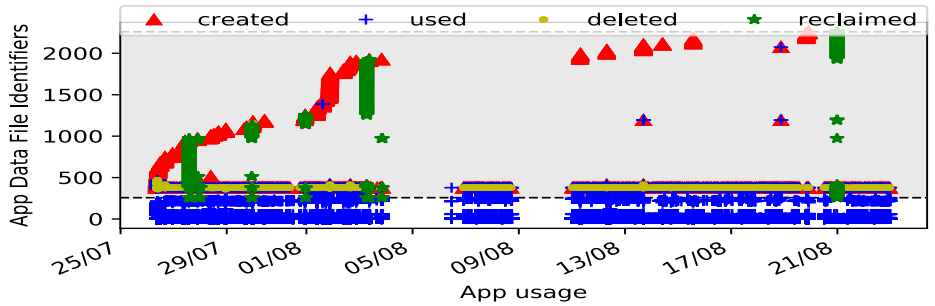


Fig. 10. *Fitbit app storage consumption over time for user-O. The area shaded in grey represents the private cache dir. Cache files are created (shown as red triangles) and accumulated over time. However, only a small fraction of those files are actually used (shown in blue plus symbols), and only a few are deleted by the app (shown in yellow dots). When the device storage consumption reaches 90%, space occupied by cached files is reclaimed (depicted by green stars). Due to blind reclamation of cached data, files in the working set are recreated on demand, yielding poor performance.*

Yet, apps are not constrained by storage consumption limits [25]. When the overall device consumption reaches 90% threshold, all app caches are automatically cleared by Android storage service. However, blind removal of cached working set not only affects the performance of frequently used apps, but also for such apps the caches are quickly repopulated on next usage. For example, for user-O storage space occupied by Fitbit cache is cleared, but is repopulated the same day as seen in Figure 10.

Apart from frequently caching temporary data, We found that apps also freely hoard persistent data as needed, such as analytics to track user engagement, crash reports for debugging, and advertisements (ads) for monetization. Developers use a number of third-party SDKs for ads and collect user analytics (see Table 3). The number of apps using advertisement SDKs doubled in 2019 from 2014 as reported in §4. Such SDKs allow users to build custom, interactive (video), and engaging ads for monetizing and promoting their apps. Since some ads may require more bandwidth, and can take time to load, they are persisted on the device for a period of time to avoid re-fetching. Persisted ads are also shared across apps using the same advertisement SDK by installing them on the public `/sdcard` partition.

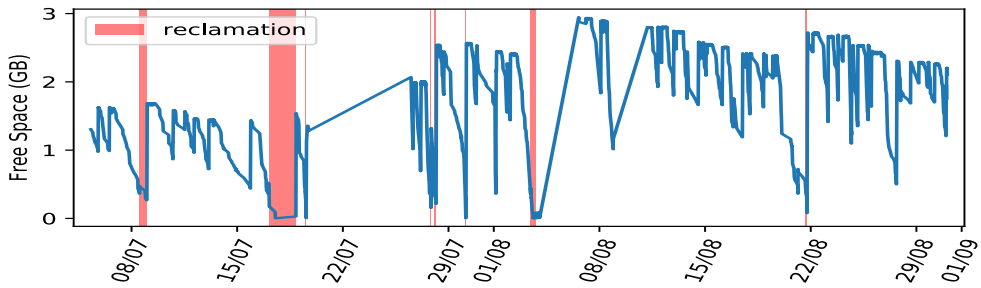


Fig. 11. *Free space available on device over time for user-L. As the storage consumption reaches 90%, the system reclaims storage space by blindly deleting app caches. However, due to extensive usage, app caches are repopulated quickly. Red regions represent reclamation periods.*

During our study we found that free games, in particular, download and persist high-definition in-app video ads. While app caches are temporary and reclaimed automatically, persistent data is never reclaimed. Removal requires manual deletion or app uninstallation. We detected several old video (mp4) and image (png) ads from AdColony, UnityAds, and Chartbooster across user devices. Hoarded crash logs and analytics data were also found.

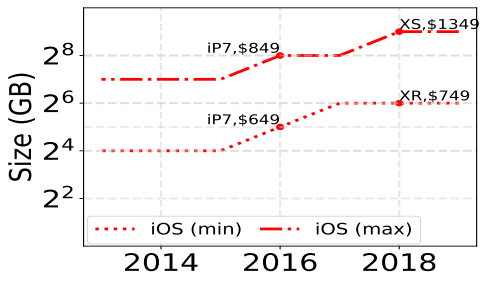


Fig. 12. *Min storage on iPhones grew from 16 GB in 2014 [68] to 64 GB in 2019 [69], while the average capacity on Android remains 54 GB [15].*

