User scripting on Android using BladeDroid

Ravi Bhoraskar[☺], Dominic Langenegger[†], Pingyang He[☺], Raymond Cheng[☺], Will Scott[☺], and Michael D. Ernst[☺]
[☺]University of Washington {bhora, pingyh, ryscheng,wrs,mernst}@cs.washington.edu
[†]ETH Zurich, Switzerland dominicl@ethz.ch

Abstract

Compared to desktop and web applications, mobile applications have so far been developed in an extremely siloed environment. The apps running on our phone are developed by a single entity with operating system protections between sharing of data or code between programs. However, application extensibility is often desired. In the web, a secondary ecosystem flourishes around browser extensions, enabling users to customize the web as they wish. This paper presents BladeDroid, a system enabling user customization of mobile applications, using a novel combination of bytecode rewriting and dynamic class loading. We describe four extensions that we have built to evaluate BladeDroid's usability, robustness, and performance.

1 Introduction

In all mobile operating systems today, apps are large binary silos that users can and must install in an all-or-nothing proposition. However, users may want to customize the behavior of the app beyond what the developer provides. For example, a user may want to rearrange the user interface to fit their usage patterns. In the current mobile ecosystem, it is difficult to even write a simple ad blocker without root access.

In this paper, we introduce BladeDroid, a system that supports custom user scripts, which we call "Blades", on unmodified Android phones. While such extensibility has been supported and built into web browsers for decades, a number of unique challenges exist in the mobile space.

An extension system must enable custom user scripts without the explicit support of the host operating system. Requiring modification of the operating system would dramatically limit deployability. Likewise, the system must adhere to the existing mobile operating system security model. Unlike web apps where the DOM and client-side JavaScript is fully accessible to other scripts, mobile apps are opaque packaged binaries. An extension system then needs to not only inject code into app store packages without access to source code, but also provide hooks enabling meaningful functionality.

By enabling custom user scripts in mobile applications, BladeDroid opens a world of new customization possibilities on smartphones. BladeDroid can be used for debugging and fuzz testing, automating repetitive tasks, hiding promotional content, changing visual layout, or even modifying the default behavior of existing apps. Like with web extensions, one would expect some Blades to be app specific, and others to generalize across several apps. We have written four example Blades that will serve as running examples for the rest of the paper:

Ad Blocker: An ad blocker enforces a blacklist of advertising networks, removing relevant UI elements across all applications.

Social Media Plugin: A "Like" or "Share" button is inserted into every application to share the current app context to a social media platform. For example, a user can "like" level 10 of Angry Birds during gameplay and post it to their social news feed.

Quiz Cheater: A custom script highlights the correct answer in a specific quiz app, helping the user to win the game.

Record and Replay: This script creates a visual overlay over the UI and records user interactions. The action log can then be replayed with the same timings to reproduce previous interactions. Record and replay is a valuable debugging tool [9].

Users interact with BladeDroid by installing a BladeManager app from the Android app store, like any other application. At run time, users can choose to inject Blades into other applications from the BladeManager app. BladeDroid uses a novel combination of bytecode rewriting and dynamic class loading to instrument existing applications on the device to accept dynamic Bladeloading. Because a Blade runs in the same sandboxed environment as the app it is injected into, Blades are compatible with the existing Android security model. In fact, the BladeManager app itself requires no more permissions than the Google Play Store. As Blades are developed, we expect the BladeManager app to facilitate discovery and sharing. We believe BladeDroid can be released on existing app stores without modifying the underlying Android OS and security model.

In the rest of the paper, we will elaborate on the following contributions:

- We show how to enable custom user scripts on an unmodified Android phone, using a novel combination of bytecode-rewriting and dynamic class loading. (Sec-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
APSys '14, June 25-26, 2014, Beijing, China
Copyright 2014 ACM 978-1-4503-3024-4/14/06 ...$15.00.

tion 3)

- We implemented BladeManager, a user-level application that hooks into unmodified applications on the phone and injects custom user-scripts (Blades) at runtime. (Section 3)
- We demonstrate several Blades — an ad blocker, social plugin, quiz cheater, and debugger tool — to show the range of app extensions that are enabled by our system. Preliminary results show low space and performance overheads. (Section 4)

2 Background

Android is an open source mobile operating system (OS). Android applications are written in Java and are compiled to Dalvik bytecode. They are distributed as packages with the bytecode and resource files. The Android OS has a Dalvik virtual machine, which loads and runs the application using a just-in-time compiler [3].

