Serving Mobile Apps – A Slice at a Time

Ketan Bhardwaj
Georgia Institute of Technology
ketanbj@gatech.edu

Matt Saunders
Georgia Institute of Technology
msaunders34@gatech.edu

Nikita Juneja
Georgia Institute of Technology
nikita.juneja@gatech.edu

Ada Gavrilovska
Georgia Institute of Technology
ada@cc.gatech.edu

Abstract

End users wanting to do more and more with mobile apps has led to explosive growth in the number of available apps. This has widened the gap between developers making apps available and end users being able to install all the apps they want on their device. To address this, Google introduced Instant Apps for Android where users can access selective app features on demand without having to download and install entire apps. But this requires developers to refactor apps and limits the apps' functionality.

In this paper, we present AppSlicer – a solution that automates the creation, delivery, execution and cleanup of lightweight app slices from native apps. With AppSlicer, app slices are created from existing native apps, without requiring any additional developer effort. App slices run on end-user devices and correspond to arbitrary single functionality (task) that is carried out using an app. We demonstrate that app slicing is practical, it provides users with seamless access to app functionality with performance matching that of native installed apps, and better than other technologies for on-demand delivery of apps.

CCS Concepts • **Software and its engineering** **Functionality**; *Distributed systems organizing principles*; *Organizing principles for web applications*; *Completeness*; *Dynamic analysis*; *Software performance*; *Software usability*; *Runtime environments*; • **Information systems** *Multimedia streaming*; Location based services; • **Security and privacy** *Mobile platform security*;

ACM Reference Format:
Ketan Bhardwaj, Matt Saunders, Nikita Juneja, and Ada Gavrilovska. 2019. Serving Mobile Apps – A Slice at a Time. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303989

1 Introduction

Mobile apps have permeated every aspect of our daily life. The number of available mobile apps in app stores has exploded and has sustained a momentum that is unprecedented. As we move toward more digitized user experiences through development of new smart city services and applications, this trend will continue. However, this has also created a problem for end users. End users want to do more with mobile apps [23], but have to put a lot of effort in explicitly installing or managing the many apps on their devies. This unwanted app clutter negatively impacts user experience and thus, drives down new app adoptation rates, particularly for apps providing transient, infrequent interactions. For instance, a tourist traveling through Europe may need to find and install dozens of different apps each time they visit a new city, if they want to get a full benefit of the digital technologies enriching user experiences in these places [6, 33, 34]. This is a global trend that will not slow down in future [35, 36].

Several technologies can be employed to address this problem. First, browsers are continuing to embed richer functionality for better location-based web app experience. Other examples are based on using an image with a link to a web app, redirecting users to install an app, or showing a screen capture of an app running in a remote location [16]. Some of these solutions fail to provide users with a full native app experience; yet, numerous reports provide evidence of significant preference of users toward native apps-based interactions [27, 29]. The other solutions lack in that they require complete app installation, cluttering a device with seldomly used apps.

One may argue that a simple solution based on app recommenders or cleaners [5] will suffice to address the problem of having to install/uninstall rarely used apps. Typically, such solutions require explicit installation and use-based (e.g., app not used in the last n-days) automatic uninstallation. There are two main issues with such approaches. First, installation still requires minutes to download and 10s of seconds to install which has been known to deter users from trying new apps. Second, uninstallation may remove all the app packages but does not clean the state stored by an app on the device and/or undo any changes made to the system state. In practice, most of the high level and seemingly simple solutions have side effects that need to be explicitly handled. In a nutshell, users want a native app experience but without any hassles.

To provide for an authentic native app experience without an explicit download and installation, Google recently introduced Instant Apps [20], which provides a way for users to immediately start using an app. Instant Apps is an API to be used by developers to refactor existing apps into a form suitable for piecewise on-demand delivery. Google's Instant Apps

aim to reduce the user efforts associated with on-demand app delivery and thus to expose end users to larger number of apps instantly. However, to do that, apps need to be re-written to be delivered as Instant Apps, a process that is reported to take substantial developer effort [40]. This just shifts the onus of solving the problem to app developers by making them reimplement their apps without solving it generally for all apps. Furthermore, the Instant Apps API introduces inherent limitations on the functionality which can be provided by the on-demand apps.

In response to these problems, we present **AppSlicer** – a solution that *automates* the creation, delivery and execution of *appropriate and lightweight slices of native apps* on end user devices *on-demand*. Realizing such a system faces the following main problems:

Creating app slices from native apps. This refers to the problem of identifying the appropriate subset of app components that implement a particular functionality. Inspired by the concept of program slicing [2], AppSlicer uses *dynamic program slicing* to automatically create app slices from existing native apps, without requiring any additional developer effort. App slices correspond to a single functionality (task) that is carried out using an app. Compared to conventional program slicing which relies on static and dynamic program analysis and algorithms to find slicing boundaries, App Slicer relies on observation of execution state to find slicing boundaries in native apps.

Efficient delivery of app slices. AppSlicer addresses this by new support for *app streaming* in mobile OSes, allowing end users to retain the full native app experience for on-demand streamed apps. A dynamic app streaming backend eliminates the intractable problem of finding an optimal complete slice, since missing functionality in app slices can be simply streamed if needed.

Secure execution and clean up of app slices. App slices should not cause unwanted effects or clutter on a device. AppSlicer support in mobile OS needs to ensure that slices delivered to end user devices are not tampered with and are cleaned up after their use. AppSlicer addresses this by integrating new *runtime-integrity-checks and ephemerality* support in the mobile OSes. Further, the code as well as state created for or by the app slice should be granted access to device features, remain isolated from other apps and cleared from devices automatically when no longer needed. This is addressed by leveraging the existing permission model and running app slices securely in a sandbox. In this manner, AppSlicer provides at least the same level of isolation as for native apps, and does not present any functional or security issues for existing apps.

We prototype AppSlicer for Android, and evaluate it using 50 of the top apps from the Google Play Store. In our experimental evaluation, we show that it is feasible and efficient to automatically create app slices without any developer effort. Further, we show that app slices have lower barrier to use, better performance and lower resource consumption than alternatives. Finally, we show that device-side support for AppSlicer keeps the device clean of any clutter. In summary, with AppSlicer we contribute:

- The concept of app slices (§3) and supporting system-level mechanisms needed to create, deliver, execute, and clean up after app slices in mobile OSes (§5), along with the implementation of these concepts for Android (§6).
- Demonstration and functional evaluation (§7.1) of automatic creation of app slices from native apps with no additional efforts from app developers, compared to the refactoring needed by Instant Apps, made possible via use of execution boundary as a criteria for program slicing.
- Experimental evaluations showing that app slices outperform other technologies such as web apps and thin clients (§7.2, §7.3), and are on par with installed native apps and Instant Apps, including based on a user survey for quality of experience (QoE) (§7.4).

2 Why App Slices?

We motivate app delivery via app slices based on the rise of scenarios promoting transient app usage, the opportunity for reduction in app delivery overheads, and based on user convenience.

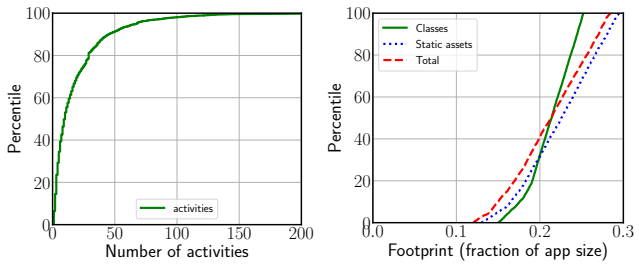
Emerging smart environments promote app usage. Users are increasingly more reliant in their experiences on app-based services and information. A vacation in Orlando will call for different apps for different amusement parks, which can range up to several tens of MB. For instance, Disney's app is 82 MB, and Six Flags' app is 57 MB. The apps integrate many different types of features – from navigation, to checking wait times, requesting priority passes, searching menus and ordering food, adjusting restaurant or hotel reservations, etc. A user may not need all of those features. When visiting some of the cities which are already leading the way toward digitalization and Smart Cities, tourists may be exposed to tens of different apps. Just the City Council of Barcelona, for instance, offers currently 30 apps, with sizes going up to 73 MB [6], and many other types of apps are available through other public or private stakeholders. While useful and highly rated, the expectation that users will continue to manually download, install, and subsequently delete apps they only transiently use , challenges the apps' adoption.