Overall	**Avg (%)**
Apps	53.70
UserData (Pic/Vid)	30.13
Miscellaneous	16.17
App	**Avg (%)**
Code (APK, OAT, OBB)	73.12
Cache (/cache)	14.23
Data (Files, XML, DBs)	12.65

Table 12. *Different sources of mobile storage consumption.*

Unused apps/features. While our static analysis (§4) shows that modern apps pack 2x more features, findings from our user study shows that app usage follows the *Pareto Principle*; i.e., only a 10% of app features and files are actively used, on average. We found this behavior to be consistent across most of the study participants. Table 11 summarizes our findings. This usage behavior suggests that modern apps are heavily bloated: 90% number of app executables and third-party libraries that are persisted on device and optimized for performance during installation (see §5), are not actively used.

For example, apps integrate multiple third-party social media SDKs (e.g., Facebook) to offer easy account creation functionality. However, only one is used; code of remaining SDKs is never executed. Many free apps (e.g., games) offer in-app purchases of additional features. Such features will only be accessed once paid for and unlocked. Similarly, only a fraction of native libraries are used. For example, Facebook APK installs over 100 native libraries. However, we found that for most users less than 50 such libraries were used over the course of 70 days of the app usage. Similarly, LinkedIn app installs four native libraries. However, only three were used for all users.

Furthermore, app usage is highly correlated to user context. For example, user-H uses the *Maps* app only once a week, every Sunday. Similarly, user-N uses the *Yelp* app only on weekends.

Redundant metadata. Functionally similar apps operating on the same data create their own copy of metadata, resulting in unnecessary storage consumption. For example, various photo viewing, editors, file browsers, and Cloud-based personal storage apps (e.g., Dropbox) allow users to browse stored docs, photos, videos. For better performance, each such app generates their own thumbnails of all videos and photos, resulting in redundant metadata that wastes storage space.

To summarize:

- A significant chunk of device storage is consumed by apps, which includes a number of pre-installed system apps. However, only a small fraction of apps and features are actively used, resulting in sub-optimal use of device storage.
- Apps are not constrained by storage quota limits, and freely consume persistent storage by frequently caching data and creating auxiliary files as needed.
- All app caches are automatically and blindly deleted in Android under high storage pressure, which impacts the performance of actively used apps.

7 MOBILE STORAGE MANAGEMENT

In this section, we first discuss the need for an efficient storage management system for mobile devices, and then present future opportunities and challenges.

Storage demand vs. capacity. Our longitudinal study shows that modern mobile apps have evolved as large feature-rich monolithic packages: an app from 2019 packs 2x more features and consumes 2x more storage than that from 2014 (see Table 6). Furthermore, such storage-heavy apps are not limited to games or e-books, but span across multiple categories. We believe that this trend will continue, particularly as the mobile technology (e.g., 5G) evolves and richer, more immersive apps supporting Augmented Reality, Artificial Intelligence, and 4K graphics are introduced.

To accommodate increasing demands, smartphones have seen growth in storage capacity. For instance, the minimum storage on iPhones quadrupled, from 16 GB in 2014 [68] to 64 GB in 2019 [69]. Figure 12 shows the trend. However, low-end budget devices are still prevalent. According to Counterpoint Research [15], the global average storage capacity of smartphones sold in 2019 was about 54 GB and 134 GB for Android and iOS smartphones, respectively. Since Android holds about 86% of the global market [37], we deduce that conservatively at least 43% of the smartphone users own low-end devices, with less than 64 GB of storage in 2019. Such devices, more common in developing countries [58], severely limit user experience [54, 57, 64]. Therefore, similar to battery and cellular data, storage is an increasingly critical budgeted resource on mobile devices.

Storage capacity vs. bloatware. Although users can choose to pay a premium price for additional mobile storage [69], our analysis shows that mobile OSes and apps are not optimized for storage. A high percentage of storage is consumed by OS and pre-installed bloatware (shown in Figure 8). Modern apps are monolithic feature-rich and universal packages that contain: a) 3x more Java code, all of which is precompiled to native machine code for best runtime performance, trading at least 50% more storage per app (see Figure 6), b) cross-architecture native libs for best performance on every device type (see Table 5), and c) generic third-party libs, which are only partially used (see §4). As such, today's apps are heavily optimized for performance at the cost of higher storage consumption. While high performance does matter for apps that are frequently used, our user study shows that only 10% of apps/features are actively used (see §6). Furthermore, developers freely abuse persistent storage to cache data for streamlined user experience, host ads and user analytics for monetization.

7.1 Design space: challenges and opportunities

In the light of aforementioned findings, there is an increasing need for an efficient storage management system on smart mobile devices. Here, we discuss a few design opportunities and challenges.

