X-Droid: A Quick and Easy Android Prototyping Framework with a Single-App Illusion

Donghwi Kim,* **Sooyoung Park,*** **Jihoon Ko,*** **Steven Y. Ko,†** **and Sung-Ju Lee***
*KAIST †University at Buffalo
Daejeon, Korea Buffalo, NY, USA
{dhkim09, sypark0614, jihoonko, profsj}@kaist.ac.kr, stevko@buffalo.edu

ABSTRACT
We present X-Droid, a framework that provides Android app developers an ability to quickly and easily produce functional prototypes. Our work is motivated by the need for such ability and the lack of tools that provide it. Developers want to produce a functional prototype rapidly to test out potential features in real-life situations. However, current prototyping tools for mobile apps are limited to creating non-functional UI mockups that do not demonstrate actual features. With X-Droid, developers can create a new app that imports various kinds of functionality provided by other *existing* Android apps. In doing so, developers do not need to understand how other Android apps are implemented or need access to their source code. X-Droid provides a developer tool that enables developers to use the UIs of other Android apps and import desired functions into their prototypes. X-Droid also provides a run-time system that executes other apps' functionality in the background on off-the-shelf Android devices for seamless integration. Our evaluation shows that with the help of X-Droid, a developer imported a function from an existing Android app into a new prototype with only 51 lines of Java code, while the function itself requires 10,334 lines of Java code to implement (i.e., 200× improvement).

Author Keywords
Programming by demonstration (PBD); Android; Functional prototyping; Development frameworks

CCS Concepts
•**Software and its engineering** → **Reusability;** *Development frameworks and environments;*

INTRODUCTION
The ability to quickly prototype an application is critical in software development. Using prototypes, developers can evaluate their design decisions in a realistic fashion, solicit feedback from potential users to check if their applications meet the users' specifications or expectations, and shape the final designs and features before releasing actual products. The process of iterating over different prototypes can significantly improve the final user experience and save the cost of having to fix problems after release.

Because of its importance, there are many proposals that aim to enable quick prototyping. Throw-away prototyping [1], opportunistic prototyping [2,3], and patchwork prototyping [4,5] are some of the well-known prototyping strategies that enable developers to produce prototypes rapidly. There are also many commercial tools available for prototyping for domains such as web app development [6–8] and mobile app development [9–11].

When developing a prototype, it is crucial to demonstrate not only the UI of an application, but also the *functionality* of it. This is especially true for mid- to final-stage prototypes. In an early stage of development, it is perhaps acceptable to just show a set of static images of UI mockups or a prototype with limited interactivity (e.g., mockup UI clicks and transitions). However, at later stages of development, it is necessary to use a prototype and evaluate it in real-life situations [5]. A functional prototype is also necessary for software development outside commercial domains, such as academic research, where resource constraints often prevent investing in full-scale software development. Although leveraging open source projects can help in creating functional prototypes, it still requires significant development effort due to the need for extracting (and perhaps sanitizing) code from an unfamilar code base that is potentially large.

We present X-Droid, a framework for Android app developers to quickly prototype experimental features for ongoing app development. X-Droid enables developers to import different kinds of functionality from other *existing* Android apps without having the source code or understanding the internals of those apps. By allowing developers to leverage existing apps' functions quickly and easily, X-Droid enables developers to iterate over functional prototypes and test out different features rapidly. X-Droid makes this possible by adopting *programming by demonstration (PBD)* [12] and combining it with our proposed *background execution of existing Android apps*.

X-Droid adopts programming by demonstration in its development tool, where a developer "demonstrates" a series of

UIST 2019, October 20–23, 2019, New Orleans, LA, USA

ISBN 978-1-4503-6816-2/19/10…15.00
DOI: 10.1145/3332165.3347890

UI actions on an existing Android app. The UI actions are the ones that trigger the functionality that the developer wants to use from the existing app in her prototype. For example, consider a developer of a cookbook app prototype who wants to test out a feature where the prototype shows the delivery dates of necessary ingredients. The developer could do this with X-Droid by demonstrating the UI actions performed on a grocery app that shows delivery dates of selected ingredients. Once a developer demonstrates such UI actions, X-Droid generates a piece of code that the developer can embed into her prototype. This generated code is essentially a series of X-Droid commands that X-Droid executes at run time.

In order to execute those commands, X-Droid implements a new run-time system that directly performs UI actions on an existing app to ultimately execute the function that the UI actions trigger. The salient feature of this run-time system is that it performs all UI interactions with an existing app *completely in the background*, without displaying anything on the screen. For example, consider the same cookbook app prototype mentioned earlier, and suppose that a UI element of the prototype (e.g., "Get Delivery Dates" button) executes a series of X-Droid commands that drive the grocery app. When a prototype tester clicks the button, X-Droid executes all commands on the grocery app without displaying any UI interaction occurring with the grocery app. In other words, the tester still sees the UIs displayed by the cookbook app prototype, not the UIs of the grocery app even when X-Droid is executing a function from the grocery app. This provides what we call a *single-app illusion*—an illusion that a prototype tester is interacting with a single-app, which is important when evaluating the UX of a prototype.

There are many prototyping tools available for mobile app development, but they are limited to creating UI mockups [9–11]. Moreover, previous work on creating functional prototypes in general has focused on web apps, not mobile apps [3]. X-Droid's goal is to support developers who want to create functional Android app prototypes.

The main contributions of X-Droid are as follows.

- We design and develop a new Android app development tool for quick and easy prototyping of functional app features. X-Droid empowers developers to import app functionality from other apps without requiring source code or understanding the implementation of the functionality.

- We develop a technique to execute an existing app's functionality completely in the background. The novelty of our technique lies in converting user-visible, foreground tasks into background ones. Our technique works with existing apps on off-the-shelf Android devices; we do not impose any disruptive barrier to entry, such as operating system modifications or source code access to existing apps.

- We evaluate the usefulness of X-Droid by conducting a developer study with five Android developers (two professionals and three university students) involving app feature prototyping. Our results show that X-Droid is easy to use, enabling a participant to import an app function from an existing app by writing only 51 lines of Java code. Without X-Droid, the participant would have needed to migrate 10,344 lines of Java code from an open-source code base (i.e., X-Droid provides more than 200× improvement).

RELATED WORK

We classify our related work into three categories—(i) technique that provides new functionality with UIs playing a central role, (ii) techniques for rapid mobile app development, and (iii) techniques that enables alternate forms of Android app execution.

UI-based Functionality Provisioning

Programming by demonstration (PBD) has been extensively used for end-user programming on mobile apps. For example, SUGILITE [13] allows smartphone users to develop custom smartphone automation scripts by recording UI actions. uLink [14] helps users define custom deep links in uLink-enabled apps. As it partially records and replays UI actions to navigate to an activity, deep link execution is similar to X-Droid's execution. However, UI actions occur in the foreground in both SUGILITE and uLink, therefore not suitable for developing a prototype.

For the web, UI-based functionality provisioning has been leveraged for various purposes. CoScripter [15] enables web processes in enterprises (*e.g.*, conference room reservations) to be automated through PBD. Highlight [16] enables end-users to re-create desktop web pages as mobile web pages using PBD. C3W (Clip, Connect and Clone for the Web) [17] empowers users to combine two web apps to develop a new function. However, by the nature of the web, these systems are vulnerable to the changes in the web page design. In contrast, X-Droid allows developers to integrate functions from other Android apps, not web apps, and utilizes Android app package files (APKs) that do not change once compiled.

On the other hand, general UI automation has been used mainly for mobile app testing [18–24]. As they typically generate random UI actions with monkey runners or replay recorded UI actions to cover various test cases, they are different from X-Droid that focuses on easy programming of intended UI actions.