2.1 Android App Structure

Each Android app is organized as a set of screens that users can navigate between, called `Activities` [1]. Each `Activity` usually takes up the entire screen and consists of UI elements, known as `Views`, such as textboxes, buttons, lists, and images. A View can be associated with a callback function that is invoked when a user interacts with the View. The developer defines these callbacks in the Activity's Java class. A new Activity is instantiated using a runtime mechanism called an Intent.

2.2 Android Security

Android provides a manifest-based permission system, where applications must explicitly declare permissions for the resources they use. Examples of such permissions include the `INTERNET` permission to access data on the Internet or the `READ_EXTERNAL_STORAGE` permission to access files on the external storage of the device (typically an SD card).

Each Android application runs in a sandboxed environment with an isolated file system, and can only communicate with other apps using an interprocess communication mechanism.

3 Design and Implementation

BladeDroid was written with the following high-level design goals:

Powerful for developers: The Blade API should be useful across arbitrary Android apps, and should not require the Blade developer to understand app-specific code unless the Blade specifically requires it. For example, the same ad-blocker Blade should be able to run on any app that displays an ad. In particular, the Blade API must expose the UI and events of a host app to the Blade such that it is easy to write concise Blades.

```
1  public interface Blade {
     void onCreate(Activity activity,
      Bundle savedInstanceState);
3    void onStart(Activity activity);
     void onResume(Activity activity);
5    void onPause(Activity activity);
     void onStop(Activity activity);
7    void onDestroy(Activity activity);
     boolean onKeyDown(Activity activity
      , int keyCode, KeyEvent event);
9    boolean onKeyLongPress(Activity
      activity, int keyCode, KeyEvent
      event);
     boolean onKeyUp(Activity activity,
      int keyCode, KeyEvent event);
11 }
```

Listing 1: The Blade Interface, with the methods implemented by all user scripts in order to extend the functionality of existing Android Activities.

Easy for users: It must be easy for a non-technical user to dynamically load and run Blades at runtime. BladeDroid needs to fit the user's mental model of installing and running apps, without understanding the technical details of how apps are instrumented to support Blades.

Secure: A Blade should run in the same sandbox as the host application it is injected into. As such from a security perspective, the Blade cannot do more than the application can. Furthermore, it must be impossible to inject Blades without the explicit consent of the user.

Low Overhead: BladeDroid should impose little to no overhead to the performance of existing applications when no Blades are present. When Blades are injected into an app, the act of loading a blade must have minimal effect on the responsiveness and performance of the application.

3.1 The Blade API

Because Android does not inherently expose all UI elements and events in a structured DOM like the web does, we need to create an API against which Blades can be written. Listing 1 shows the `Blade` interface against which all Blades are written. The Blade class definition may contain a Java annotation `BladeScope` defining a filter for Activities that it should run on using a regular expression that is matched to the full Activity name. For example, an ad-blocker may run on all activities, whereas a game guide may be tailored to a specific activity.

The Blade API design closely mirrors the Android app lifecycle itself. The Blade can register callbacks that are fired when the app is created, destroyed, started, stopped, paused, and resumed. The Blade is also given hooks into physical button events (e.g. Home, Back, Volume). Blades are passed a reference to the current Activity when these callbacks are fired. Blades also have access to all of

the Android APIs that the host application has access to, which allows it to programmatically modify the Activity's view and layouts.

3.2 Blade Ecosystem

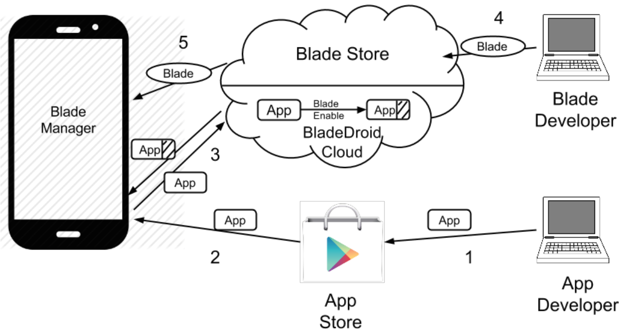


Figure 1: The BladeDroid Ecosystem. (1) App developer submits apps to App Store. (2) User downloads app from app store. (3) Blade Manager uploads app to BladeDroid cloud and reinstalls the Blade-enabled app. (4) Blade Developer submits Blade to Blade-store (5) Blade Manager fetches and manages Blades.