1. Cloud storage. The cloud offers a naturally attractive solution to mobile storage constraints. In fact, iOS v11 can offload infrequently used apps from iPhone storage to iCloud [5]. Currently, only app packages are offloaded, which are re-downloaded on demand from the app store. Whereas, app data (e.g., login credentials) are retained on device for future stateful accesses.

However, this design, may not be optimal if entire app packages are downloaded repeatedly on demand only to access a handful of app features, which is a typical usage pattern as per our findings in §6. Additionally, blindly offloading all app data will also offload unwanted files, such as crash logs, user analytics, video/image ads that are hoarded on the device (see §6), merely shifting the problem to managing the cloud storage; users still need to perform manual management to delete data or pay more for usage.

2. Storage quota. A typical approach to limit and manage storage consumption is to introduce per-app storage quotas. While quota limits may prevent apps from abusing storage to freely cache and hoard data (e.g., analytics, ads), identifying appropriate limits is challenging.

Static limits will not scale as more functionality is introduced and apps grow in size. Additionally, developers may have to redesign apps to work within the quota limits without any discernible impact on user experience.

3. Elastic storage. Elastic quota model [72] overcomes these shortcomings by hard-limiting only persistent consumption and allowing apps to temporarily grow beyond the quota limits depending upon the available storage space. Temporary space consumed beyond quota is reclaimed under high storage pressure. Nevertheless, enforcing elastic quota requires one to identify when and what data must persist on the device or be reclaimed.

One way to address this is to let each app implement its own logic to identify and reclaim unwanted data. However, it will result in duplicate effort across apps, and place the burden on developers. We assert that mobile OSes must perform centralized and transparent management of storage resources on behalf of the user. For example, a core storage management service that automatically reclaims space occupied by unwanted apps/features would be useful to the entire community of app developers.

Our user study shows that app usage is highly correlated to user context and not all apps are equally important to all users. Specifically, we found that the relative importance of apps not only varies with the user (e.g., gamer vs. non-gamer), but also varies for the same user depending on the context (e.g., home vs. office). Table 11 captures this behavior. Therefore, for efficient use of mobile storage, novel techniques must be developed to learn app usage behaviors of the user and perform storage management according to the user's context-sensitive storage requirements.

Modern mobile OSes already host a personal assistant that tracks user's device usage habits and provides context-based app recommendations. This functionality could be extended to also learn about user's everyday storage needs, and build working set storage profiles to automatically identify unwanted apps/features, which could be reclaimed for more efficient use of storage.

Besides relying on the cloud for hierarchical management (i.e., offloading data), which may not always be optimal as mentioned above, multiple techniques such as content adaptation, deduplication, compression, could be leveraged to reclaim contextually unwanted data. Here, we evaluate these techniques to provide an estimate on potential storage saving and performance implications.

a.) Content adaptation. During app installation, Java code is pre-compiled into native code (OAT) for best runtime performance (see §5). By default, all Java methods are compiled, which results in storage consumption of at least 1.5x the size of app APK. However, our findings in §6 show that

only a small fraction of app features are used. For example, in-app purchases are only accessed if the user has paid for and unlocked them. Similarly, not all Java methods of third-party SDK are used (see §4). Depending on the functionality, only a few Java classes and their corresponding resource files (e.g., images, sound files) are accessed. As such, optimizing all app features by trading in more storage leads to inefficient use of storage resources.

Performing pre-compilation on only a subset of Java classes (e.g., frequently used features) can save storage by reducing the size of the generated OAT file, and potentially offer similar performance. Further savings could be achieved by temporarily deleting OAT files of rarely or infrequently used apps and falling back to regular (non-optimized) executable code, thereby freeing up at least 50% of the APK storage space per-app. Figure 13 shows storage savings achieved under adaptive compilation.

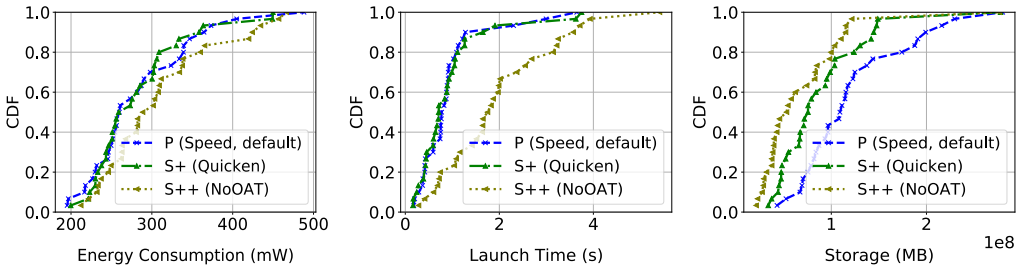


Fig. 13. *Resource trade off in OAT generation as measured across top 30 apps in Figure 6.*