Full app delivery is not always necessary. For infrequently or transiently used apps, it is highly unlikely that all of the app functionality is needed. We make the following observations:

- *Apps contain more than one functionality.* Figure 1(a) shows the CDF of the number of activities per app for the top 5000 apps downloaded from Google Play store. Concretely, it shows an average of 19 activities per app with a minimum of 1 and maximum of 403 . That doesn't include any ActivityFragments (a feature introduced in Android 3.0) in a single activity that an app might be using to

Approach	Barrier to use	Developer effort	Functionality	Ephemerality	Performance
Native apps	High	None	Unlimited	No	High
Web apps	Low	None	Limited	Yes	Low
Thin Clients	High	High	Limited	Yes	Low
Instant Apps	Low	High	Limited	No	High
App Slices	Low	None	Unlimited	Yes	High

Table 1. Advantages of AppSlicer over existing approaches.



(a) (b)

Figure 1. (a) CDF of the number of activities per app for the top 5000 apps downloaded from Google Play store. (b) Memory footprint of apps when launching the main activity vs. their app package size.

let end users perform many tasks from a single activity. As a concrete example, the Six Flags app has 70 activities.

- *Apps do not require to load all content available in an app package to offer a single functionality.* Figure 1(b) shows the memory footprint of apps when launching the main activity vs. their total app package size. We observed this to be always less than 25% of the memory used by apps including all code (JAVA classes, JNI native libraries, etc.), assets (UI images, etc.) and resources (font, string values, etc.) . As a concrete example, during our analysis using 22 of the 70 activities in the Six Flags app, a user can get all relevant info about a particular water park using only 5.4% of the apk.

3 Limitations of existing technologies

Other existing technologies have technical or deployment limitations to enable on-demand delivery of functionalities. The advantages of AppSlicer over other relevant technical approaches are briefly discussed below (Table 1):

- **Web Apps:** Web apps are plagued with well known concerns (security, privacy) and limitations (access to device features) of browser based apps or web apps on mobile devices. They are susceptible to bloat caused by the use of web technologies, highlighted in works around web page loading latency issues [11, 30].
- **Thin client apps:** At first glance, the thin client approach, i.e., streaming pixels of an app run remotely to the end user device, seems reasonable, owing to mobile devices' support for efficient video processing [16]. However, inherent limitations in terms of the network and power usage, and lack of access to on-device sensors, limit their use.
- **Full native apps:** As an alternative, proactively pushing full native apps on-demand to a device is another option. However, this can be burdensome and time consuming, defeating the whole purpose of providing on-demand experiences ; [10] reported that the lower bound of the installation times for 5000 of the top apps in Google Play range from 1 to 16s in addition to their download time.
- **Instant Apps:** Recently Instant Apps were introduced by Google. However, they have seen very low adoption from developers, and suffer from the following limitations.
  Developer Effort: Google requires the developers of an app to refactor their entire source code. The code needs to be broken into separate modules (one base + multiple features) where each module is less than 4 MB. Every module should be able to run independently, having an entry point activity. These activities need to be addressable, i.e., each one of them has to correspond to a unique URL address. Vimeo, which was an early partner for Google Instant Apps, reported that their dedicated team of Android engineers started refactoring their app in Feb'17 to be ready to present Instant Apps in May'17 [40]. They had to use tools such as APK analyzer and Dexcount Gradle plugin to see a visualization of the tree of dependencies. Our own experience of developing Instant Apps confirmed that the level of effort needed is commesurate to that of writing a new app from scratch.
  Functional Limitations: Along with the 4 MB size limit, Google also imposes a number of additional limitations which hinder the functionality of these Instant Apps. Background services are not supported and only 11 types of app permissions are allowed. If an Instant App includes a login it must integrate with Google's smart lock for passwords, and use the Google payment API to accept payments. Instant Apps may only save their state using SharedPreferences on a device's internal storage which is an unstructured key value store. The base module of Instant Apps always remains in the cache, unless evicted during garbage collection or cleared at reboot, making it difficult to comment on whether it addresses app clutter, or makes it worse. Finally the ability to provide the user with push notifications has only just been included in the latest Beta release along with increasing the size limit to 10 MB. Instant Apps

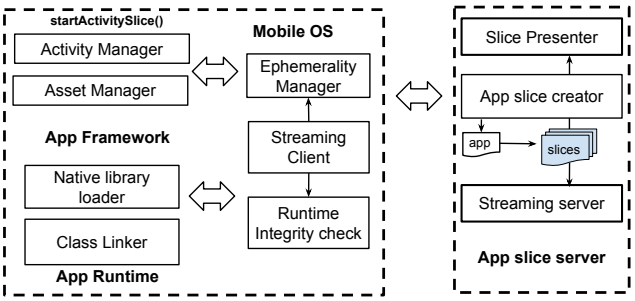


Figure 2. High level components of AppSlicer, including the app slice server and the device-side support.

seem to be designed to supplement an exisiting app in order to drive user installations of the full version instead of providing full support for streaming-based app delivery.

4 Overview of AppSlicer

Scope. The scope of AppSlicer is to provide support for creation, delivery and execution of app slices on end user devices, by adding appropriate support in mobile OSes. The context tagging and intelligent matching capabilities required to select appropriate app slices to be delivered to users based on their context is orthogonal to the mechanisms provided by AppSlicer. Note that apps with background services are not good candidates for being delivered as slices because app slices are intended to augment the experience on users' explicit intent rather than for continuous tasks, e.g., delivering a footstep counter app as an on-demand slice is futile, instead it should always be installed. AppSlicer focuses on on-demand app experiences. We envision that AppSlicer is run as a web service in data centers and may be supported by its edge deployments. Techniques to reduce server-side bloat such as server-side deduplication, compression, etc., are out of the scope of this paper.

Brief. The three main problems in enabling a practical on-demand app ecosystem are creating app slices from native apps, ensuring efficient delivery and secure execution of these slices, and their cleanup from end user devices. AppSlicer solves those using its three main components:

1. an app slice server which automatically restructures native apps into app slices to serve them on-demand;
2. app slice streaming and execution support in mobile OS
3. runtime-integrity, ephemerality support in mobile OS.

Figure 2 shows the high level overview of AppSlicer. On the AppSlicer server side, app slices are created apriori from unmodified app packages (.apk files) and presented to end users as icons of apps available on their home screen [10]. If a user chooses to launch them, they are delivered and run as if they were installed native apps. If any app component not part of the delivered slice is needed, it is fetched from the app slice server just-in-time. This is made possible by the support added to Android to enable app slice streaming and fine-grained runtime integrity checks. These mechanisms are designed to provide the same experience, security guarantees, and performance as native apps. Finally, ephemerality support added to Android keeps devices clear of app slices or any on-device state they create during execution.

App slices are sound and complete in the sense that they can be executed on a device as soon as they are delivered. However, AppSlicer does not guarantee optimality or preciseness of slices. In general, optimal program slicing is an intractable problem due to its dependence on inputs, internal and external program state. For app slicing, these include dependencies on user inputs, existing installed apps, and instantaneous context (sensors, orientation, location, etc.) To reduce the complexity of such slicing, we back the slice delivery with support for app streaming [9]. If more functionality is needed due to user interactions or change in context, the corresponding app components can be streamed on-demand.

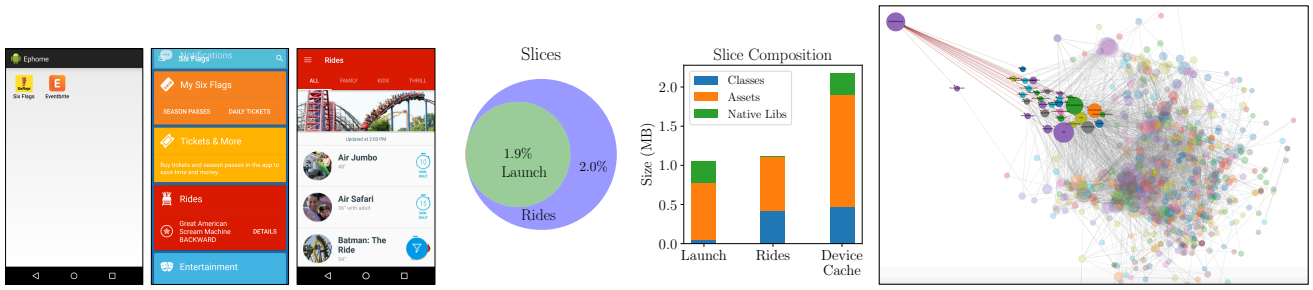
One can argue for slicing to be just-in-time as opposed to creating user-specific app slices offline, but this may end up adding additional time to make an app slice available to end users. In this work, we take an offline approach to slicing to ensure the *time to use* a slice remains within reasonable bounds ($< 5s$) to offer quicker interaction to end users.

Once delivered, runtime integrity checks are performed for each fetched slice and its contituent app component , before they execute. App slices are run in the same type of sandboxes as native apps. However, sandbox boundaries are tighter than for native apps as a result of using tighter SEAndroid policies. App slices are constrained by the same permissions model as native apps.