For web apps, web drivers [25–27] are popular for automating web testing. Similar to X-Droid, they support so-called a "headless" mode of execution, where they perform UI interactions on a web page in the background without displaying anything to their users. In that sense, one can say that X-Droid supports a headless mode for driving Android apps.

UI mashup creation systems [3, 28–32] are also popular and they allow developers to compose existing UI of web pages or apps in a new layout. X-Droid is different from them as it allows developers to implement a new function using existing apps' functions.

Modular Programming of Mobile Apps

App Inventor [33] and Thunkable [34] allow users to create new apps by composing existing program modules through visual programming. As they ease app development, they could

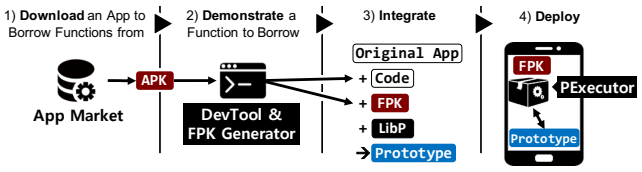


Figure 1: X-Droid usage model.

be useful for functional prototyping of mobile apps. However, App Inventor and Thunkable only allow developers to re-use App Inventor or Thunkable modules developed within their own frameworks while X-Droid allows developers to borrow functions from any existing Android app.

Alternative Forms of Android App Execution

Techniques that enable alternative forms of execution on Android generally design an app that dynamically loads and executes another Android app. For example, Parallel Space [35] and others [36–39] allow a user to clone already-installed apps in a virtual app space and execute them as separate copies. This is useful when using an app with different configurations (e.g., different login profiles). Dynamic APK loading frameworks [40–44] split an app into smaller chunks and dynamically fetch and load each chunk as needed. This method reduces an app's binary size and speeds up the booting time. The biggest advantage of X-Droid over above techniques is background execution as these techniques execute other apps in the foreground.

X-DROID OVERVIEW

Figure 1 summarizes a prototype development process using X-Droid. We present X-Droid's three design principles (DP) and app development steps with the following example use case scenario: Alice and her team are developing a chatting app. Alice has an idea of a new app feature, smart snoozing, that snoozes chat notifications while users are asleep. To collect feedback from her team members, she decides to quickly build a functional prototype with X-Droid.

DP1 *Utilizing existing apps without manual modification*

To detect whether a user is sleeping, Alice wants to import sleep tracking functionality from an existing app. From an online app market, she finds a sleep tracker app that uses smartphone sensors to infer the user's sleep state. She downloads the sleep tracker app and uses the app's package file (called an APK file on Android) as input to X-Droid.

DP2 *Providing ease of programming for developers*

After obtaining the APK file, Alice feeds the file to X-Droid's developer tool named DevTool running on her computer. DevTool enables Alice to demonstrate UI actions that trigger the sleep tracking function provided by the downloaded sleep tracker app. To do so, DevTool launches the sleep tracker app in a special Android emulator that Alice can use to demonstrate UI actions. Using the emulator, Alice navigates through the sleep tracker app's UIs to where sleep states are displayed.

Alice then informs DevTool that the UI action demonstration is finished and DevTool generates a corresponding code segment. This code segment is in Java and uses X-Droid's API

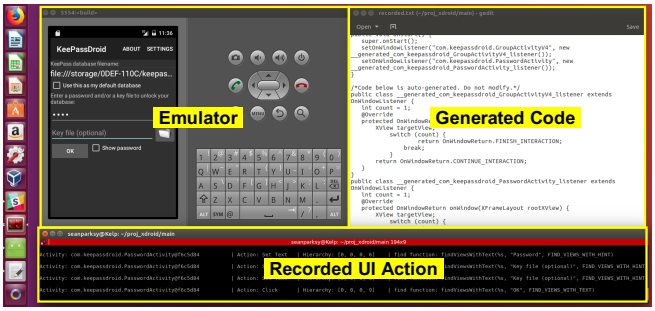


Figure 2: Screenshot of development assistance tool.

to describe X-Droid commands that perform demonstrated UI actions. Alice embeds this code segment into a new method that she implements for her prototype, `isSleeping()`, which returns `true` if 'sleeping" is displayed or `false` otherwise. The code generated by DevTool uses X-Droid's API, which is provided as a library (named LibX). Thus, Alice links LibX with her chatting app.

Alice then runs a tool called FPK Generator and provides the sleep tracker app's APK file as input to the tool. As we discuss in the 'Borrowed Function Execution' section, FPK Generator transforms an APK file into a X-Droid-compatible form named Functionality Package (FPK). At run time, X-Droid uses this FPK file to execute the sleep tracking functionality that Alice uses in her prototype app.

DP3 *Making new apps easily deployable on off-the-shelf Android devices*

To test Alice's new prototype, she and her team install two apps; Alice's new prototype and a special Android app named XExecutor provided by X-Droid. After installing the two apps, Alice or her team member uses the new prototype just like any regular Android app. When the prototype app must execute the embedded sleep tracking functionality, the app communicates with XExecutor and XExecutor executes the function in the background using the sleep tracking app's FPK. XExecutor does all of the above as a regular Android app and there is no need to implement anything in the Android OS. This makes it possible to easily deploy our solution on off-the-shelf Android devices. We detail the exact mechanism of our background execution in the 'Borrowed Function Execution' section.

PROGRAMMING BY DEMONSTRATION

X-Droid provides an API and DevTool that enable developers to convert UI actions into a piece of code. DevTool enables *programming by demonstration*; using DevTool, a developer can launch an Android emulator, install a provider app, and visually interact with the provider app as a regular user. As the developer interacts with the provider app, DevTool records all interactions and automatically generates a Java class called `XTask` that implements the interactions using X-Droid API (we detail `XTask` and our API in later sections).

To implement this mechanism, we modified the Android emulator to capture and log all UI interactions (*i.e.*, touch events, key press events, etc.) in the Android `View` class that rep-

resents a UI element. `View` class is the root class for all UI classes on Android and all UI elements are first delivered to it. Thus, instrumenting `View` allows us to catch all UI interactions. Figure 2 shows a screenshot of logged UI actions and corresponding code generated automatically.

X-Droid assists developers to generalize code created through PBD, handle unpredictable UI paths, and identify UI elements in the code. We detail each below.

Generalization: Since X-Droid generates Java code for demonstrated UI actions, a developer can modify the code to generalize as necessary. For example, if a developer searches "New York" on a bus ticketing app during a demonstration, a corresponding Java code segment with hard-coded "New York" string will be generated. The developer can generalize search queries by replacing the string with a variable.

Handling unpredictable UI paths: For UI actions that could potentially take different program paths due to non-determinism (e.g., input variation, uses of random numbers, etc.), X-Droid assists developers by allowing them to register a fallback callback on unhandled UI events and output. Developers can thus handle unpredicted failures and unintended executions. Developers can also use it for debugging. We discuss details in 'XTask' and 'Error Handling' sections.

Identifying UI elements: It is possible to identify a UI element in multiple ways, e.g., by its index or text. By default, DevTool uses a text string to identify a UI element if the string is unique screen-wide. If not, DevTool uses the index given by its parent element. If desired, developers can review and revise auto-generated Java code to choose the proper UI element specifications. We detail X-Droid API for identifying UI elements in the 'Navigating through UI Objects' section.

APP DEVELOPMENT WITH X-DROID

While X-Droid provides DevTool for programming by demonstration, internally it provides an API that enable developers to manually convert UI actions into code if they desire more control. X-Droid supports a developer in the following ways.

1. X-Droid API empowers a developer to declare her desired functionality in an executable program block. In the block, the developer programs UI actions (e.g., clicking and typing) to trigger the desired functionality of a provider app. The result of the execution displayed in the UI can be retrieved from a callback, which is invoked when all programmed user actions are completed.