To evaluate expected storage savings and performance impact from adaptive OAT generation, we use the same set of 30 top GooglePlay apps as in §5. All experiments have been carried out on a LG Nexus 5 device, running Android v6.0.1 and Linux kernel v3.4. The power measurements are taken using the Trepen profiling app from Qualcomm [51]. We first install fresh copies of apps. Then each app is started once manually since the first launch may perform necessary initialization (e.g., account creation, fetch/create additional files, cache data) and may take longer than subsequent launches. Once all the apps have been initialized, we reboot the device and perform scripted launch of each app 3 times and report average numbers. Previous instances of apps are force-killed before each run to make sure that cold performance numbers are captured. We disabled device network connectivity to eliminate external noise (e.g., battery drain from fetching data over the network) Inspired by prior works [71], the app launch time reported is simply the total time taken to display the app main activity as measured by Android `ActivityManager` framework.

Figure 13 shows the CDF of cold app launch times, battery consumption, and storage savings as measured under following settings:

– `Speed` pre-complies all Java code to native for best runtime performance (default Android setting),
– `Quicken` [23] optimizes some Java code to get better runtime interpreter performance, and
– `NoOAT`, which decompresses Java methods on the fly from the zipped app APK in memory and relies exclusively on runtime interpretation of code.

On average, we see about 25 MB (21.34%) reduction in the size of OAT file per app for `Quicken`, with almost no latency or battery drain. This is because the generated code is optimized for runtime interpretation. Whereas, `NoOAT` offers more savings, about 50 MB (or 42.78%) per app on average, as there is no optimized OAT file anymore. However, these storage savings come at a small cost in terms of app latency (avg 1 sec) and battery drain as shown in Figure 13 due to on-the-fly decompression of code. We believe this overhead is acceptable to access infrequently used apps.

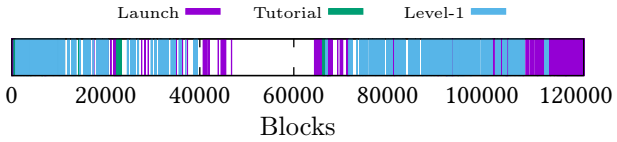


Fig. 14. OBB accesses across three different levels

Similarly, compression can also be employed to transparently reclaim storage space occupied by persistent data of rarely used apps. However, on-the-fly decompression of data will incur battery overhead and runtime latency as in case of `NoOAT` above. Since mobile devices work with limited resources (e.g., battery) and high app latency can degrade user experience [60], an efficient mobile storage management system must carefully balance multiple concerns.

b.) Code deduplication. Common files across apps can be consolidated using deduplication. For example, we found 2 of the top 30 apps use same the version of Joda-Time SDK to implement timezone functionality. Microsoft Skydrive app and Fruit Ninja game contain the same version of `crashlytics` SDK native library, each consuming 121 KB and 185 KB of storage space on ARM and x86 architectures, respectively. Similarly, Dropbox and Snapchat apps contain the same version of `librsjni` library, each consuming 50 KB. Overall, we found that a total of 1 MB (0.85%) per app could further be saved by simple file-level deduplication of native libraries found in the APKs of top 30 apps. Deduplication at finer granularity (e.g., block-level) could potentially yield additional storage savings.

However, online (synchronous) deduplication scheme that hashes every I/O block while it is being written to disk can negatively impact the device battery life and user experience on mobile devices. Therefore, offline (asynchronous) deduplication is more suitable for mobile devices. Furthermore, opportunistic offline deduplication can be performed when the device is being charged.

c.) Deletion. Our findings show that apps freely use persistent storage to offer streamlined user experience by frequently caching data. Since cached data is temporary and could be fetched (from cloud) on demand, additional storage savings could be achieved by simply deleting caches of infrequently used apps. However, careful app profiling must be done so as to not delete the app working set files or on-demand downloading of such data may drain the battery, incur cellular data usage, and cause runtime latency.

Similarly, unnecessary cross-architecture `x86` and `MIPS` native libraries on `ARMv7` devices could be safely deleted as they will never be used.

Savings from OBB files. We monitored accesses to OBB files during app usage by leveraging file system tracing hooks, and found that different objects inside monolithic OBB archives are accessed at different times, depending upon usage. For instance, games contain multiple levels. Every level accesses only a few objects (associated with it) to display on the screen. Figure 14 shows accesses to a 475 MB OBB archive from three different levels of Nightmares from the Deep, an adventure game. As shown, a markedly different chunk of the OBB is accessed by different levels of the game.

Each game level demands different playing skills for users ranging from beginners to experts. As such, not all levels are accessed simultaneously. Depending upon the user's expertise, different levels and features associated with them will be accessed at different times. For inexperienced users, level with the highest difficulty may even never be accessed. Similarly, experts may no longer access initial levels if the player is far through the game.