Finally, the on-device state for each app slice is maintained separately from the state of installed apps; each app slice is isolated from slices of other apps and cleaned up when the app slice is closed. The support for app slice streaming and ephemerality in the mobile OSes enables that. The choice of reusing the state of a previously ran app slice is explicit and can be selected by users if they want the state to persist on a device. This does not apply to the executable or the assets part of a slice to ensure that if in different contexts different app slices of the same functionality of an app needs to be delivered, they do not conflict with each other.

Visualizing app slices. To better understand app slices, we provide a visual illustration. Let us consider the same example of the Six Flags app. The entire app is 57 MB, containing a total of 70 activities such as map, cart, ridedetails, events, restaurants, tickets, parking, etc. If a person who already has a ticket and does not need to park is visiting Six Flags and only wants to check the wait time for a ride, why should they need to download and install 57 MB on their phone? As stated earlier, using 22 of those 70 activities, a user can get all relevant info about a water park using only 5.4% of the apk file.

Figure 3(a) shows how the user advances through the app to see ride wait times. They click on the icon shown to them



(a) (b) (c)

Figure 3. (a) User advancing through Six Flags app to check ride wait times via AppSlicer- Icon, HomeActivity and Rides; (b) *HomeActivity* and *RideDetailsFragment* slices constituents; (c) classes in *RideDetailsFragment* slice vs. all classes in Six Flags app.

on their home screen (Figure 3(a) left), and then click on Rides. On the system side, when the user clicks on the icon, the app's launch activity slice is delivered. *SplashActivity* is Six Flags' launch activity. *SplashActivity* tries to open *HomeActivity* (Figure 3(a) middle) which is dependent on the *SelectedParkManager* class.

In contrast to the full app, Figure 3(b) (left) depicts the size of the initial home slice as percentage of the total apk size which is delivered to open *HomeActivity*. When a user clicks the Rides option, the dependencies (app components) for *RideDetailsFragment* are streamed to the device. This corresponds to an additional 2.0% of the apk size.

Figure 3(b) (right) shows the breakdown of the components delivered to the device in terms of Android assets, Java classes, and native libraries. Figure 3(c) illustrates the unnecessary classes which did not have to be downloaded because of AppSlicer. For brevity, only classes are shown but other assets, libraries, etc. i.e., app components follow same proportion.

5 Design

AppSlicer combines four key components, responsible for app slice creation, streaming, managed execution, and clean up. In the remainder of this section, we illustrate the design of each of these components for the Android-based AppSlicer.

5.1 App slice creation

AppSlicer creates slices from an app package at the *execution boundary*, i.e., based on the classes and static assets that are loaded at run time when an activity is launched and before it starts servicing the user input. Detecting this condition is simple – launch an activity and do nothing. To do that, the AppSlicer server creates an execution dependency set for each activity in an app as a subset of all app components available in the app package, using static and dynamic analysis of app packages. It extracts this subset by launching app's activities on an instrumented Android running on a virtual machine to create lists of app components associated with different activities. Once these lists are created, AppSlicer uses standard Android SDK tools to create multiple packages – or app slices – from the apps.

Note that this does not involve searching through a large search space to create precise slices. Trading off preciseness for tractability, AppSlicer is able to address the intractable problem of optimal program slicing. However, slicing is sound. First, slices are observed to be executing on devices when verifying for functional correctness during our experiments. More generally, AppSlicer provides for on-demand delivery of slice components. When a user navigates to a part of an app that is not part of a delivered slice, the identification of the missing app components needed for that functionality happens lazily, on execution. When a requested app component is not found on-device, it is requested from the app streaming server. How often this happens depends on the different ways in which users use a particular app (or would use app slices). For correctness, slices are created by using a dependency set consisting of the app components that led to successful execution, so they are always correct and sufficient to execute that slice. But they may not include all components needed by every user for further interaction with an app slice. This is the reason app slicer has to be backed by an app streaming server, so any missing functionality can be dynamically delivered.

For each app slice, AppSlicer creates an *app slice manifest* containing only the app components required to make the app slice appear (not execute) on an end user device. Once an app slice appears on a device, it can be subsequently launched, and the actual app slice would be fetched . The slice manifest is automatically extracted from the native app package without any developer effort, and contains the package name, label, main activity name, link for an app's icon and the top level hash of the merkle tree of the uncompressed app slice components as leaf nodes. The actual app components (classes, static resources, etc.) required to run an app's activity are only fetched at the launch time.

To further provide for slicing of arbitrary apps, AppSlicer also addresses the following challenges:

Multiple device form factors. AppSlicer creates slices separately for each form factor because apk files often contain multiple versions of the same resource. Normally, the correct version of the resource is used at runtime while the other versions sit idly, taking up space on the user device. AppSlicer is able to eliminate this waste by delivering only relevant components based on the form factor of the requesting device. By eliminating unnecessary device form factor-specific resource versions from the apk files, AppSlicer is further able to reduce wastage of device resources.

On the server side, AppSlicer creates slices separately for each form factor because of current lazy resolution of requested components in Android. For the same app, even though the name of the resource is the same, it may be resolved to a different directory in the app package at runtime, based on the device form factor. However, this is not a fundamental limitation because slices can also be generated dynamically, on-demand, depending on a user device form factor.

Hardened apps. AppSlicer works for all apps regardless of app hardening methods, e.g., obfuscation, including string encryption, method renaming, control flow obfuscation, etc., simply because of the fact that slicing is done on execution boundaries which do not require any analysis of the app code or resources to create slices. In that sense, app slicing is compatible with such compile time app hardening solutions.

Developer input. Developers may want a way to ensure that some functionality is always part of every slice of their app, e.g., ad fetching in their app. AppSlicer can address that simply by always including a static list of app components in the dependency sets of all slices created for their app. This is a practical design decision as a large number of free apps rely on advertising for their revenue. Forcing developers to exclude ad libraries from app's slices will automatically rule out use of AppSlicer for a large number of apps.

Issues with other approaches to create slices. One could argue that there are a number of other ways to achieve similar functionality. One approach would be to partition the app based on the static code boundaries, i.e., classes in case of Android apps. However, we found that launching apps not only requires classes but also other static assets and sometimes native libraries. Further, apps are structurally dense in terms of their code [3]. This ruled out slicing on static code boundaries. Another approach would be to create app slices as bundles of classes and assets that are part of any particular activity, for instance those referenced from the main activity class. However, as it turns out for most apps, the activities defined in typical Android apps are tightly linked to each other. Hence, for every activity the corresponding bundle is almost equivalent to the entire app package due to the spaghetti dependency structure of the apk [3], leaving out only classes and static resources which are unique to the device configurations rather than the app's functionality itself.

5.2 On-demand app slice delivery

The app slice delivery in AppSlicer is performed in two phases – presentation and execution phase – which involve an app slice presenter app, the Android app framework, and the app runtime. In the *presentation phase*, an app slice manifest is delivered to the end user device. At this point, the app slice is displayed to end users as an icon with a short description of what the slice can be used for. Until this step, there is no involvement of Android's system components.

When a user decides to launch an app slice, the same interface as used for the native app is shown to the end user by fetching the app components listed in the app slice manifest from the AppSlicer server. The app slice is then launched as if it were an installed app on the end user device, after checking their integrity using the merkle tree shared in app slice manifest. We observed that streaming individual app components one-at-a-time in a slice requires multiple connections and thus, incurs high overhead. Further, additional app components may be required if a user tries to do anything with that app slice, say launch another activity. Those app components are made available on-demand via streaming from the app slice server where they are present as an uncompressed copy of the native app package. Note that once an app component is delivered as part of a slice, it is cached on the device and does not need to be delivered again. All app components are made available on-demand and checked for their integrity at run time. We observed that related activities in an app have high overlap among app components they use, and fetching them individually would include high network overhead. If a user requests another launchable activity of the app for which a slice is already delivered, it is delivered as a new slice (see Section 7.2).

For native apps, loading app components from installed app involves interaction among the Android frameworks context, package manager, and activity manager. Briefly, app component requests are made from the Android runtime's class linker, JNI loader, native library loader (ld), and asset manager in the Android framework. These are all modified to support delivery of app slices. Additional support is needed for authentication of the app slice server. This is achieved by adding certificates to Android in a similar way to what is done with web browsers. In that sense, app slices carry with them all enhancements of Android apps over JAVA applets.