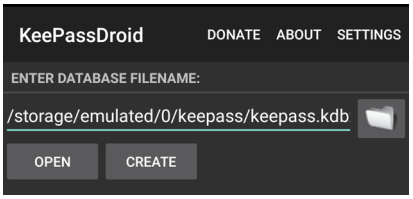
2. X-Droid API provides proper abstractions to minimize the development effort. For example, choosing a UI element to click and waiting for a new activity[1] to come up are trivial tasks for users. For developers, however, these tasks lead to a data search problem (i.e., finding a UI element) and a synchronization problem (i.e., waiting for a new activity to come up). Using simple-yet-expressive abstractions, X-Droid API provides ease of programming.

3. X-Droid API provides an error-handling mechanism that enables developers to catch asynchronous error notifications. Developers can catch both visible (e.g., notification bar) and invisible (e.g., vibration) error notifications and inspect the state (e.g., color) of a UI element indicating an error.

[1] An activity represents a single window on Android.

```
1: public class MyTask extends XTask {
2:   public void onStart() {
3:     // When MyTask starts, reg. OnActivityListeners
4:     setOnActivityListener(new FSALsnr(),
5:             "com.keepassdroid.FileSelectActivity");
6:     // Omitted more OnActivityListener reg.
7:   }
8:   private class FSALsnr extends OnActivityListener {
9:     @Override
10:    protected OnActivityReturn onActivity(
11:            XViewGroup rootXView) {
12:      ArrayList<XView> views = new ArrayList<>();
13:      // Search a view with "open" text
14:      rootXView.findViewsWithText(views, "open",
15:                        FIND_VIEWS_WITH_TEXT);
16:      // Finish MyTask if the view is not found
17:      if (views.size() < 1)
18:        return OnActivityReturn.FINISH_XTASK;
19:      // Click the view
20:      views.get(0).click();
21:      // Wait for a next activity triggered by click
22:      return OnActivityReturn.WAIT_FOR_NEXT_ACTIVITY;
23:    }
24:  }
25:  // Omitted more OnActivityListener def.
26: }
```

(a) XTask for opening default DB file using KeePassDroid.



(b) KeePassDroid's file select activity. (Empty space is cropped.)

Figure 3: Code snippet for interacting with KeePassDroid.

XTask

X-Droid API defines a class called `XTask` that a developer extends to implement user interactions on a provider app. `XTask` is a unit of execution and a developer starts a new `XTask` in her app when she wants to execute a borrowed function. Every `XTask` should describe a sequence of emulated user interactions starting from the main activity (*i.e.*, the first activity launched when an app starts) of a provider app. `XTask` always gets executed from the main activity with a clean initial state (*e.g.*, an empty data directory and local DB), freeing developers from the side effects of previous `XTask` executions.

`XTask` defines two lifecycle callbacks, `onStart()` and `onFinish()` that a developer can implement. As the names suggest, `onStart()` is invoked at the beginning of an execution and `onFinish()` at the end. For each activity that a developer wants to interact with, the developer must first register a callback for the activity within `onStart()`. If a callback is registered for an activity, it is invoked when the activity is launched. Callback registration is done by calling `XTask.setOnActivityListener()` and supplying an activity name and a callback.

The rationale behind this callback-based design is two-fold. First, since the developers are Android developers, it is natural to provide an event-driven programming model similar to that of Android [45]. Second, as callbacks effectively capture event trigger points (e.g., a new activity), it is also intuitive to use.

Figure 3a shows a custom `XTask` as an example and Figure 3b shows a screen shot of an activity that `XTask` uses. Our `XTask` interacts with an open-source Android app called KeePassDroid [46], which is a password manager that keeps usernames and passwords in a DB. `XTask` performs a task of opening the default DB from KeePassDroid. To do so, `XTask` implements `onStart()` and registers an activity listener for each activity that it wants to interact with. For example, line 4 in Figure 3a calls `setOnActivityListener()` with an activity name and an activity listener. Lines 8-24 show an example activity listener that implements `onActivity()` that is invoked when the associated activity is launched.

`onActivity()` should return one of the following three values. A developer should return `WAIT_FOR_NEXT_ACTIVITY` when a subsequent activity launch is expected, `RETURN_TO_PREV_ACTIVITY` when `XTask` should continue at the previous activity, and `FINISH_XTASK` when an `XTask` should finish.

Navigating through UI Objects

In order to enable developers to programmatically interact with the UI elements that trigger the execution of borrowed functions, we provide a hierarchical set of classes that mirror Android's UI classes. On Android, the `View` class and its subclasses represent UI elements such as scrollers, buttons, text boxes, etc. In our design, we represent UI elements with `XView` and its subclasses that exactly mirror Android's UI classes. For example, a button is represented by `XButton` in X-Droid and `Button` in Android. Some apps define custom UI elements by extending Android UI classes; e.g., an app can extend `Button` to define its own button. When this occurs, we represent it using the closest ancestor in the class hierarchy.

We also provide `XViewGroup`, a mirror of Android's `ViewGroup`, which can contain a hierarchy of UI elements using nested `XViewGroup` objects and `XView` objects. When a new activity starts, we invoke `onActivity()` and pass an `XViewGroup` object that contains the hierarchy of all UI elements within the activity. To further assist developers, we provide navigation helper methods such as `getChildAt()` that enables access to a direct child contained in an `XViewGroup` object. Table 1a lists our navigation helper methods. X-Droid API is designed to adopt class and method convention from Android API as much as possible to ease the learning curve.

To perform UI actions on UI elements, developers can call UI action methods implemented in `XView` and its subclasses. For example, `XTextView` implements `readText()` that returns the text it contains. Table 1b lists some UI action methods we define (we do not list the entire set due to space constraints). We define these methods using Java interfaces to customize the behavior of a UI action for each UI element.

Our design allows developers to interact with UI elements directly at the object level. This is more advantageous than the traditional pixel-level approaches, where developers use X-Y coordinates to locate target UI elements [47–50]. These approaches have a limitation of not supporting different devices easily as device screen sizes and resolutions differ widely.

We further assist developers to inspect and identify any UI elements in a provider app through DevTool. For example, a developer can open an Android emulator and click a UI element of an activity, at which point DevTool displays the information on the UI element and the full path to the UI element on the activity UI tree.

Passing Parameters & Retrieving Results

Developers can pass parameters for `XTask` execution and retrieve the results shown in a UI in the same way that they interact with the UI. For example, a developer can write an `XTask` for purchasing a movie ticket using a theater app and pass a movie name by using the `setText()` method targeting the theater app's search view. Similarly, she can retrieve a ticket number on the UI by using the `readText()` method.

Error Handling

X-Droid API provides callbacks that can catch both visible and invisible error notifications (*e.g.*, pop-up messages notifying no Internet access, vibrations indicating wrong password inputs, etc.). X-Droid hijacks Android APIs that could be used as error notifications such as toasts, notification bar updates, and vibrations, and invokes corresponding callbacks where developers can implement their own error handling logic.

X-Droid also provides a callback for handling crash and ANR (Application Not Responding) errors during borrowed function execution. To prevent `XTask` from proceeding in directions unintended by developers, X-Droid provides another callback that is invoked when it cannot find any registered activity listener of a newly launched activity.

Optimizing the FPK Size

Since an FPK must be distributed with a developer app, reducing its size is important to make the app lightweight. X-Droid provides an FPK size optimization tool, FPK Optimizer, for that purpose. When a developer runs the tool, it asks her to select activities in an FPK that are necessary for her app. It then removes other activities from the FPK and runs a bytecode optimization tool, ProGuard [51], to remove all classes and methods that are never referenced.

FPK Optimizer also removes unused resources (*e.g.*, images). Android apps usually have multiple versions of each resource for different display configurations (*e.g.*, size and DPI). Since X-Droid does not display anything, FPK Optimizer removes all resources except one copy of each resource for default configuration.