Similarly, due to the complexity and graphical richness, these large apps contain interactive tutorials for improved user learning and engagement. Some apps, such as adventure games and digital books targeting kids, also contain story narratives to build context and provide background

information. However, these tutorials and narratives may never be accessed once the user gets acquainted with the features. Therefore, depending upon usage, a significant chunk of storage occupied by OBB files could be freed and downloaded again from the app store when needed.

Role of app developers. Developers can follow Google's guidelines and create smaller device-optimized apps [22, 26, 27] as opposed to creating bigger universal apps. Additionally, developers can modify Android compiler flags (e.g., include `-fvisibility=hidden` for GCC) to reduce the number of unnecessary debug symbols, generating smaller libraries.

8 RELATED WORK

The following prior works compare closely to the work presented in this paper.

Study of mobile apps. A number of prior works have analyzed mobile apps and app store data for app popularity/download patterns [41, 43, 48, 70], app ratings/reviews [62], security issues [13, 16, 19, 50, 74], identifying app clones [42, 67, 73], and third-party code reuse [7, 19]. For example, Petsas, et al. [48] studied app download patterns, popularity trends, and development strategies in the mobile app ecosystem. Xu, et al. [70] used cellular network traces to study diverse usage behaviors of mobile apps. Tian, et al. [62] evaluated mobile apps in terms of code complexity, Android API usage, and a number of other factors to distinguish between high and low rated apps.

PlayDrone [66] was the first work that analyzed GooglePlay Android apps at scale and provided insights into app store data, such as app categories, download statistics, and security issues. CredMiner [74] uses app decompilation and program slicing to detect apps that leak developer credentials. OSSPolice [19] focused on identifying license violations and security issues with third-party open-source JAVA and native software libraries in mobile apps. Similarly, LibScout [7] studied security issues with Java third-party libraries. Both the tools detect libraries and correlate them with publicly known vulnerability data to identify impacted apps. However, none of the prior works studied mobile storage consumption, which is the focus of this work.

Study of storage and file systems. Douceur, et al. [17] carried out a large-scale study of file systems. Agrawal, et al. [1] studied temporal storage changes in Windows systems over time such as directory size, file age, and total storage consumption by taking snapshots.

Harter, et al. [33] studied I/O behavior of the Mac OS file system. Whereas, Lee, et al. [40] focused on I/O behavior on Android-based smartphones. In this work, we focus on smart mobile storage, and provide detailed insights into the storage consumption behavior of mobile apps.

Storage management. Harmonium [56] introduced motif abstraction for storage management that allows applications to reconstruct contracted files either by decompressing or fetching over the network. Similar techniques could be leveraged to manage mobile storage.

Elastic quota [72] manages storage by hard-limiting only persistent data and allowing temporary data to grow or be reclaimed depending upon the available storage space. However, it requires the user to identify when and what data should persist or be thrown away. We propose context-sensitive storage management to automatically identify unwanted apps/features on mobile devices.

Context-aware computing. On mobile devices, several studies have looked at using contextual information for app prediction for better manageability of apps on the home screen [8, 36, 75], reduce launch latency [71], target ads [53], and data prefetching [34, 47]. In this work, we propose the use of contextual information for automated storage management on mobile devices.

Other related work. RedDroid [39] performs static analysis of Android apps to remove code bloat. Like RedDroid, we also carry out static analysis of mobile apps to identify various sources of app storage consumption.

WearDrive [35] offloads energy intensive tasks from smartwatches to smartphones. In contrast, we identify multiple techniques to reclaim storage space occupied by infrequently used apps on smart devices.

9 CONCLUSION

Smart mobile devices have evolved as versatile personal digital tools with availability of millions of apps for everyday computing needs. However, dependence on a single device for a host of computing tasks leads to high storage demands. Amid mounting anecdotal evidence suggesting lack of enough storage on mobile devices, a little is known about the utilization of storage by modern mobile apps.

In this work, we presented the first-ever large-scale longitudinal static analysis of millions of Android apps to study the increase in their storage footprint over time and identify various sources of storage consumption. Our results show that mobile apps have evolved as large complex installation packages and heavily optimized for performance at the cost of more storage. Developers create universal apps to reduce the engineering effort and offer best performance on every device type, pack more features in the same app to monetize/engage users, and use a number of third-party libraries. While it clearly signifies evolution of the mobile app ecosystem in terms of functionality, it also implies heavy storage demands. A single app can occupy up to 4 GB of storage space. We believe this trend will continue as the mobile technology evolves (e.g., 5G) and richer, more immersive apps supporting Augmented Reality, Artificial Intelligence, and 4K graphics are introduced.