Issues in other approaches to deliver app slices : One may argue that app slices can be delivered via application defined JAVA class loaders. However, components of the slice are requested from different subsystems and accessed in different ways because everything is not loaded as a JAVA class. Specifically, the App's JAVA classes are loaded by the ART either as byte code from classes.dex in the app's package or as a mix of byte code and compiled binaries from the OAT file. Native libraries used in apps are loaded by ART using the Linux dynamic library loader. Static assets (e.g.,

from .arsc files or res or asset folders) and non-assets (e.g., AndroidManifest.xml) are accessed using the asset manager implementation in the androidfw app framework module. Further, these components are used in very different ways, e.g., an app can use either a URI and/or a binary buffer mode to load static assets.

One may also advocate mounting a storage partition on an end user device, used by networked file systems hosting packages of apps for app delivery. Although this is technically feasible, it would require the app mounting partition to be assigned root access on a device, which is not a good security decision. Mounting a network file system is possible without root privileges using FUSE drivers. But that means that the interactions with the file systems need to be implemented explicitly in the app, creating even more work for developers. Also, since most app components are files mapped and loaded during execution, relying on a network file system to optimize the app component transfers makes it impossible to exploit opportunities to reduce the device side overheads. For instance, it becomes impossible to defer the class/asset look up to the AppSlicer server, or to use server side un-compression of app packages shown to provide better app loading performance. Finally, it makes it difficult to track the fetched slice components, as the caching is controlled by the network file system.

5.3 App slice execution

To keep things clean and simple for launching delivered app slices, AppSlicer uses a newly added API in Android framework – startActivitySlice() – via its Android context. Android context is an interface to global information about an app environment which is used to launch activities via the startIntent() API for native apps. In addition, the AppSlicer design also addresses the following challenges in enabling app slice execution:

Preserving sandbox for app slices. The *execution phase* starts when a user launches an app slice by touching its icon. The AppSlicer support in Android includes provisions to create and maintain a sandbox environment for the app slice similar to that of an installed app. To ensure compatibility with existing apps, this involves assigning appropriate permissions, applying the app slice security context, and forking a new instance of the enhanced Android runtime. Specifically, before zygote-forking an instance of the Android runtime, the Android package manager ensures that identifiers (uid, gid) for app slices belong to a separate range from those used for installed apps that are returned to the Android Activity Manager. They are later used to handle permission exceptions to provide lazy permission approvals for app slices (e.g., if an app requires more permission to access a device resource).

To keep app sandboxing guarantees for app slices, AppSlicer adds a new SEAndroid security context and associated policy: app slice domain. During execution, all additional requests to load classes, native libraries, and static assets are intercepted and redirected to the AppSlicer's app slice servers via a streaming client and runtime integrity checker. App slices execute under stricter security constraints and with the same permission model as installed Android apps. The JNI code in an app slice is constrained by a stricter SEAndroid policy that limits what an app slice can access to only a single directory, newly created specifically for each app, and no other component on the device's file system. Although each slice is created separately on the server side, on the device all slices of an app are executed in a single sandbox. Therefore, all activities of a single app can share state and/or files seamlessly. In that sense, app slices are at least as secure as native apps.

Handling app slice state. For native apps, app components are stored on device's local storage at a location assigned at installation time, which is not changed as long as the app resides on the device. For app slices, there is no such location when they are first fetched. Instead, app component requests are intercepted, and appropriate directory structures are created and managed at runtime. This also allows the app streaming client to cache the fetched app components for better performance.

Handling app slice failure due to disconnection. For native apps, all necessary app components are always available from device storage populated during install time. However, app slices may rely on missing components, or trigger other activities, which need to be fetched on-demand; thus, loss of connectivity presents a challenge. AppSlicer's device-side support handles disconnection gracefully. If the app components needed by user actions are already fetched then the user does not see any difference. Otherwise the Android runtime generates an exception which is communicated to the app slice presenter app that launched that app slice, which informs the user about disconnection. The state generated by that app slice until that point can either be saved, if opted by the user, or would simply be removed during clean up.

5.4 App slice cleanup

Ensuring clean up of an end user device is non-trivial due to the multitude of state an app creates on the device. AppSlicer keeps end user devices clear of the system and the app slice's private state created during its delivery or execution. To facilitate this, hooks are added in relevant places in the app framework, runtime or libraries, to explicitly log the on-device app slice state on a per app slice basis using additional library-supported APIs. Android provides an existing well-defined API which allows apps to log newly created or modified app-specific or global state, preferences, etc. AppSlicer uses the API to capture all changes made by an app slice in a per-app log. The same API provides an interface to remove global state during uninstallation. This is used by AppSlicer to undo changes without taint tracking. This does not require any changes in apps or their slices, and is fully supported at the level of the Android framework, runtime and libraries. This

is sufficient for apps that follow the Android guidelines, as shown in our experiments in Section 7. However, apps using native libraries and root access on a user device may not be covered by this approach. Fortunately, there are only few such apps and they do not form good candidates for slicing in the first place, such as device management apps.

These per-app logs, implemented as SQlite databases, are key to enabling system level app ephemerality guarantees and are handled by an ephemerality manager, which is part of a new system library: libephemeralutils. The ephemerality manager, allows processing of the logs under different policies, each of which is derived from end user preferences about the state created by app slices. For instance, a user may indicate they wish to discard all app slice state on app slice exit. The ephemerality manager can then gather the app slice state generated by that slice using the logs and proceed to delete such state and ultimately the logs from the database. However, app slices of a single app can share the same database to log its slices. It can also be argued that instead of a per app ephemeral log, a single log would suffice for all app slices of different apps. However, that prevents users from being able to specify different policies of how to handle removal of the state for different apps. For instance, take a shopping app's slice. The end user could be allowed to choose that they want to remove all app components for that app slice, but retain their on-device shopping cart or invoice created by the app slice. Further, libephemeralutils also implements checks to ensure integrity of each received app slice component from the streaming client using the top level merkel tree hash shared during the presentation phase, thereby ensuring runtime integrity check.

Issues with other approaches for cleanup. One may argue that confining accesses granted to running app slices to specific directories should suffice for tracking state. But, not all app state is maintained at a file or directory granularity. For instance, permissions added for an app are appended to a file /system/packages.xml which if removed or corrupted would break the system. Similarly, other extreme approaches such as those described in forensic deniability [15] can be used to run app slices in private sessions, monitor their IPC, and track all memory allocations and interaction with any device features, so as to track and remove every possible trace of an app execution. However, it is important to note that Android apps are already run in a sandbox and sandboxing a sandbox entails unnecessary overhead. Our goal is to keep a device clean as opposed to providing strong guarantees. So, we can minimize the overhead associated with tracking and still avoid device pollution by using a lightweight approach that relies on the mobile OS rather than on invisible tracking.

6 Implementation

Creation of app slices is performed per app using offline scripts that use the Android SDK tools to identify all launchable activities within an app, and launch those activities on an instrumented Android running on a virtual machine. Contents of app slices are determined by creating a log containing the set of loaded app components per activity, and creating a profile for the app. If the launch is successful, then the set of app components are obviously sufficient for functional correctness. We do not require any further steps to capture user interactions. Being backed by the app streaming functionality [9], if any app components are needed, then those are fetched/streamed on-demand from the AppSlicer server. Names of components to be delivered on-demand are available in the code itself, e.g., the class linker uses class names, the asset manager uses resource id, and System.load() uses library name. This results in execution dependency sets for all the different activities in those apps. We then uncompress the apps using 7z and decompile the app's classes using dex2jar tool. We rebundle and recompile the identified app components and create app slices. Additionally, we extract app slice manifests from app packages and calculate the root of the merkle tree for runtime integrity checking. Merkle tree is set up for the uncompressed apk and a hash is generated using an apk signature. Each app slice can be verified independently of others by relying on verification of its constituents using the top level hash at runtime.

Changes in Android. The implementation of AppSlicer's device-side modules can be summarized as a presenter app , system level changes in Android app framework, Android runtime (ART), SEAndroid, and ephemerality support that is newly added to Android. To support app slice launch, the following classes of the app framework are modified: Context, ActivityManager, PackageManager, AssetManager. Android runtime's class linker and the native library loader are also modified to support app slice execution. In SEAndroid, we added a new policy to support app slices. Finally, we added a logging library for ephemeralutils which is used by each of the components that creates or modifies any state. The logs are then used to remove any state created or modified by an app slice when its execution finishes. A detailed list of changes in Android and other source code is available at https://github.gatech.edu/kernel/appslicer.

7 Evaluation

In this section, we present our experimental evaluation of the four key components of AppSlicer.

Experimental setup. Table 2 lists the details of the experimental setup used in the experiments. The testbed includes a device, and three possible deployment alternatives for an AppSlicer server.