APP EXECUTION WITH X-DROID

Once an app is developed with X-Droid and installed on an Android device, X-Droid executes the functionality that it borrows from provider apps. X-Droid's app execution involves three components, LibX, XExecutor, and FPK Generator as shown in Figure 4.

LibX implements X-Droid API and is linked to a developer's app at compile time. It communicates with XExecutor to send

Table 1: Available methods in X-Droid API.

(a) Navigation methods implemented by `XViewGroup`.

Method Prototype	Description
`int getChildCount()`	Get the number of children
`XView getChildAt(int idx)`	Get the view at the position
`void findViewsWithText(ArrayList<XView> outViews, CharSequence text, int flags)`	Finds the views that contain a given text
`int indexOfChild(XView child)`	Get the position of a child

(b) Available UI action methods.

Interface	Method Prototype	Description
`XClickable`	`void click()`	Click the `XView`
`XAdapter`	`void clickItem(int idx)`	Click i-th child
`XTextReadable`	`String readText()`	Read text
	`String readHint()`	Read hint text
`XTextSettable`	`void setText(String text)`	Set text

Figure 4: X-Droid execution overview.

UI action requests and receive the action results. XExecutor loads FPKs generated by FPK Generator, receives UI interaction requests from developer's apps, executes them, and returns the results. It is designed to provide an illusion that a developer's app is providing all functionality (i.e., a single-app illusion) to users.

With these components, X-Droid provides the following three benefits desirable for using and testing the app as a prototype.

1. **Avoiding OS modification:** We design XExecutor to be a regular Android app that can be installed and run on any Android device. XExecutor can dynamically load and execute app functions in X-Droid-compatible FPK files, generated by FPK Generator. Therefore, prototype apps developed through X-Droid can be tested on off-the-shelf Android devices.

2. **Providing a single-app illusion:** XExecutor is also designed to avoid user experience (UX) disruption due to app switching as the Android OS requires an app to be foreground when executing UI actions on it. XExecutor provides a new UI framework and a new activity management framework that mirror Android's, and executes a borrowed function completely in the background.

3. **Enforcing privacy and security:** Since provider apps are executed within XExecutor, they are isolated from other user-installed apps as the Android OS isolates each app. In addition, XExecutor internally isolates a provider app from other provider apps. It also prevents permission escalation that might happen while executing borrowed app functions.

LibX: API Library for Developer's Apps

LibX is a library that provides X-Droid API implementation. At compile time, a developer must link LibX to the app that she developed. The main task of LibX is to communicate with XExecutor to execute borrowed functions. For example, when a developer's app creates a new `XTask`, LibX instructs XExecutor to load an FPK and launches its main activity; or when a developer's app invokes a method that performs a UI action for an activity, LibX instructs XExecutor to perform the action on the activity. LibX also communicates with XExecutor to invoke callbacks as necessary. For example, it receives the results of UI actions from XExecutor and delivers them to the developer's app by invoking appropriate callbacks. All communication occurs with IPC between LibX and XExecutor.

Borrowed Function Execution

FPK Generator and XExecutor work in tandem to load and execute borrowed functions on an Android device. FPK Generator is an offline tool that uses bytecode instrumentation techniques [52–58]. It rewrites portions of a provider app to help XExecutor load and execute app logic. It also allows a provider to choose the exact functions she wants to share; using this as input, FPK Generator extracts only the selected functions from a provider app and packages them into an FPK. On the other hand, XExecutor handles all run-time issues of borrowed function execution. Together, they provide a single-app illusion, handle the execution of a provider app, manage permissions and isolation, and optimize execution delay.

Providing a Single-App Illusion

In order to provide a single-app illusion to users, FPK Generator and XExecutor implement two necessary mechanisms. The first mechanism is emulating UI interactions for executing a borrowed function and the second mechanism is converting all foreground tasks of a provider app into background tasks. Since Android always allocates a portion of a screen to a foreground task, if we use only the first mechanism (i.e., emulating UI interactions), a user will see a blank space when the borrowed functionality is running on XExecutor. Combined with the second mechanism, every aspect of executing the borrowed functionality is hidden from the users.

Emulating UI Interactions: Emulating UI interactions consists of two parts. First, XExecutor provides a new UI framework that is a drop-in replacement of the Android UI framework, composed of 496 Java classes. This new UI framework, called the X-UI framework, works almost the same as the Android UI framework; the only differences are that it does not display anything on a device screen and all UI animations are skipped. Since a provider app is a regular Android app that does not use the X-UI framework, FPK Generator rewrites a provider app and forces it to use our UI framework. This rewriting process is straightforward—the X-UI framework uses the exact same class structure as Android and there is a one-to-one mapping between every Android UI class and our UI class. The job of FPK Generator hence is renaming.

Second, XExecutor mirrors system services that potentially display some output on a device screen, e.g., Notification Service, Layout Inflator Service, etc., and modifies their behavior so that they do not display anything. FPK Generator also rewrites the provider app and forces the FPK to use our system services. This is not a complicated task; Android apps access all system services via a call to `getSystemService()`. Thus,

FPK Generator overrides this call and returns our version of a service whenever the original version is requested.

Converting Foreground Tasks to Background Tasks: The second necessary mechanism for providing a single-app illusion to users is converting foreground tasks of a provider app into background tasks. On Android, foreground tasks are activities implemented with the `Activity` class and background tasks are services implemented with the `Service` class. Thus, in X-Droid, FPK Generator and XExecutor work together to convert every activity of a provider app into a service. To do this, XExecutor provides a new class called `XActivity` that is a subclass of `Service`. Although `XActivity` is a subclass of `Service`, hence runs in the background, it mirrors the structure of Android's `Activity` and emulates the behavior of `Activity`. Using `XActivity`, FPK Generator rewrites the provider app and converts every `Activity` to `XActivity` via renaming, effectively converting every foreground task into a background task.

Dynamic Loading of an FPK

We illustrate how X-Droid loads an FPK and executes a borrowed function at run time in two steps.

Step 1: Loading an FPK: XExecutor loads an FPK via `DexClassLoader`, which provides the capability of loading the classes of an app into a different app. In addition, XExecutor loads all resources files (e.g., images and icons) in an FPK. These resources define custom UI layouts and styles specific for a provider app. They are necessary for correct execution as they are referenced by app logic throughout the lifetime of a borrowed function. Android typically uses `AssetManager` to load resources when it starts an app. However, as XExecutor loads an FPK using `DexClassLoader`, its app resources are not automatically loaded. XExecutor thus instantiates a mirrored `AssetManager` that loads resources from an FPK.

Step 2: Executing a Borrowed Function: As discussed earlier, FPK Generator converts every `Activity` to `XActivity` so we can execute original foreground logic in the background. This is possible because `XActivity` is a child of `Service`. The problem is that an activity and a service on Android have different lifecycle callbacks, and we cannot simply convert an activity to a service via renaming and expect it to work. For example, when an activity starts, `onCreate()`, `onStart()`, and `onResume()` are invoked; however, when a service starts, `onCreate()` and `onStartCommand()` are invoked. Thus, we need to resolve these differences.

XExecutor thus has mirrored versions of Android's Activity Manager Service (AMS) and ActivityThread that manage regular activities on Android and manages `XActivity` in the exact same way that Android manages `Activity`. On Android, AMS and ActivityThread control the lifecycle of `Activity`; AMS orders ActivityThread to launch, resume, pause, and destroy `Activity` instances, and ActivityThread invokes `Activity` callbacks. XExecutor's mirrored versions also perform the same task for `XActivity`.