Although users can pay a premium price for additional storage, our Android storage usage study with 140 participants shows that modern apps are bloated with features. Users typically use only a fraction of apps/features actively and usage is highly correlated to user context. Our analysis suggest a high degree of feature bloat and unused functionality in modern apps, which leads to wastage of storage.

We further show that app developers freely abuse persistent storage by frequently caching cloud data for best performance and streamlined user experience, download advertisements for monetization, and store user analytics as well as debug/crash logs. Through the daily and extensive use, these devices accumulate large amounts of data and quickly exhaust storage space.

In the light of our detailed analysis and insights, there is an increasing need for efficient mobile storage management. In this work, we take the first step by proposing an elastic storage design for smart mobile devices, identify research challenges, and show promising preliminary storage savings from such a design.

10 ACKNOWLEDGMENTS

We would like to thank our shepherd, Dr. Arif Merchant, and all anonymous reviewers for their insightful feedback and suggestions, which substantially improved the content and presentation of this paper. This work was funded in part by NSF CPS program Award #1446801, NSF CNS program Award #1909346, and a gift from Microsoft Corp.

REFERENCES

[1] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. 2007. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)* 3, 3 (2007), 9–es.
[2] AppBrain. 2019. Number of Android applications. https://www.appbrain.com/stats/number-of-android-apps
[3] Inc. Apple. 2008. iTunes preview (app store). https://itunes.apple.com/genre/ios/id36?mt=8
[4] Inc. Apple. 2018. Reducing Your App's Size. https://developer.apple.com/documentation/xcode/reducing_your_app_s_size
[5] Inc. Apple. 2019. If you need more space for an update. https://support.apple.com/en-us/HT203097
[6] Apple, Inc. 2015. Now Accepting Larger Binaries - News and Updates - Apple Developer. https://developer.apple.com/news/?id=02122015a
[7] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and its Security Applications. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*. ACM, Vienna, Austria.

[8] Ricardo Baeza-Yates, Di Jiang, Fabrizio Silvestri, and Beverly Harrison. 2015. Predicting The Next App That You Are Going To Use. In *Proceedings of the Eighth ACM International Conference on Web Search and Data Mining*. Shanghai, China, 285–294.

[9] Ashish Bijlani and Umakishore Ramachandran. 2018. A Lightweight and Fine-grained File System Sandboxing Framework. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys)*. ACM, Jeju Island, South Korea.

[10] Ashish Bijlani and Umakishore Ramachandran. 2019. Extension Framework for File Systems in User space. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. USENIX, Renton, WA, 121–134.

[11] Bintray.com. 2016. JCenter is the place to find and share popular Apache Maven packages. https://bintray.com/bintray/jcenter

[12] Box, Inc. 2015. Box | Free Cloud Storage. https://www.box.com/

[13] Kai Chen, Peng Wang, Yeonjoon Lee, Xiaofeng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. 2015. Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.

[14] J. Corbet. 2017. Superblock watch for fsnotify. https://lwn.net/Articles/718802/

[15] Counterpoint Research. 2019. Average Storage Capacity in Smartphones to Cross 80GB by End-2019. https://www.counterpointresearch.com/average-storage-capacity-smartphones-cross-80gb-end-2019/

[16] Matthew L Dering and Patrick McDaniel. 2014. Android market reconstruction and analysis. In *2014 IEEE Military Communications Conference*. IEEE, 300–305.

[17] John R. Douceur and William J. Bolosky. [n. d.]. A Large-Scale Study of File-System Contents. *SIGMETRICS Perform. Eval. Rev.* (May [n. d.]), 59–70.

[18] Dropbox, Inc. Feb 2015. Dropbox - Your stuff, anywhere. https://www.dropbox.com/

[19] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*. ACM, Dallas, Texas, 2169–2185.

[20] R. Flament. 2019. LoggedFS - Filesystem monitoring with Fuse. https://rflament.github.io/loggedfs/

[21] Inc. Google. 2015. Google Drive - Cloud Storage & File Backup for Photos, Docs & More. https://www.google.com/drive/

[22] Inc. Google. 2019. About Android App Bundles. https://developer.android.com/guide/app-bundle/

[23] Inc. Google. 2019. Configuring ART. https://source.android.com/devices/tech/dalvik/configure

[24] Inc. Google. 2019. Data and file storage overview. https://developer.android.com/training/data-storage

[25] Inc. Google. 2019. Faster Storage Statistics. https://source.android.com/devices/storage/faster-stats

[26] Inc. Google. 2019. Multiple APK support. https://developer.android.com/google/play/publishing/multiple-apks.html

[27] Inc. Google. 2019. Reduce APK size. https://developer.android.com/topic/performance/reduce-apk-size

[28] Inc. Google. 2020. Android Logcat system. https://developer.android.com/studio/command-line/logcat

[29] Google, Inc. 2010. Android Market Client Update | Android Developers Blog. http://android-developers.blogspot.com/2010/12/android-market-client-update.html

[30] Google, Inc. 2012. Android Apps Break the 50MB Barrier | Android Developers Blog. http://android-developers.blogspot.se/2012/03/android-apps-break-50mb-barrier.html

[31] Google, Inc. 2019. APK Expansion Files | Android Developers. https://developer.android.com/google/play/expansion-files

[32] Google, Inc. September 2015. Support for 100MB APKs on Google Play | Android Developers Blog. http://android-developers.blogspot.com/2015/09/support-for-100mb-apks-on-google-play.html

[33] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2011. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, 71–83.