Workload. We perform a more detailed evaluation of app slicing and responsiveness with a small subset of 50+ apps

Setup	Description
Phone	Nexus 5 phone, based on AOSP Lollipop v5.0.1.3
Edge	Linksys WRT1900ac 802.11ac Wi-Fi router, OpenWRT, 20GB Samsung SSD via USB 3.0
Simulated CDN	OpenStack Ubuntu 14.04 VM on local cloud test bed: 4 vCPUs, 16GB Memory
Remote Cloud	Amazon EC2 Ubuntu 14.04 instance (m3.xlarge): 4 vCPUs, 15GB Memory
Internet	Gigabit Ethernet connected via cable to edge

Table 2. Experimental setup

Variety	Rationale
Tops 40 free	Evaluate apps from every different categories.
Scenarios	**Context**
Taxi (T)	Booking a cab in new city
TV Remote (R)	Controlling a device at home or hotel
Vending (V)	Interacting with ad-hoc device
Conference (C)	Checking an event schedule
Hello World (H)	Microbenchmarking
Performance	**Known to exercise resources**
Facebook (S)	CPU, storage
Chrome (W)	Network, Memory
Zedge (P)	Network, power
8Ball pool (G)	CPU, power
Guidebook (E)	Network, power
SixFlags	Running example

Table 3. Apps used in evaluation with rationale of choosing them.

listed as top apps on Google Play store, seen in Table 3. We chose these apps by (i) selecting 40 popular apps, ensuring that we include the top app in each category, (ii) including 5 contextual apps that are representative examples of using app slices, and (iii) including 5 apps that are known to pressure specific resources on end user devices. The rationale behind choosing these apps is twofold: (i) we want to highlight that AppSlicer works for the most popular apps which often are the most complex apps implementation-wise, and (ii) to highlight that AppSlicer can support different kinds of apps which can be used in different realistic contexts and which exhibit different resource requirements.

7.1 App slice creation

To characterize the effectiveness of AppSlicer in creating functionally correct app slices, we use the following metrics:

Correctness. We report that arbitrary functionality can be delivered in the form of app slices, by automatically creating them with AppSlicer. We verified that for all activities in the 50 apps we chose as our workload. Important to note here is that even if a user decides to use beyond what is delivered as an app slice, the app slice leverages app streaming to fetch the required components and to provide seamless uninterrupted experience for users. Since app slices execute as native apps, they have access to all device features.

Precision and Soundness. As stated earlier, slicing is not precise but that trade-off is what enables AppSlicer to address the intractable problem of optimal program slicing. It is impossible to make a general assertion regarding precision and soundness without complete analysis of the different ways in which different users use a particular app (or app slice). So, we provide empirical evidence based on common usage of the subset of apps that can help us subjectively comment on the slicing precision and soundness.

We define common usage for each app independently by excersizing the app in ways an average user would do. Take the Facebook app for example. We used it because it is known to be one of the most complex apps. We performed the following activities with it. *Newsfeed:* Open the app, scroll through news feed (about 100 posts worth) click a couple embedded videos and click a couple articles that link to the website through the Facebook webview. *Notification:* Open the app, click notifications, scroll through notifications, click on suggested friend, go back, click on an event page, go back, click on a comment the user was tagged in, go back. *Marketplace:* Open the app and scroll through marketplace, extend a few categories, click a few posts and scroll through them, send "ask for more details" message on a post. We defined app specific activities for all such apps. Each time we logged all components requested by the application. Since the application is backed by the AppSlicer server, as we reach the functional boundary of a slice, any necessary components for the user to continue could be streamed on demand. Using this log data we visualize how the various components required for each use case are related. Figure 4-top shows it for 5 representative apps. Again, for the Facebook app example, we measured that launching the app and scrolling through the newsfeed takes 19.8% of the total apk size. If we open up and scroll through the notifications panel while selecting notifications at random we only have to stream an additional 2.7% and if we then browse through the marketplace an additional 9% is required as seen in Figure 4-a. Similarly we were able to launch and use the TV remote app while only needing 5% of the apk size (Figure 4-b). These space savings are a direct result of the ability of AppSlicer to deliver the core functionality of an app, and then add additional components only as needed. In all of these examples, there is a core set of app components always used but then based on where a user navigates additional components are required. This does not change the size of a slice on the server side. However, it demonstrates that fetching subsequent components does not have to entail getting full slices. This further justifies the need for slicing to reduce the download requirements of apps . However, the time needed to create those slices also needs to be reasonable.

App slicing time. The time required to perform slicing is important because there are 3.9 million apps available from

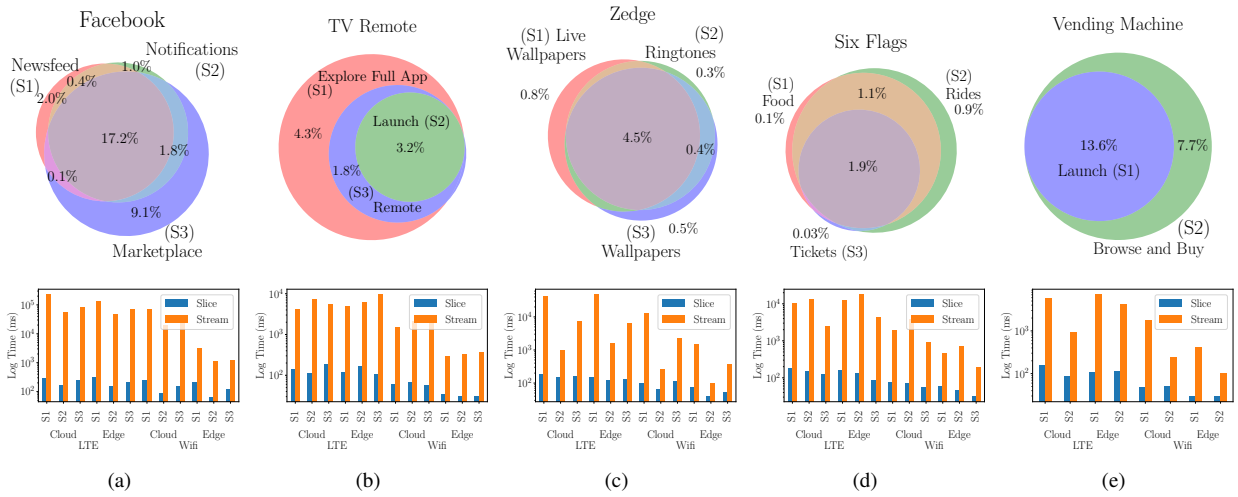


Figure 4. Showing app slices for 5 apps and comparing their delivery performance with full streaming.

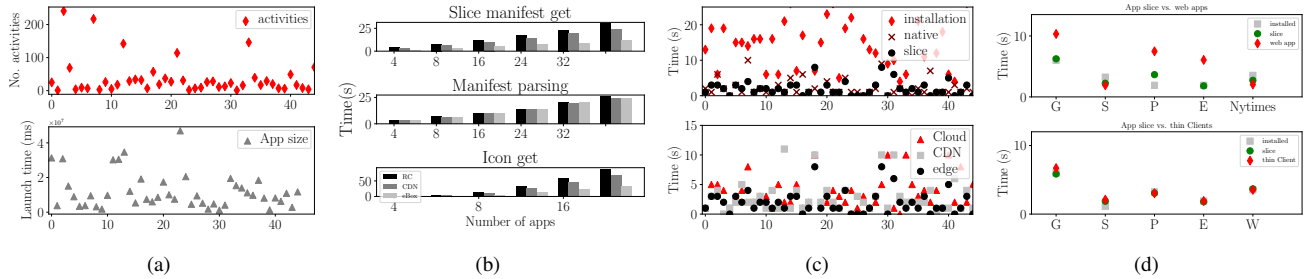


Figure 5. (a)-top Showing number of activities in apps and (a)-bottom aggregate size of app slices relative to original app package (b) Showing the time taken to deliver app slice manifest; (c)-top Comparing lower bounds of installation time (using adb over USB and launching app) and launch of installed apps vs. app slice launch using AppSlicer server, (c)-bottom deployment of AppSlicer server at different infrastructure points, (d)-top Comparing launch time observed for app slice using remote cloud v/s web apps and (d)-bottom with thin client apps.

the Google play store for which slicing may be carried out. The actual time to perform app slicing varies based on the app and the number of activities it has, as well as on the AppSlicer servers. Using a desktop grade machine (the same machine as the simulated CDN in experimental setup), we measured that, for most apps, slicing was performed in the order of a few seconds but always remains less than a minute. The majority of time is spent in decompressing and repackaging, which is done offline and hence, not limiting AppSlicer.