Handling Permissions & Isolation

In order to prevent malicious apps from abusing X-Droid, XExecutor is designed to (i) isolate the execution of borrowed function from other apps on the same device, (ii) isolate each execution of a borrowed function from previous or later executions, and (iii) enforce developer's apps and provider apps to acquire all app permissions required to execute a borrowed function, in order to prevent permission escalation. We note that traditional function sharing methods, such as libraries and sharing via IPC, suffer from these problems [59–61].

For (i), our design naturally isolates a borrowed function from other user-installed apps since a borrowed function executes as part of XExecutor. XExecutor is a regular Android app and by default, Android OS isolates each app.

Regarding (ii), a malicious app might leak private data (*e.g.*, search history) created by the previous execution of a borrowed function and left in the storage space for XExecutor. To prevent such a scenario, XExecutor discards the class loader that loaded an FPK when XExecutor completes executing a borrowed function. This cleans up the used memory and storage while executing the borrowed function for potential privacy leaks.

For (iii), XExecutor verifies that developer's apps and provider apps acquire required permissions before executing sensitive Android APIs. XExecutor looks up a permission table [62] for each system API and checks whether a provider app has specified the permission on its app manifest and a developer's app has acquired the permission from Android.

USE CASE SCENARIOS

We present four use cases we have implemented using X-Droid. Among the four, we use the first one as the main use case when evaluating X-Droid.

Main Scenario: Password DB Migration

The choice of a function to borrow affects nearly all aspects of X-Droid, especially what a developer implements and how X-Droid executes what is implemented; therefore, we evaluate various aspects of X-Droid with one consistent use case scenario. We carefully select the scenario considering the following: (i) both a developer's app and a provider app have download counts over 100,000; and (ii) both apps are open-sourced, so that for comparison purposes, we can prototype an app feature by borrowing code without X-Droid.

We have selected a use case scenario satisfying both aspects above; password DB migration from a password manager app to another. Google Play market has tens of different password manager apps that store user passwords in a database (DB) file secured by a master password. When a user wants to migrate from one app to another, she has to manually copy-and-paste each stored password, as a custom DB file created by one app is usually unreadable by another app. App developers can help this process by providing a *migration* feature, which we implement for our evaluation using two apps—KeePassDroid [46] and PasswdSafe [63].

KeePassDroid and PasswdSafe are open-source password manager apps but are not compatible with each other. While KeePassDroid uses the external storage of Android to store a password DB file (*i.e.*, other apps can access the file), PasswdSafe cannot properly read it due to KeePassDroid's unique

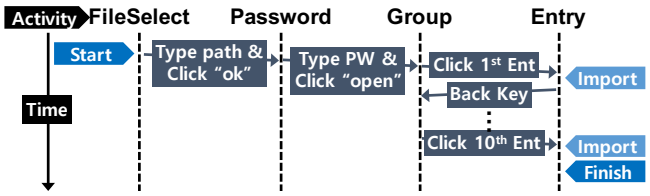


Figure 5: The execution of a XTask for our main scenario.

file format (KDBX). We have implemented a KDBX-format DB-import feature by modifying the PasswdSafe app with X-Droid. With the '*import KDBX*' feature, users can specify the file path of a KDBX file, enter the master password of the KDBX file, and import the file by clicking an 'import' button. When the import button is clicked, our modified PasswdSafe app leverages KeePassDroid via X-Droid to open the KDBX file and migrates the password entries to its own DB file of PasswdSafe. Although KeePassDroid is open-source, we only used its APK file obtained from Google Play.

Figure 5 shows the operation of XTask utilizing KeePassDroid. XTask navigates through four activities to open a KDBX file and reads the entries from it. XTask enters the KDBX file path and its master password in the FileSelect activity and the Password activity, respectively. A user must provide both the file path and the master password, and XTask handles possible errors of incorrect information by catching Toasts from KeePassDroid. Password activity triggers an asynchronous task for decrypting the KDBX file and launches the Group activity that lists password entries in the file when decryption completes. The decrypted information of each password entry is displayed at the Entry activity that can be launched by clicking each entry listed in the Group activity. XTask navigates to one Entry activity by emulating a click of an entry in the Group activity and exports the entry to PasswdSafe's DB by reading the text information such as titles, IDs, passwords, and memos. After an entry is complete, it returns RETURN_TO_PREV_ACTIVITY from the activity handler callback attached to the Entry activity to return to the Group activity and import the next entry.

Use Case Scenarios with Google Play Apps

In addition to KeePassDroid, we implemented three other use cases with apps downloaded from Google Play.

WishList: WishList is an app that helps users create and share their shopping wish lists. In this scenario, a developer wants to prototype a new WishList app feature that notifies users when items in their wish lists appear in local listings. Through prototyping, she can evaluate how useful notifications would be when used items become available. We prototyped the new feature by using WishLocal [64] that shows local used items listings as a provider app. Our prototype periodically reads listings on WishLocal and notifies users when there is a match.

TVGuide: In this scenario, a developer wants to prototype a 'jump-to-my-favorite' feature for a TV guide app, which can automatically switch channels when a user's favorite TV show is on. We prototyped this feature by using Universal TV Remote Control [65], an IR/Wi-Fi TV remote app, as a provider app. Once a user saves the name of her favorite show, our prototype searches the channel number from a TV schedule table and switches channels when the show airs.

RadioAlarm: In this scenario, a developer wants to prototype an alarm app that wakes users up to their favorite radio channel. We prototyped this feature by using RadioDroid [66], an Internet radio streaming app, as a provider app. Our app lets users to select a favorite radio channel and use it as an alarm.

Table 2: Developer study outcome.

ID	Android dev. experience	LoC	Time
P1	Professional (7 yrs)	51	80 mins
P2	Professional (6 yrs)	68*	180 mins
P3	Intermediate	55	90 mins
P4	Beginner	59	150 mins
P5	Beginner	-	-

*P2 could only partially complete the given task.

DEVELOPER STUDY

To evaluate the usability of X-Droid, we conducted an IRB-approved user study with five Android app developers. Two are professional Android app developers and three are CS undergraduate students. They were paid $90 for up to three hours of Android app development. Both professional developers have more than six years of Android app development experience. The first participant (P1) has experience developing a drawing app, a mobile game, and a 3D scanner app. The second participant (P2) has developed a blockchain wallet app and an online payment app.

All three undergraduate student participants major computer science in the same university and have Android app developing experience, while the quality of the developed apps and the depth of Android understanding vary widely. We labeled their development experience by the longest lines of code they have ever contributed for a single Android project—beginner ($<$ 1,000 LoC), intermediate ($<$ 10,000 LoC), and expert ($>$ 10,000 LoC). The third participant (P3) is a junior student who has developed a full research prototype Android app. The fourth participant (P4) is a senior student who has developed an object recognition Android app for a mobile app development course. The fifth participant (P5) is a junior who has displayed client information on a map view and lists of an Android app during an industry internship.

Assuming a scenario where each participant is a developer of PasswdSafe, we asked them to implement a DB import function and provided our X-Droid prototype with X-Droid API documentation. Before the experiment, we explained the basic mechanism of X-Droid and how to use the provided API. We provided them with an example implementation of creating three dummy password entries in PasswdSafe to reduce the time to understand the source code of PasswsSafe. During the experiment, we gave no assistance related to implementation, apart from the instructions given at the beginning.

For each developer, we measured how long one took to complete the task. As Table 2 shows, P1 successfully completed the task within 80 minutes, requiring only 51 lines of code. P2 could import one password entry but could not extend it to

multiple entries in time. This is because there was a task that he wanted to implement, which downcasting (from `XView` to `XviewGroup`) could easily accomplish. However, he did not use that method but spent most of his time figuring out how to accomplish the task (without downcasting). According to the participant, this was due to downcasting being uncommon in industry practices. P3 completed the task within 90 minutes by writing 55 lines of code. P4 who has less Android experience took 150 minutes to complete the task, with 59 lines of code. P5 had difficulties completing the task, as he barely had experience with Android's event-driven programming model. Despite this, we observed that he had no trouble accessing provider app's UI components. He, however, struggled with activity transitions and callback handling.