Figure 1 shows how we envision BladeDroid would extend the current application distribution. Users continue to install applications from an app store (like Google Play). In order to use BladeDroid, the user must install a BladeManager app. With BladeManager, a user can choose to instrument any existing installed application to be Blade-enabled, which is necessary before any Blades can be injected into that app. In this process, the installed application package (APK) is sent to a BladeDroid cloud, which performs the bytecode-rewriting and recompiles the application with a BladeLoader and BladeExecutor. The device then reinstalls the Blade-enabled application.

The BladeManager app a provides a Blade installation interface, allowing the user to download, install, and uninstall Blades from an online BladeStore. The BladeManager provides a nice user interface to let the user choose which of the Blades to run on specific Blade-enabled apps.

The BladeManager internally organizes the Blades on the external storage of the Android device, which acts as a shared filesystem across apps.

3.3 Blade Injection

Because a mobile app is not directly programmable in bytecode form, it needs to be instrumented so that Blades written using the Blade API may be able to execute within the context of an app. We used *bytecode rewriting* [10, 11] techniques to add a BladeLoader and a BladeExecutor to an existing application.

BladeLoader: The BladeLoader executes when the application first starts and contacts the BladeManager app using the Android interprocess communication mechanism (Binder). The BladeManager responds with a list of Blades to load, as well as the location of the Blades on

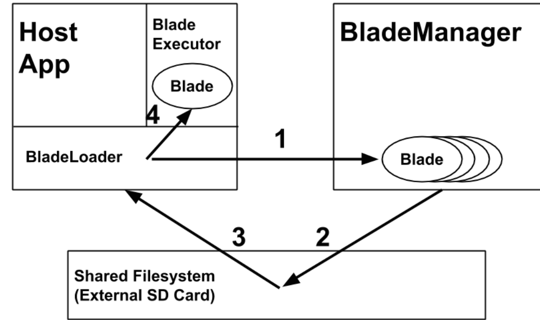


Figure 2: The workflow of installing and executing Blades. (1)The BladeLoader sends a request to the BladeManager. (2) BladeManager responds with the location of Blade on the shared filesystem. (3) BladeLoader verifies the Blade and loads it in memory (4)The BladeExecutor executes the Blade on the appropriate events at runtime.

external storage. Because all Blades must be signed by the BladeStore, the BladeLoader can verify the signature of each Blade. BladeLoader then uses Android's *Dynamic Class Loading* to load each Blade into memory. Figure 2 shows the relationship between the BladeLoader and the BladeManager.

BladeExecutor: The BladeExecutor forwards all Activity lifecycle and key press events, triggering the respective callback in the Blade when these events fire.

Our key observation is that these two techniques — Bytecode Rewriting and Dynamic Class Loading — when used together, create a powerful mechanism that enables user-side programmability on traditionally unprogrammable apps.

3.4 BladeDroid Cloud

The BladeDroid Cloud uses the Soot framework [12] to perform bytecode rewriting, which adds the BladeLoader and BladeExecutor to a given app. Using Soot, we convert an Android application into Soot's intermediate representation, Jimple, which is designed to ease analysis and manipulation. BladeDroid uses Soot's class visitor methods to iterate through all classes in the application, and determine the Activities. For all of the methods corresponding to the ones in the Blade interface (creating them if necessary), BladeDroid inserts code to call the corresponding Blade methods. To read the Blades from external storage, BladeDroid requires the `READ_EXTERNAL_STORAGE` permission which it adds to the application during rewriting.

In the current implementation, we manually blade-enable applications and sideload them back onto the device. As future work, future versions of the BladeManager should automatically blade-enable applications on installation.

3.5 Security

Since Blades run within an existing application context, they naturally run with the same permissions as their host application, and cannot gain additional privileges

beyond what already has been received from the system (except External Storage read permission, which is added to enable Blade-loading). Similarly, the Blade is restricted to the same sandbox.