Developer effort. We report that creation of app slices from Android native apps is automatic and has *zero developer effort*. We created app slices from the *top 5000 apps* downloaded from the Google Play store and were able to run those slices on end user devices as if they were installed natively . As stated in Section 3, creating Instant Apps requires substantial developer effort. Contrary to that, in order to integrate AppSlicer on a device, our new Android API is used by the relevant system apps. The proposed support for app streaming, runtime integrity checking and app ephemerality in Android, does not require any change in the app source code which eliminates any burden on the app developers.

Contextual app (size)	install (s)	download (s)	Slice (s)
Hello World(H) (0.78 MB)	2.6	6.6	6.3
Vending(V) (1.3 MB)	3.6	10.3	6.1
TV Remote(R) (7.1 MB)	11.5	15.8	8.8
Conference(C)(9.7 MB)	13.4	18.8	9.4
Taxi booking(T)(13.2 MB)	14.6	20.8	6.3

Table 4. Comparing minimum time to use for native contextual apps vs. app slices launch. Note that time to use for native apps = (download time + install time + launch time) and time to use for app slices are simply their launch times after being delivered on-demand.

7.2 App slice delivery performance

App slice delivery invariably depends on the size of the app slice.

App slice size. For the workload apps, Figure 5(a) shows the total number of activities in the app (top) and the aggregate size of app slices compared to the total package size (bottom). It shows that the app storage footprint increases with the number of activities. The overall increase in the aggregate app size due to app slices is as expected, as it removes the

optimization to reduce the size of an app in the dex file format, which packs all classes into one file. That is offset by only using a small number of static resources for each activity which, we pointed out already, is typically only a fraction of the overall app package. As a result, there is no size bloating on the client side. It is important to note that when clients use different slices of the same app in succession, redundant app components are fetched only once because their availability is registered in the runtime environment.

Time-to-use (Launch time). We measure delivery performance as the time to deliver the app manifest, display the app, and start using an app slice, shown in Figure 5(b). We use launch time as a metric for measuring the barrier to use because it is crucial for a good user experience. For web apps, it is the web page load time metric. Also, launching is the heaviest activity in terms of responsiveness, requiring the largest number of app slice components on a device.

vs. native apps. For native apps, the minimum time required for a user to use an app equals the sum of the app install time and the app's first launch time, i.e., ignoring completely the time taken to download an app from an app store over the Internet. In contrast, for AppSlicer's app slices, it is simply its launch. We measured it by installing apps on a phone from a local machine, where the apps were downloaded apriori, using Android's debug bridge (ADB) [1]. To highlight the improvement in user experience, in Figure 5(c) we compare this time for native apps installed over ADB and native app slices over WiFi network, for all apps in the workload except for the "scenario" apps listed in Table 3.

vs. web apps and thin clients. We used 5 apps that are known to pressurize different devices listed in the bottom of Table 3, to compare the time to use an app slice to other potential technologies that can be used to offer similar functionality – web apps and thin clients. We chose these apps because both web and native versions were already available. We compared app slices served from an AppSlicer server on a remote cloud to web apps, for it is the only fair comparison to serving web apps. We used a developer version of Chromium to access these web apps while controlling the cache behavior. Figure 5(d) (top) shows the responsiveness in terms of launch time for these apps, not including the browser launch time. Due to lack of standard thin client servers for Android, we implemented a thin client using screen captures that are sent to clients using the VNC protocol. We ran the workload apps on Android-x86 running in a VM connected to a local wired network – best case for thin client apps. For fair comparison, we used the same VM to run the AppSlicer server. Figure 5(d) (bottom) shows the launch time for our 5 apps. We captured launch time inspired by the web page speed index calculation [37]. We recorded the thin client session on the device and then manually subtracted the time between when the app is launched and the screen completely fills. To be fair, we kept the screen recording running while we ran AppSlicer as well. As shown in the figure, the location of the AppSlicer server has an impact on performance, so we next evaluate app slices with different deployment models.

Different deployment models. Figure 5(c)-bottom shows the comparison of launch time when running the AppSlicer server from an edge server, CDN, or from a remote cloud. The findings highlight that when served from the edge tier, app slices can provide responsiveness comparable to native installed apps. However, launch time does not paint a complete picture as responsiveness may be usage dependent. This also leaves out the question on whether slicing has benefits over simply streaming all needed app components separately.

Full streaming vs. slice based delivery. To address this, we ran experiments for the same 5 representative apps with typical usage scenarios discussed earlier. We measured the difference between downloading each individual component separately in a slice vs. downloading the slice as one object. We used both WiFi and LTE networks and served our content from both edge and cloud locations. For our experiment each slice only consists of what is not already "cached" on the device, i.e., consecutive slices contain only distinct components. We made sure the server cache was warm and the device cached was cleared before recording the delay. The network connection setup overhead and cache misses caused by streaming each component independently made downloading a slice as a whole much faster, as evident in Figure 4-bottom. Even for the largest Facebook slices, streaming the whole slice took worst case 300ms. Most of the slices however were able to be streamed much faster.

7.3 App slice execution performance

In addition, for the same 5 apps, we measured the runtime performance of the different app versions, in terms of responsiveness to user input and resource consumption. We chose these because they are known to exercise certain resources, as listed in Table 3. We carried out our measurements using Trepn Profiler built for Qualcomm's Snapdragon mobile processors used in the Nexus 5 smartphone. We compare CPU load, memory consumption and power consumption of installed apps vs. app slices. During the experiments, we exercise the UI manually for at least 2 minutes in typical ways. For instance, with the Chrome app, we launched the browser, opened a website (nytimes), scrolled the webpage, followed a predefined link on that website, opened a second tab, opened another website, then closed the second tab during both runs. We made sure to follow the same steps and clear any on-device cache of these apps before running experiments in both cases. Figure 6 shows the results which can be summarized as follows. App slices provide same or better runtime performance compared to other technologies. There is no significant difference in resource consumption for an installed native app vs. an app slice. Native apps fetch ads, updates, etc., requiring use of network (and power), which is another reason for the similar resource footprint. App slices have reduced execution time and memory requirements compared to

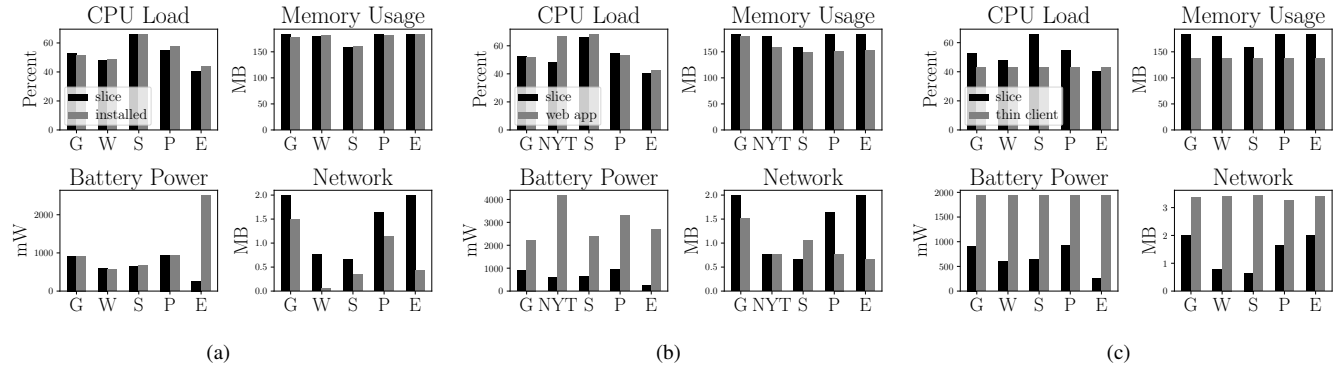


Figure 6. (a) Comparing app slices using the edge vs. native apps; (b) Comparing apps slices using remote cloud vs. web apps and (c) app slices using the edge vs. thin client in terms CPU load, memory usage, battery power and network use measured during execution.

web apps. We attribute these to the number and size of web app components (JavaScript, HTML, CSS, etc.) to be fetched and the additional processing (DOM objects, handling dependencies and rendering) on the device. Finally, thin clients use the least amount of on-device resources as they only keep at-most two memory buffers (in case of double buffering in clients) sized to store a RGB image of the screen. Despite lower memory and less processing, thin clients consume more power because of their aggressive use of the network.

7.4 Quality of experience

To get user feedback on AppSlicer, we recruited 100 participants via Amazon's Mechanical Turk. They were shown two videos, each illustrating two user experiences in succession – first obtaining a native app and second streaming an app slice to perform the same task. The videos are hosted at https://goo.gl/fuxAQQ and https://goo.gl/Ra6Fhd.

The tasks have been chosen to illustrate use cases which are performed only intermittently so that the user is not likely to have the corresponding apps installed on their phone already. In the first one, the user is looking for an app to control a TV, while in the second one, they want an app which they can use to buy products from a vending machine.