[34] Brett D. Higgins, Jason Flinn, T. J. Giuli, Brian Noble, Christopher Peplin, and David Watson. 2012. Informed Mobile Prefetching. In *Proceedings of the 10th ACM International Conference on Mobile Computing Systems (MobiSys)*. ACM, Lake District, United Kingdom.

[35] Jian Huang, Anirudh Badam, Ranveer Chandra, and Edmund B. Nightingale. 2015. WearDrive: Fast and Energy-Efficient Storage for Wearables. In *Proceedings of the 2015 USENIX Annual Technical Conference (ATC)*. USENIX, Santa Clara, CA, 613–625.

[36] Ke Huang, Chunhui Zhang, Xiaoxiao Ma, and Guanling Chen. 2012. Predicting Mobile Application Usage Using Contextual Information. In *Proceedings of the 2012 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, Pittsburgh, Pennsylvania, 1059–1065.

[37] IDC. 2019. Smartphone Market Share. https://www.idc.com/promo/smartphone-market-share/os

[38] IOVisor 2017. eBPF: extended Berkley Packet Filter. https://www.iovisor.org/technology/ebpf

[39] Yufei Jiang, Qinkun Bao, Shuai Wang, Xiao Liu, and Dinghao Wu. 2018. RedDroid: Android Application Redundancy Customization Based on Static Analysis.. In *Proceedings of the 29th International Symposium on Software Reliability*

Engineering (ISSRE). IEEE, Memphis, Tennessee, 189–199.

[40] Kisung Lee and Youjip Won. 2012. Smart layers and dumb result: IO characterization of an android-based smartphone. In *Proceedings of the tenth ACM international conference on Embedded software*. 23–32.

[41] Huoran Li, Xuan Lu, Xuanzhe Liu, Tao Xie, Kaigui Bian, Felix Xiaozhu Lin, Qiaozhu Mei, and Feng Feng. 2015. Characterizing smartphone usage patterns from millions of android users. In *Proceedings of the 2015 Internet Measurement Conference*. 459–472.

[42] Mario Linares-Vásquez, Andrew Holtzhauer, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2014. Revisiting android reuse studies in the context of code obfuscation and library usages. In *Proceedings of the 11th Working Conference on Mining Software Repositories*. 242–251.

[43] Wei Liu, Ge Zhang, Jun Chen, Yuze Zou, and Wenchao Ding. 2015. A Measurement-Based Study on Application Popularity in Android and IOS App Stores. In *Proceedings of the 2015 Workshop on Mobile Big Data (Mobidata âĂŹ15)*. Association for Computing Machinery, 13–18.

[44] Robert Love. 2005. Kernel korner: Intro to inotify. *Linux Journal* 2005 (2005), 8.

[45] Anandatirtha Nandugudi, Anudipa Maiti, Taeyeon Ki, Fatih Bulut, Murat Demirbas, Tevfik Kosar, Chunming Qiao, Steven Y. Ko, and Geoffrey Challen. 2013. PhoneLab: A Large Programmable Smartphone Testbed. In *Proceedings of First International Workshop on Sensing and Big Data Mining (SENSEMINE'13)*. Association for Computing Machinery, 4:1–4:6.

[46] NetworkWorld. 2014. Intel confronts a big mobile challenge: Native compatibility. https://www.networkworld.com/article/2360304/intel-confronts-a-big-mobile-challenge-native-compatibility.html

[47] Abhinav Parate, Matthias Böhmer, David Chu, Deepak Ganesan, and Benjamin M. Marlin. 2013. Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, Zurich, Switzerland, 275–284.

[48] Thanasis Petsas, Antonis Papadogiannakis, Michalis Polychronakis, Evangelos P Markatos, and Thomas Karagiannis. 2013. Rise of the planet of the apps: A systematic study of the mobile app ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference*. 277–290.

[49] PhoneLab. 2016. Data Release. https://phonelab.readthedocs.io/en/latest/data.html

[50] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. 2014. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 1354–1365.