Regardless, all five developers, including the last, rated their API understanding fairly well. According to our analysis, KeePassDroid is composed of 28,996 lines of Java/C code, and 10,334 lines must be migrated to accomplish the same task without X-Droid. This shows that although prior development experience can influence the outcome, X-Droid API is easily usable, and X-Droid effectively lightens the development burden for the given prototyping task.

Through the observation of developer study participants' app development process, we learned a few lessons on UI-based functionality provisioning and updated X-Droid's design:

Hidden Program Structure underneath UI: For interacting with UI elements, X-Droid API follows Android API's convention that Android developers are already familiar with. For instance, to click an item from a dynamic list view, programmers should call `clickItem(int index)` on the list view as Android API uses `onItemClick(int index)` callback for dynamic list views. This is different from how they would interact with *static* list views where they should directly call `click()` on the item as Android API uses `onClick()` callback for each item. P4 stated that he was confused with a dynamic list with a static list as they are not visually distinguishable. Thus, to assist developers further, we have revised X-Droid APIs so both `clickItem(int index)` and `click()` work for both types of list views.

Gap between Using UI as User and Developer: P3 mentioned that he had difficulty while programming UI actions with event-driven X-Droid API because he is used to making UI actions in sequence. This is an interesting observation but not a serious drawback of X-Droid's event-driven API design as (i) developers can learn how UI interaction translates into code segment using DevTool and (ii) P3 successfully accomplished the given task despite this difficulty.

In our early design of X-Droid, developers should have called `dispatchKeyEvent(int key)` to emulate a 'back' button to go back to previous activity similar to the way that users click a 'back' button. P4 suggested that utilizing the return value of `onActivity()` callback would be more intuitive as both clicking 'back' button and returning from a callback indicate no more UI actions on current activity. We took his suggestion and added `RETURN_TO_PREV_ACTIVITY` as a possible return values of `onActivity()` callback.

Readability of Auto-Generated Code: We observed that UI action code generated by DevTool not only functions as a UI action record but also serves as an example that developers might later refer to. As our early design of DevTool generated code used only `getChildAt()` to navigate to a specific UI element, P4 stated that it was hard to catch the UI elements that the generated code was interacting with, especially when `getChildAt()` was chained very deeply. Therefore, we modified DevTool to properly use (i) `findViewsWithText()` that specifies a UI element with a text on it and (ii) `getDescendantAt()` that is equivalent to calling `getChildAt()` iteratively but is more compact.

Performance Dependency on the Path of UI Action: Through the developer study, we found that developers can take different UI action paths to navigate to the same activity. Although the path of navigation is often deterministic (*i.e.*, it does not change over different executions of a borrowed function) and does not affect the correctness of a borrowed function, it impacts the performance of the borrowed function as some navigation paths are simpler and faster than others.

We thus devise a mechanism to skip the execution of a *deterministic* activity. We define an activity as deterministic when (i) a borrower app always performs the same UI interactions on the activity and (ii) the interactions do not change any state outside XExecutor (*e.g.*, filling out a static input form). Since deterministic activities do not have any side effect, we can remove them safely for optimization as long as we ensure the correct execution of a borrowed function.

Deterministic activities can be identified automatically or manually. For automatic identification, we inspect `OnActivityListener` class that implements all UI actions to be performed on an activity. If the `OnActivityListener` of an activity is automatically generated by DevTool, the activity is deterministic as it is a part of repeating recorded UI actions. For manual identification, a developer can tag an activity by setting an optional 'isDeterministic' flag when she calls `setOnActivityListener()` method for the activity.

Once we identify an activity to be deterministic, we cache the `Intent` issued by it. On Android, activity is an independently executable program block that can be launched by issuing an `Intent`, which is a message containing complete information for launching a new activity. By caching the `Intent` issued by a deterministic activity, we skip the execution of the activity and directly launch the next activity.

MICROBENCHMARK

With our X-Droid prototype, we evaluate whether it delivers its design goals to Android app developers. Our microbenchmark seeks to answer the following questions. First, what is the performance of X-Droid when it executes a borrowed app function? Second, does X-Droid support various off-the-shelf Android devices? Third, does FPK Optimizer effectively reduce the size of an FPK? Fourth, does Intent Caching effectively reduce the execution delay of a borrowed app function?

Experiment Setup: We use the main use case scenario of password DB migration for our evaluation. To run our experiments, we use a Nexus 6 smartphone running Android 6.0.

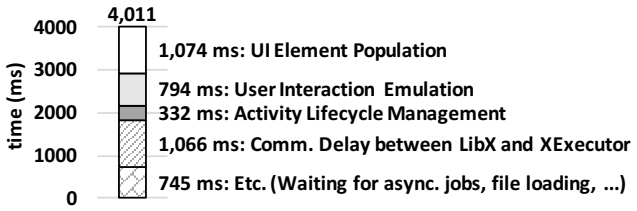


Figure 6: XTask execution delay breakdown.

Table 3: Size of FPKs and their original APKs. (MB)

	KeePassDroid	WishLocal	TVRemote	RadioDroid
APK	3.6	6.9	7.1	4
FPK	2.4 (-33.3%)	2.7 (-60.9%)	5 (-29.6%)	2.5 (-37.5%)

We put ten entries in a KDBX file at the default DB file path of KeePassDroid. We generated KeePassDroid FPK file using our FPK Generator and KeePassDroid APK file downloaded from Google Play.

Run-Time Performance: To evaluate the run-time performance of X-Droid, we measure the KDBX file import delay. KDBX file import took 4,368 ms, out of which 4,011 ms comes from XTask execution. XExecutor initialization took 248 ms, which is a one-time delay incurred when the FPK file is loaded to XExecutor. To understand the source of the XTask execution delay, we profiled XExecutor and Figure 6 shows the result. XTask execution delay is 4,011 ms and the UI element population consumes the largest portion with 1,074 ms. The communication delay between LibX and XExecutor is the next largest with 1,066 ms.

This run-time execution delay is acceptable as password DB migration is just a one-time job. However, there is room for optimization; one possible method is batching IPC calls to reduce the delay between LibX and XExecutor, which we leave as our future work.

Heterogeneous Hardware & OS Versions: To investigate whether borrowed functions through X-Droid can run on off-the-shelf Android devices, we test the *import KDBX* feature on four different Android devices—Nexus 5X running Android 8.1.0, Galaxy J7 running Android 7.0, Nexus 6 running Android 6.0, and Nexus 9 tablet running Android 6.0. We have verified that the *import KDBX* feature backed by X-Droid successfully runs on these devices.

Effectiveness of FPK Optimization: To minimize the storage and distribution overhead of FPK, X-Droid provides FPK Optimizer for developers. We tested FPK Optimizer with our four use case scenarios. As shown in Table 3, the optimizer produces much smaller FPKs than their original APKs.

Impact of Intent Caching: To provide efficient XTask execution, our Intent Caching skips redundant activity listener executions. To evaluate the efficiency of Intent Caching, we implemented a new version of XTask for the *import KDBX* feature. Different from the previous XTask described with the main use case scenario, the new version intentionally reads only one password entry; thus, importing all passwords requires the creation and execution of multiple XTasks, one for each password to import. Although this has an obvious performance drawback of creating and executing multiple XTasks, it might be a preferred implementation option for its simplicity. Figure 7 depicts this implementation.