The user opens our facilitating app which has a chatbot like interface and types in what they want to do. For sake of clarity, let's call this chatbot as SmartAgent. In the backend, the SmartAgent sends this text to a model trained via Language Understanding Intelligent Service (LUIS), which is a part of Microsoft Cognitive Services. The model analyzes the user's intent from the text and sends it to our server which has multiple applications indexed on 'intents'. In the first case, the server sends back the links to install the relevant apps from PlayStore, or any other repository, which if clicked downloads and installs the apps. . In the second case, the server presents app slices, which, if selected are streamed and launched. All our surveyees were above 18 years old and owned a smartphone. We simply asked a question of what do they prefer and why? We carried this experiment out with users having different smartphone usage habits and ages, listed in Table 5.

Survey question	**Results**
Age group (years)	18-20(2%), 20-25(29%), 25+(69%)
Usage (hours/day)	<1(14%), 1-5(41%), >5(45%)
Preference	app slices(76%), native apps(24%)

Table 5. Summary of the survey results.

Concerns	**Requirements**
Convenience (61%)	Eliminate install/uninstall step
Phone storage (18%)	Reduce app clutter, space use
Security/Privacy(12%)	Need same as Play store

Table 6. Summary of the survey comments.

Survey results. Table 6 summarizes the users comments about app slices. We found that 76% users preferred app slices over installation for the following reasons: convenience of using an app, reducing clutter and hence, storage usage. The main concern raised by the group not preferring app slices was security and privacy. In the comments section, they further pointed out that if the on-demand delivered app slice is delivered from the Play store, it would address the security and privacy concerns associated with using app slices .

7.5 Ephemerality overheads

Ephemerality overheads arise from two basic operations: (i) logging, which includes writing to the ephemeral database implicitly during streaming of app components, or explicitly via JNI when an app tries to create and/or modify its on-device state, and (ii) processing those logs, by reading from the database and removing the state by either modifying or deleting files. Summarizing, the overheads are accesses to the SQLite3 database, JNI overheads, and the amount of state created by an app slice on devices' local storage. It is worth noting that implementing AppSlicer as changes to the Android app framework allows us to seamlessly provide app

Footprint	Minimum	Maximum	Median
No. of Classes	75	2070	1090.5
Classes (KB)	48	2030	970.0
No. of Assets	1	353	31.5
Asset (KB)	5.5	18829	358.6
No. of Native	0	45	1
Native (KB)	0	29581	17

Table 7. Showing number of ephemeral logs in terms of classes and static assets requested during launch of app slices.

slices access to the local storage, but at the same time, to keep app slice accesses contained to a specific location. This serves two purposes. First, since the framework is responsible fot managing apps' access to local storage, it makes it amenable. Second, there are no additional performance overheads other than what native app incurs. This ensures that support for ephemerality does not incur any additional overheads in the critical path of app slice execution. However, for app slices, additional system state is created and maintained while streaming app components.

Concerning this state, Table 7 shows the number of requests and the total bytes transferred in a form of classes, assets, native libraries or any non-asset app component, during the launch of the 50 apps used in evaluation. We claim that the overhead introduced to support ephemerality is negligible, considering the benefits. In favor of brevity, we omit two set of results from this paper (i) benchmarking results of JNI overheads and SQLite3 performance and (ii) amount of on-device state of created by apps on local storage, which has been studied extensively in previous research [22, 24–26] around the impact of storage access patterns on app performance.

7.6 Developer Generated Slices vs. AppSlicer

We conducted an empirical comparison of developer generated slices with AppSlicer generated app slices. We used Google's Instant Apps to as proxy for optimal developer generated slices. We observed that the available Instant Apps broadly fall in for following three categories.

The first category includes Instant Apps which are just toned down versions of the original app with limited functionality. We chose Buzzfeed's Instant App as representative of this category. The app only displays a limited subset of the original app content in a webview without any menus. The second category is of Instant Apps which carry out a specific action from the original app. We chose UN's Share The Meal app, which is used to donate money to various humanitarian organizations, as representative of this category. The Instant App can be used to donate money quickly in the same way as the original app but forces a full app download if the user would like to perform other app functions. The third group comprises Instant Apps which are simply advertisements or demos for the full app. We chose Dotloop's Instant App that only consists of their app demo and requires a full install if the user would like to actually use the app.

App	Full App (MB)	Instant App (MB)	App Slice (MB)
Buzzfeed	20.29	4.37	1.92
Share the Meal	30.14	4.20	4.24
Dotloop	30.89	4.38	2.17

Table 8. Showing size comparison of Instant Apps vs App Slices for the same functionality.

We compare the total size of the app slice necessary to match the Instant App's functionality by performing the same actions in the original app and monitoring the resource use. As seen in Table 8, AppSlicer-generated app slices are around half the size of the corresponding Google Instant App for Dotloop and Buzzfeed, and approximately the same size for the UN's Share The Meal app.

It is clear from these numbers that the benefits from AppSlicer are very compelling with regards to the real-world applicability of a system such as this. Although, the evaluation is limited to 3 apps primarily because of limited number of available Instant Apps and difficulty in being able to statically analyze them. Since instant apps are only available from Google Play store, it wouldn't be a fair to compare their runtimes with app slices regardless of whether they are delivered from Google cloud platform or their edge cache. So, we did not include runtime runtime performance comparison with Instant Apps.

However, important to note here is that AppSlicer created slices do not necessarily have to replace Instant Apps. Instead app slices could be thought of as automatically created Instant Apps. In that sense, AppSlicer can complement Instant Apps by reducing developer effort in creating better on-demand experiences.

7.7 Evaluation Summary

The key takeaways from experimental results are:

1. Feasibility (§7.1) and utility (§7.4) of app slices without any additional developer effort.
2. App slices beat other technologies in delivery (§7.2) and are at least on par in execution performance (§7.3).
3. App slices are at least as secure as native apps with ephemerality supported at modest costs (§7.5).
4. AppSlicer generated slices are just as small if not smaller that developer generated slices while providing the same functionality (§7.6).

8 Related work

Program Slicing. A large body of work has followed work on program slicing [39] after the initial proposal of using this technique for debugging purposes [41] . Most work in this space has focused on the algorithmic aspect of finding optimal slices, which is a known intractable problem. Instead, we present a practical application of program slicing, made

possible by backing it up with dynamic app streaming. Other applications of program slicing include program analysis for predicting performance, debugging, resource consumption, and software configuration management [28]. In principle, we apply a similar methodology but apply it to Android apps. Further, we do not use program analysis instead rely on slicing on execution boundaries, closely related to dynamic slicing [2], to achieve the goal of reducing developer effort toward on-demand delivery of app functionality.

On-demand application delivery. Prior work on on-demand application streaming includes 'thin' desktop clients [7, 32, 38, 42], Numecent's cloud paging [13], Instart logic's [21] webapp streaming, or Amazon's mobile app streaming using a wrapper SDK. Our own prior work on native app streaming to mange app updates [9] assumes already installed apps Compared to this, app slicing is more than just making app components available for already installed apps. It is more similar to the manual slicing proposed in Apple's app thining functionality [4]. Google's Instant Apps [20] provides on-demand delivery, as discussed already in this paper, but is limited due to the burden it puts on developers to redo their apps. More recently, WeChat and Alipay announced apps without installation in a similar direction, but these also require developer effort of implementing their existing apps for the corresponding frameworks. Compared to them, AppSlicer's focus is on automatically creating and delivering app slices as opposed to full apps. Finally, none of those works address app clutter created by such apps.

Application partitioning. Prior work on system level application partioning was presented in [19] for Microsoft's applications based on the COM model. A system to split apps across mobile phones and cloud was put forward in [18]. [31] proposed optimal partitioning of apps among different kinds of devices for sensornet applications. These efforts are primarily concerned with minimizing the apps' resource footprint on a mobile device. In contrast, AppSlicer is primarily focused on creating the infrastructure for dynamic, on-demand delivery of app slices, particularly for apps with transient usage; AppSlicer could however benefit from more sophisticated app slicing methods such as those developed in the above prior work. App partitioning has also been leveraged with the goal of offloading computation from mobile apps to cloud [12, 14]. In contrast, AppSlicer works in the opposite direction of reducing functionality that is delivered to and runs on a device, as opposed to offloading computation from apps on user devices. [17] proposes dynamic partitioning of apps between cloud and mobile devices to achieve better efficiency. Compared to that, AppSlicer takes a more practical approach in that it partitions apps based on minimum needed app components to be delivered backed up by streaming.