By signing Blades, the system prevents the Blade-Loader from arbitrary code. Furthermore because the BladeLoader checks with the BladeManager app when an app is first loaded, it can verify which Blades the user has granted permission to run in the application. Note that the BladeDroid cloud and BladeStore can easily merge with an existing app store, such that the application is simply recompiled to be Blade-enabled at install time.

4 Evaluation

This section describes our experience with applying Blade-Droid to multiple mobile applications from the Google Play Store including detailed evaluations on performance and usability. Additionally, we identify the strengths and limitations of the proposed model by evaluating what types of user scripts are possible to implement and which are not.

4.1 User Survey

We conducted a user survey to determine the value of BladeDroid as perceived by users. The survey was distributed on the Android subreddit on Reddit[1], where it got 13 responses. We ask the respondents if they would find user-scripting useful on mobile apps, and if so, what some of those Blades would be. Of the 13 participants, 10 knew of multiple mobile applications that they would like to customize. We list the possible Blades suggested by the survey participants. Each of these can be built within the BladeDroid framework.

Ad Blocker: An ad blocker was the most named extension in the survey. We explain our implementation of it in Section 4.2.1.

Game cheating: Either helps the user by e.g. showing her the right answers in a quiz game or directly gives her high scores. We explain our implementation of one such Blade in Section 4.2.3.

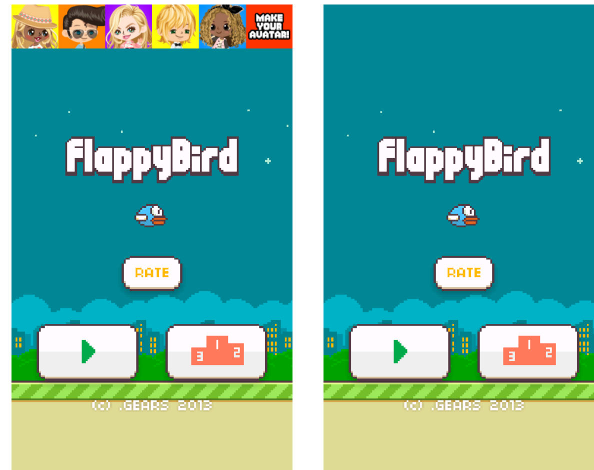
Automate repetitive tasks: Simplify tasks that are done very often, for example automatically poke-back a friend on the Facebook app.

Hide promotional content: Hide promotional tweets or posts in the Twitter or Facebook app respectively. The implementation of this Blade would be very similar to the Ad Blocker.

App design: Change the user interface by adapting color, element dimensions, or layout.

Launch screen change: Starts the app with a different Activity. The survey participant suggested applying this

[1]http://reddit.com/r/android



(a) Without Ad Blocker (b) With Ad Blocker

Figure 3: Flappy Bird with and without Ad Blocker. Notice that the ad on top of the screen has been removed in the second screenshot.

to the OneBusAway application[2] in order to have it start directly in the bookmarks page.

Although the data set may not be representative, we conclude that user scripting on mobile platforms is indeed something that users would use, if it's simple enough to do so. The user study also showed us some applications for BladeDroid we had not thought of and allowed us to analyze their feasibility.

4.2 Sample Blades

We wrote four Blades for four different applications, and measure the ease and efficacy of using the Blade interface (Section 4.3). The four Blades are described here. We then run them in different apps which are rewritten by BladeDroid, and measure the performance overheads of running them (Section 4.4).