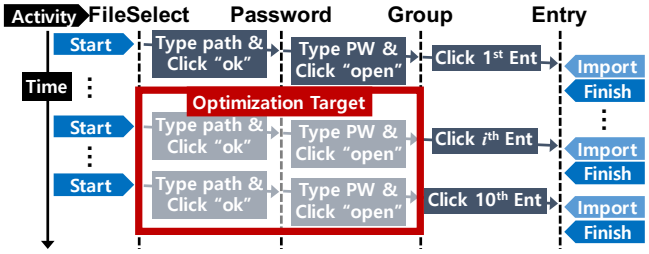


Figure 7: The execution sequence of a XTask for intent caching microbenchmark.

To evaluate how our Intent Caching improves the performance of this new version, we tag activity listeners on the FileSelect activity and the Password activity as *deterministic*, as described UI actions are repeated across different XTasks and do not vary.

The execution delay of this new *import KDBX* implementation is 11.29 seconds, which is 2.58 times slower than the default implementation without our optimization (4.37 seconds), due to redundant UI actions. When we enable Intent Caching, the execution delay is reduced to 5.39 seconds as it effectively skips redundant UI actions.

DISCUSSION

Limitations: Although X-Droid optimizes its UI emulation (*e.g.*, skipping animations), UI emulation overhead induces considerable delay on borrowed function execution as shown in the Microbenchmark section. X-Droid is still useful for many prototyping tasks where development efficiency is more important than performance; however, the use of X-Droid might be limited when prototypes require responsiveness.

Also, the current version of X-Droid does not support reading images, sounds, or videos from UI elements or writing data on microphone, camera, or sensor inputs of provider apps. This does not entirely prevent X-Droid from borrowing app functions that require multimedia input (e.g., face recognition from a specified image file) but limits the coverage. X-Droid also does not support customized UI: e.g., mobile game UI managed by custom game engine.

Use of External Resources: During a borrowed function execution, the borrowed function might use external resources (*e.g.*, a remote server) of a provider app. Since the owner of the provider app cannot distinguish X-Droid from regular users, external resources could be abused when a prototype executes a borrowed function too frequently. X-Droid could employ a rate-limit strategy to reduce the impact of such misuse.

Legality: Based on our understanding and legal consultation, we believe that the use of X-Droid would not violate any legal issue under the "fair use" doctrine if the following three

Table 4: Motivation for sharing functions and code.

Reasons	Q1	Q2
Sharing knowledge and skills	82.8%	70.6%
Improving products of other developers	55.2%	64.7%
Learning and developing new skills	69%	56.9%
Improving my job opportunities	46.6%	37.3%
Belief that software should be free, not proprietary	36.2%	23.5%
Distributing not marketable software products	8.6%	9.8%
Concern about large software companies' influence	5.2%	3.9%

conditions are satisfied (here, we assume that a developer *D* uses X-Droid to incorporate an existing app *A*'s feature).

First, app *A*'s feature is not protected by patent laws. Second, app *A*'s terms of use does not explicitly prohibit reverse engineering. Third, assuming app *A* is distributed through Google Play, developer *D* uses X-Droid for her own educational purpose to learn about app *A*'s feature.

To fully incorporate the feature for commercial purposes, developer *D* completely disregards the prototype and implements the feature from scratch. The cleanest way for ensuring this would be taking *the cleanroom development strategy*; a different developer (developer *E*) develops the feature without copying anything from the prototype or developer *D*. We emphasize however that our discussion here is not to be taken as legal advice in any way.

X-Droid for Apps to Release: If permitted by original developers, X-Droid could be used to develop a production app, not just a prototype. Among the issues in releasing apps developed through X-Droid (e.g., how developers can control any side effect of another app), one issue could be whether the developers of provider apps would donate their app functions for others. To find out if developers are willing to share their apps' functions, we sent online surveys to the developers of popular open-source third-party libraries for Android [67] and received 58 replies.

We asked those developers *Q1*: what is the motivation for developing third party libraries. We also asked them if they would share the functionality of their apps with others if they owned an app, and 51 of 58 answered that they would. To these 51 developers, we asked *Q2*: why would they share their app functionality. We provided options based on a prior survey of open-source developers [68] and allowed the participants to select multiple options.

Table 4 shows the statistics of their responses. It shows that motivations for developing open-source software are mostly to share and learn knowledge and skills, improve others' products, and improve job opportunities. All these goals could also be achieved by sharing app functions. This similarity is reflected on the answers to the second question, as the statistics for the two questions show high similarity.

In summary, we found that (1) the majority of open-source community members contribute to the community for altruistic inner motivations, and (2) their motivations can possibly be translated into their donation of app functions.

CONCLUSIONS

We presented X-Droid that empowers developers to quickly produce functional prototypes of an Android app. X-Droid repurposes the UI of an Android app as an interface where developers define functions to import, and provides LibX and DevTool to further ease the development effort. X-Droid does not require the developers to understand the function implementation details or have access to source code for other Android apps where imported functions are implemented. We evaluated X-Droid with microbenchmarks and showed that X-Droid has acceptable performance and supports various Android devices and versions. We conducted a usability study with five Android app developers and they indicated that X-Droid provides an API that is usable and easy to understand. With X-Droid, a developer has imported an another app's function that requires 10,334 lines of Java code by writing only 51 lines of Java code. We believe X-Droid makes a large step toward new mobile app prototyping where developers can quickly and easilty produce functional prototypes.

ACKNOWLEDGMENTS

We thank Chunjong Park for his early contribution. This research was supported in part by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2017M3C4A7083534), the International Cooperative R&D program funded by the Ministry of Trade, Industry and Energy(MOTIE) and Korea Institute for Advancement of Technology(KIAT) (N0002099), and the National Science Foundation, CNS-1350883 (CAREER) and CNS-1618531.

REFERENCES

[1] Steven D Tripp and Barbara Bichelmeyer. Rapid prototyping: An alternative instructional design strategy. Educational Technology Research and Development, 38(1):31–44, 1990.

[2] Joel Brandt, Philip J Guo, Joel Lewenstein, Mira Dontcheva, and Scott R Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1589–1598. ACM, 2009.

[3] Xiong Zhang and Philip J Guo. Fusion: Opportunistic web prototyping with ui mashups. In The 31st Annual ACM Symposium on User Interface Software and Technology, pages 951–962. ACM, 2018.

[4] Ingbert R Floyd, M Cameron Jones, Dinesh Rathi, and Michael B Twidale. Web mash-ups and patchwork prototyping: User-driven technological innovation with web 2.0 and open source software. In 2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07), pages 86–86. IEEE, 2007.

[5] M Cameron Jones, Ingbert R Floyd, and Michael B Twidale. Patchwork prototyping with open source software. In Software Applications: Concepts, Methodologies, Tools, and Applications, pages 1641–1656. IGI Global, 2009.

[6] Balsamiq. rapid, effective and fun wireframing software. https://balsamiq.com/, 2019.

[7] Prototype faster, smarter and easier with mockplus! https://www.mockplus.com/?r=trista, 2019.

[8] Wireframe.cc - minimal wireframing tool. https://wireframe.cc/, 2019.

[9] Marvel app. https://marvelapp.com/, 2019.

[10] Invision | digital product design, workflow & collaboration. https://www.invisionapp.com/, 2019.

[11] Uxpin | ui design and prototyping tool. https://www.uxpin.com/, 2019.

[12] Allen Cypher and Daniel Conrad Halbert. Watch what I do: programming by demonstration. MIT press, 1993.

[13] Toby Jia-Jun Li, Amos Azaria, and Brad A Myers. Sugilite: creating multimodal smartphone automation by demonstration. In Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, pages 6038–6049. ACM, 2017.

[14] Tanzirul Azim, Oriana Riva, and Suman Nath. ulink: Enabling user-defined deep linking to app content. In Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys), pages 305–318. ACM, 2016.

[15] Leshed, Gilly and Haber, Eben M and Matthews, Tara and Lau, Tessa. CoScripter: automating & sharing how-to knowledge in the enterprise. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pages 1719–1728. ACM, 2008.