[51] Inc. Qualcomm. 2018. Trepn Power Profiler - Qualcomm Developer Network. https://developer.qualcomm.com/software/trepn-power-profiler

[52] Leonard Richardson. 2019. Beautiful Soup. https://www.crummy.com/software/BeautifulSoup/

[53] John P. Rula, Byungjin Jun, and Fabian Bustamante. 2015. Mobile AD(D): Estimating Mobile App Session Times for Better Ads. In *Proceedings of the 16th International Workshop on Mobile Computing Systems and Applications*. Santa Fe, New Mexico, USA, 123–128.

[54] SanDisk. 2018. 62 Percent of Indians Run Out of Smartphone Space Every 3 Months: SanDisk. https://gadgets.ndtv.com/mobiles/news/62-percent-of-indians-run-out-of-smartphone-space-every-3-months-sandisk-1829349

[55] Jinghao Shi, Edwin Santos, and Geoffrey Challen. 2019. Lessons from Four Years of PHONELAB Experimentation. arXiv:cs.NI/1902.01929

[56] Helgi Sigurbjarnarson, Petur Orri Ragnarsson, Ymir Vigfusson, and Mahesh Balakrishnan. 2014. Harmonium: Elastic Cloud Storage via File Motifs. In *Proceedings of the 6th Workshop on Hot Topics in Storage and File Systems (HotStorage)*. USENIX, Philadelphia, PA.

[57] Ankur Singla. 2017. The mobile app industry's worst-kept secret. https://ankursingla.com/2017/01/31/reasons-for-the-high-uninstall-rates-in-india/

[58] Statcounter. 2019. Mobile Operating System Market Share India. https://gs.statcounter.com/os-market-share/mobile/india

[59] M. Szeredi and N.Rauth. 2018. Fuse - filesystems in userspace. https://github.com/libfuse/libfuse

[60] Tech Crunch. 2013. Users Have Low Tolerance For Buggy Apps âĂŞ Only 16% Will Try A Failing App More Than Twice. http://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice/

[61] The Apache Software Foundation. 2016. Apache Maven Project. https://maven.apache.org/index.html

[62] Yuan Tian, Meiyappan Nagappan, David Lo, and Ahmed E Hassan. 2015. What are the characteristics of high-rated apps? a case study on free android applications. In *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 301–310.

[63] TmoNews. 2012. Google Head Of Android User Experience Explains The Lack Of SD Cards For Nexus Devices. http://www.tmonews.com/2012/10/google-head-of-android-user-experience-explains-the-lack-of-sd-cards-for-nexus-devices/

[64] USA Today. 2014. iOS8 users massively deleting to make room. http://www.usatoday.com/story/tech/columnist/talkingtech/2014/09/18/ios8-users-massively-deleting-to-make-room/15834211/

[65] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. 2017. To FUSE or Not to FUSE: Performance of User-Space File Systems. In *15th USENIX Conference on File and Storage Technologies (FAST) (FAST 17)*. USENIX, Santa Clara, CA.

[66] Nicolas Viennot, Edward Garcia, and Jason Nieh. 2014. A Measurement Study of Google Play. In *Proceedings of the 2014 ACM SIGMETRICS Conference*. ACM, Austin, TX.

[67] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. Wukong: A scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 71–82.

[68] Wikipedia. 2015. iPhone 7. https://en.wikipedia.org/wiki/IPhone_7

[69] Wikipedia. 2019. iPhone XS. https://en.wikipedia.org/wiki/IPhone_XS

[70] Qiang Xu, Jeffrey Erman, Alexandre Gerber, Zhuoqing Mao, Jeffrey Pang, and Shobha Venkataraman. 2011. Identifying Diverse Usage Behaviors of Smartphone Apps. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference (IMC '11)*. ACM, 329–344.

[71] Tingxin Yan, David Chu, Deepak Ganesan, Aman Kansal, and Jie Liu. 2012. Fast app launching for mobile devices using predictive user context. In *Proceedings of the 10th ACM International Conference on Mobile Computing Systems (MobiSys)*. ACM, Lake District, United Kingdom, 113–126.

[72] E. Zadok, J. Osborn, A. Shater, C. P. Wright, K. Muniswamy-Reddy, and J. Nieh. 2004. Reducing Storage Management Costs via Informed User-Based Policies. In *Proceedings of the 12th NASA Goddard, 21st IEEE Conference on Mass Storage Systems and Technologies (MSST)*. College Park, MD, USA.

[73] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. 2014. ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*. 25–36.

[74] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. 2015. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM conference on security & privacy in wireless and mobile networks*. 1–12.

[75] Xun Zou, Wangsheng Zhang, Shijian Li, and Gang Pan. 2013. Prophet: What app you wish to use next. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, Zurich, Switzerland, 167–170.

Received February 2021; revised April 2021; accepted April 2021