Ephemerality. App slices ensure that app components and state are erased. However, it does not make any strong tracking and erasure of app traces required for deniability guarantees, such as what is done in Lacuna [15] to guarantee forensic deniability. The ephemeral community 4chan [8] is known for ephemeral content streaming at scale; compared to that AppSlicer can be seen as streaming of ephemeral app slices.

In summary, we believe that AppSlicer is an effort that builds on the above discussed works to present a system for Android apps that shows existential proof of a potential solution to the difficult problem of automatically creating on-demand deliverable apps in a manner that reuses the momentum of the existing native app ecosystem.

9 Conclusions

The previous decade saw refactoring of online web services to mobile apps, triggered by the introduction of smart phones. Now, apps are moving towards on-demand delivery, disrupting the app ecosystem. Any such disruption can be costly in terms of requiring mobilization of app developers, risky in terms never gaining momentum, and can take time to mature. In this paper, we propose one path – AppSlicer– to maintain the momentum of the native app ecosystem and to create new ways to deliver functionalities to end users via app slices. We demonstrate that automatic slicing of existing native apps is feasible and better in terms of its performance on end user devices than web apps and thin clients. More importantly, it removes the developer effort drastically when compared to approaches such as Google Instant Apps. Going forward we envision an intelligent, malleable app ecosystem, where functionalities can be sliced, tagged, classified according to different context, and delivered to multitude of devices rather than just phones, even "before" they are required. On a broader level, this will allow researchers and industry to focus on new things rather than repackaging what has already been done before in some other form.

References

[1] Android Debug Bridge. http://goo.gl/Mzvl3j.

[2] AGRAWAL, H., AND HORGAN, J. R. Dynamic Program Slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1990), PLDI '90, ACM, pp. 246–256.

[3] Dependency Graphs of App Apks. https://github.com/alexzaitsev/apk-dependency-graph.

[4] Working with App Thinning. https://www.appcoda.com/app-thinning/.

[5] Top 6 Android App Managers. https://drfone.wondershare.com/android/android-app-manager.html.

[6] Barcelona Municipal Apps. https://ajuntament.barcelona.cat/apps/en/.

[7] Microsoft AppV. https://docs.microsoft.com/en-us/windows/application-management/app-v/appv-for-windows.

[8] BERNSTEIN, M. S., MONROY-HERNÁNDEZ, A., HARRY, D., ANDRÉ, P., PANOVICH, K., AND VARGAS, G. G. 4chan and /b/: An Analysis of Anonymity and Ephemerality in a Large Online Community. In *ICWSM* (2011).

[9] BHARDWAJ, K., AGARWAL, P., GAVRILOVSKA, A., SCHWAN, K., AND ALLRED, A. AppFlux: Taming App Delivery via Streaming. In *2015 Conference on Timely Results in Operating Systems (TRIOS 15)* (Monterey, CA, USA, 2015), USENIX Association.

[10] BHARDWAJ, K., GAVRILOVSKA, A., AND SCHWAN, K. Ephemeral Apps. In *Proceedings of the 17th International Workshop on Mobile Computing Systems and Applications, HotMobile 2016, St. Augustine, FL, USA, February 23-24, 2016* (2016), pp. 81–86.

[11] BHARDWAJ, K., GAVRILOVSKA, A., STEINER, M., FLACK, M., AND LUDIN, S. DRIVESHAFT: Improving Perceived Mobile Web Performance. *ArXiv e-prints* (Sept. 2018).

[12] CHUN, B.-G., IHM, S., MANIATIS, P., NAIK, M., AND PATTI, A. Clonecloud: Elastic Execution between Mobile Device and Cloud. In *Proceedings of the sixth conference on Computer systems* (2011), ACM, pp. 301–314.

[13] Numecent Cloudpaging. http://goo.gl/t8soKL.

[14] CUERVO, E., BALASUBRAMANIAN, A., CHO, D.-K., WOLMAN, A., SAROIU, S., CHANDRA, R., AND BAHL, P. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services* (2010), ACM, pp. 49–62.

[15] DUNN, A. M., LEE, M. Z., JANA, S., KIM, S., SILBERSTEIN, M., XU, Y., SHMATIKOV, V., AND WITCHEL, E. Eternal Sunshine of the Spotless Machine: Protecting Privacy with Ephemeral Channels. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)* (Hollywood, CA, 2012), USENIX, pp. 61–75.

[16] Google Streaming App Content. http://goo.gl/mM5HvI.

[17] GIURGIU, I., RIVA, O., AND ALONSO, G. Dynamic Software Deployment from Clouds to Mobile Devices. In *Proceedings of the 13th International Middleware Conference* (New York, NY, USA, 2012), Middleware '12, Springer-Verlag New York, Inc., pp. 394–414.

[18] GIURGIU, I., RIVA, O., JURIC, D., KRIVULEV, I., AND ALONSO, G. Calling the Cloud: Enabling Mobile Phones As Interfaces to Cloud Applications. In *Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware* (Berlin, Heidelberg, 2009), Middleware '09, Springer-Verlag, pp. 5:1–5:20.

[19] HUNT, G. C., AND SCOTT, M. L. The Coign Automatic Distributed Partitioning System. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation* (Berkeley, CA, USA, 1999), OSDI '99, USENIX Association, pp. 187–200.

[20] Google Instant apps. https://developer.android.com/topic/instant-apps/index.html.

[21] Instart Logic - Web Application Streaming. http://goo.gl/D559rq.

[22] JEONG, S., LEE, K., LEE, S., SON, S., AND WON, Y. I/O Stack Optimization for Smartphones. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (Berkeley, CA, USA, 2013), USENIX ATC'13, USENIX Association, pp. 309–320.

[23] Native App Poularity – Kahuna. https://goo.gl/jzovpt.

[24] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Examining Storage Performance on Mobile Devices. In *Proceedings of the 3rd ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds* (New York, NY, USA, 2011), MobiHeld '11, ACM, pp. 6:1–6:6.

[25] KIM, H., AGRAWAL, N., AND UNGUREANU, C. Revisiting Storage for Smartphones. *Trans. Storage 8*, 4 (Dec. 2012), 14:1–14:25.

[26] KIM, J.-M., AND KIM, J.-S. AndroBench: Benchmarking the Storage Performance of Android-Based Mobile Devices. In *Frontiers in Computer Education*, S. Sambath and E. Zhu, Eds., vol. 133 of *Advances in Intelligent and Soft Computing*. Springer Berlin Heidelberg, 2012, pp. 667–674.

[27] Hybrid vs Native Mobile Apps – The Answer is Clear. https://ymedialabs.com/hybrid-vs-native-mobile-apps-the-answer-is-clear/.

[28] LI, Y., ZHU, C., RUBIN, J., AND CHECHIK, M. Semantic Slicing of Software Version Histories. *IEEE Transactions on Software Engineering 44*, 2 (Feb 2018), 182–201.

[29] Why "Progressive Web Apps vs. Native" Is the Wrong Question to Ask. https://goo.gl/2JtYCP.

[30] NETRAVALI, R., NATHAN, V., MICKENS, J., AND BALAKRISHNAN, H. Vesper: Measuring Time-to-Interactivity for Web Pages. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)* (Renton, WA, 2018), USENIX Association, pp. 217–231.

[31] NEWTON, R., TOLEDO, S., GIROD, L., BALAKRISHNAN, H., AND MADDEN, S. Wishbone: Profile-based Partitioning for Sensornet Applications. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation* (Berkeley, CA, USA, 2009), NSDI'09, USENIX Association, pp. 395–408.

[32] Novell - Application Virtualization. http://goo.gl/vjxWYb.

[33] Smart City Berlin. https://www.berlin.de/sen/wirtschaft/en/economics-and-technology/centres-of-technology-zukunftsorte-smart-city/smart-city/.

[34] Smart Dublin. http://smartdublin.ie/.

[35] Markets and markets report smart cities market worth 717.2 billion usd by 2023. https://www.marketsandmarkets.com/PressReleases/smart-cities.asp.

[36] Smart Cities Report Forecasts Trillions in Economic Growth. https://www.smartcitiesworld.net/news/news/mart-cities-report-forecasts-trillions-in-economic-growth-2528.

[37] WebPagetest Speed Index Metric. http://goo.gl/Rw3d5d.

[38] Symantec Workspace Streaming. http://goo.gl/NAA13p.

[39] TIP, F. A Survey of Program Slicing Techniques. Tech. rep., Amsterdam, The Netherlands, The Netherlands, 1994.

[40] Vimeo Blog – Creating an Instant App. https://medium.com/vimeo-engineering-blog/vimeo-android-instant-apps-2f8b1e94760c.

[41] WEISER, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (Piscataway, NJ, USA, 1981), ICSE '81, IEEE Press, pp. 439–449.

[42] Citrix XenApp. http://goo.gl/qbdGPT.