[16] Nichols, Jeffrey and Lau, Tessa. Mobilization by demonstration: using traces to re-author existing web sites. In Proceedings of the 13th international conference on Intelligent user interfaces, pages 149–158. ACM, 2008.

[17] Fujima, Jun and Lunzer, Aran and Hornbæk, Kasper and Tanaka, Yuzuru. Clip, connect, clone: combining application elements to build custom interfaces for information access. In Proceedings of the 17th annual ACM symposium on User interface software and technology, pages 175–184. ACM, 2004.

[18] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M Memon. Using gui ripping for automated testing of android applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 258–261. ACM, 2012.

[19] Young-Min Baek and Doo-Hwan Bae. Automated model-based android gui testing using multi-level gui comparison criteria. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 238–249. ACM, 2016.

[20] Shuai Hao, Bin Liu, Suman Nath, William GJ Halfond, and Ramesh Govindan. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In Proceedings of the 12th annual international conference on Mobile Systems, Applications, and Services (MobiSys), pages 204–217. ACM, 2014.

[21] Chieh-Jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, et al. Caiipa: Automated large-scale mobile app testing through contextual fuzzing. In Proceedings of the 20th annual international conference on Mobile computing and networking (MobiCom), pages 519–530. ACM, 2014.

[22] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: An input generation system for android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE), pages 224–234. ACM, 2013.

[23] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. Automatic and scalable fault detection for mobile applications. In Proceedings of the 12th annual international conference on Mobile Systems, Applications, and Services (MobiSys), pages 190–203. ACM, 2014.

[24] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based gui testing of android apps. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE), pages 245–256. ACM, 2017.

[25] Selenium - web browser automation. https://www.seleniumhq.org/, 2019.

[26] Helium. https://heliumhq.com/, 2019.

[27] Scrapy. https://scrapy.org/, 2019.

[28] Stuerzlinger, Wolfgang and Chapuis, Olivier and Phillips, Dusty and Roussel, Nicolas. User interface façades: towards fully adaptable user interfaces. In Proceedings of the 19th annual ACM symposium on User interface software and technology, pages 309–318. ACM, 2006.

[29] Nichols, Jeffrey and Hua, Zhigang and Barton, John. Highlight: a system for creating and deploying mobile web applications. In Proceedings of the 21st annual ACM symposium on User interface software and technology, pages 249–258. ACM, 2008.

[30] Rob Ennals and David Gay. User-friendly functional programming for web mashups. In ACM SIGPLAN Notices, volume 42, pages 223–234. ACM, 2007.

[31] James Lin, Jeffrey Wong, Jeffrey Nichols, Allen Cypher, and Tessa A Lau. End-user programming of mashups with vegemite. In Proceedings of the 14th international conference on Intelligent user interfaces, pages 97–106. ACM, 2009.

[32] Jeffrey Wong and Jason I Hong. Making mashups with marmite: towards end-user programming for the web. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 1435–1444. ACM, 2007.

[33] Pokress, Shaileen Crawford and Veiga, José Juan Dominguez. MIT App Inventor: Enabling personal mobile computing. Proceedings of PRoMoTo 2013, 2013.

[34] Thunkable.com: Drag and Drop Mobile App Builder for iOS and Android. `https://thunkable.com/`, 2019.

[35] Parallel space. `http://parallel-app.com/`, 2019.

[36] Parallel accounts. `https://play.google.com/store/apps/details?id=com.in.parallel.accounts`, 2019.

[37] Go multiple - parallel account. `https://play.google.com/store/apps/details?id=com.jiubang.commerce.gomultiple`, 2019.

[38] Do multiple - unlimited parallel account. `https://play.google.com/store/apps/details?id=com.polestar.domultiple`, 2019.

[39] Dr. clone: Parallel accounts, dual app, 2nd account. `https://play.google.com/store/apps/details?id=com.trendmicro.tmas`, 2019.

[40] Dynamicapk. `https://github.com/CtripMobile/DynamicAPK`, 2019.

[41] Dl: dynamic load framework for android. `https://github.com/singwhatiwanna/dynamic-load-apk`, 2019.

[42] Android dynamic loader. `https://github.com/mmin18/AndroidDynamicLoader`, 2019.

[43] Android pluginmanager. `https://github.com/houkx/android-pluginmgr`, 2019.

[44] Droid plugin. `https://github.com/Qihoo360/DroidPlugin`, 2019.

[45] Understand the activity lifecycle. `https://developer.android.com/guide/components/activities/activity-lifecycle`, 2019.

[46] Brian Pellin. Keepassdroid. `http://www.keepassdroid.com/`, 2019.

[47] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. Reran: Timing-and touch-sensitive record and replay for android. In Software Engineering (ICSE), 2013 35th International Conference on, pages 72–81. IEEE, 2013.

[48] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. Mobiplay: A remote execution based record-and-replay tool for mobile applications. In Proceedings of the 38th International Conference on Software Engineering (ICSE), pages 571–582. ACM, 2016.

[49] Matthew Halpern, Yuhao Zhu, Ramesh Peri, and Vijay Janapa Reddi. Mosaic: cross-platform user-interaction record and replay for the fragmented android ecosystem. In Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on, pages 215–224. IEEE, 2015.

[50] Monkey runner. `https://developer.android.com/studio/test/monkeyrunner/index.html`, 2019.

[51] Proguard, the open source optimizer for java bytecode. `https://www.guardsquare.com/en/products/proguard`, 2019.

[52] Sharath Chandrashekhara, Taeyeon Ki, Kyungho Jeon, Karthik Dantu, and Steven Y Ko. Bluemountain: An architecture for customized data management on mobile systems. In Proceedings of the 23rd Annual International Conference on Mobile Computing and Networking (MobiCom), pages 396–408. ACM, 2017.

[53] Taeyeon Ki, Alexander Simeonov, Bhavika Pravin Jain, Chang Min Park, Keshav Sharma, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. Reptor: Enabling api virtualization on android for platform openness. In Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys), pages 399–412. ACM, 2017.

[54] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO), page 75. IEEE Computer Society, 2004.

[55] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In OSDI, volume 12, pages 107–120, 2012.

[56] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS), 32(2):5, 2014.

[57] Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Instrumenting android and java applications as easy as abc. In International Conference on Runtime Verification (RV), pages 364–381. Springer, 2013.

[58] Haichen Shen, Aruna Balasubramanian, Anthony LaMarca, and David Wetherall. Enhancing mobile apps to use sensor hubs without programmer effort. In Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing (UbiComp), pages 227–238. ACM, 2015.

[59] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A taxonomy of privilege escalation attacks in android applications. International Journal of Security and Networks, 9(1):40–55, 2014.

[60] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In international conference on Information security, pages 346–360. Springer, 2010.

[61] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In Workshop on Mobile Security Technologies (MoST), volume 10, 2012.

[62] Michael Backes, Sven Bugiel, Erik Derr, Patrick D McDaniel, Damien Octeau, and Sebastian Weisgerber. On demystifying the android application framework: Re-visiting android permission specification analysis. In USENIX Security Symposium, pages 1101–1118, 2016.

[63] Password safe. `https://pwsafe.org/`, 2019.

[64] Wish local. `https://wish-local-buy-sell.en.softonic.com/android`, 2019.

[65] Universal tv remote control. `https://play.google.com/store/apps/details?id=codematics.universal.tv.remote.control`, 2019.

[66] Radiodroid 2. `https://play.google.com/store/apps/details?id=net.programmierecke.radiodroid2`, 2019.

[67] CodePath. Must have libraries. `https://github.com/codepath/android_guides/wiki/Must-Have-Libraries`, 2019.

[68] Rishab A Ghosh, Ruediger Glott, Bernhard Krieger, and Gregorio Robles. Free/libre and open source software: Survey and study, 2002.