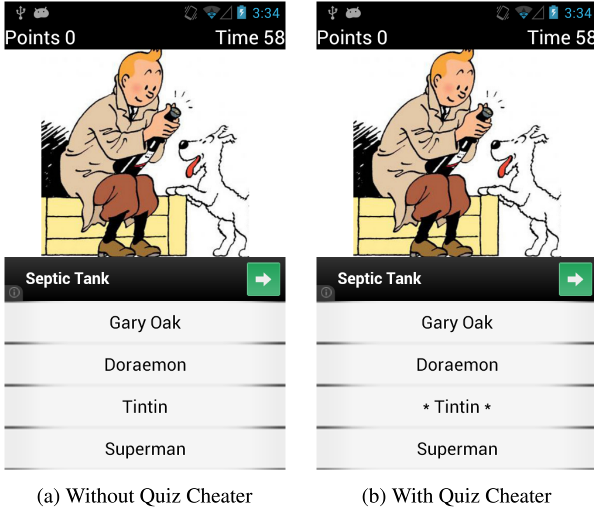
4.2.1 Ad Blocker

Ad libraries typically provide a custom UI control (View) that app developers include in their app. This View then serves ads at runtime. The Ad Blocker Blade contains a list of known advertisement Views. Once an Activity is created, the Ad Blocker will recursively find all AdView(s) in this Activity, and render them invisible. Figure 3 demonstrates an example of running Ad Blocker.

4.2.2 Social Media plugin

Our social media plugin (called *Socialify*) adds a virtual like button to every page of the app it is installed in. Hitting the like button posts an update containing the name of the app and contents of the page to the user's Facebook timeline. The Socialify Blade overrides the long press event of the back button, which is then treated as the Like button. Pressing this button gathers the app

[2]http://onebusaway.org



(a) Without Quiz Cheater (b) With Quiz Cheater

Figure 4: Cartoon Quiz with and without Quiz Cheater. Notice that the correct answer is highlighted with surrounding asterisks in the second screenshot.

and activity names, and creates an Intent with these as the parameters. It then calls the Socialify app, which is also installed on the phone, with this intent. The app in turn posts the content to the user's Facebook timeline.

4.2.3 *Quiz Cheater*

Quiz games like DuelClash, QuizUp or Cartoon Quiz [5] give the user multiple possible answers to a question one of which is correct. As a proof of concept implementation of a game-cheater, we developed an app specific Blade for the single player game Cartoon Quiz. The Blade implementation uses the Java Reflection API to access fields on the Activity that are known to contain the answers to the questions displayed. It then modifies the layout of the Activity to highlight the right answer, as shown in Figure 4

4.2.4 *Record and Replay*

Record and Replay allows one to record UI-interactions with an app, and replay them later. It is a valuable technique for debugging, as it allows reproduction of bugs. It may also prove useful for users to navigate through a complicated series of pages, for example to set up low light settings in a Camera app or automate other repetitive tasks (as mentioned in the user survey). Previous work has enabled record and replay by modifying the operating system, to interpose on the I/O event stream [9]. BladeDroid allows to do this without OS support. The record part of the R&R-Blade adds a listener to the touch event for each View in the layout and logs every interaction with timing information. The replay part reads this log, and performs the same touch events on the corresponding Views, with the same timing delays.

Blade	LOC	Comment
AdBlocker	52	(see section 4.2.1)
Social Media Plugin	30+90	Additional app required (see section 4.2.2)
QuizCheater	79	(see section 4.2.3)
Record & Replay	292	(see section 4.2.4)
Toast Blade	15	Shows a Toast message
Log Blade	14	Logs a simple message

Table 1: Lines of code (LOC) for implemented example Blades.

4.3 Ease of writing Blades

One of the primary design goals of the Blade API was that writing Blades should be quick and easy, like writing scripts for web apps. We list the entirety of the Ad Blocker blade in the Appendix. For evaluation, we use the number of lines of code as a proxy for ease of writing a Blade. For each of the four Blades described earlier, as well as two other “Hello World” Blades, we report the lines of code in the Blade class in Table 1. The median LOC is around 40, and even moderately complex Blades like the Quiz Cheater and Ad Blocker have under a hundred lines of code total, thus providing evidence for the ease of use of the Blade API.

4.4 Performance and Overhead Metrics

This sections shows measurements to evaluate the overheads of using BladeDroid and discusses its performance.

For size overhead, we measure the change in size of an Android package after Blade-enabling it. We use as our dataset 176 randomly chosen apps from a set of 15749 apps crawled from the Google Play store in December 2013. Figure 5 shows the cumulative distribution function (CDF) of the fraction increase in the size of the APK files due to Blade-enabling. All the apps have less than 12% increase in package size, with the 90th percentile having an increase of 3.6%. We consider this an acceptable overhead in storage.

Next, we measure the latency caused by the time taken to load Blades into an app. To do this, we crated a modified version of the BladeLoader that logs the timing information for loading Blades. This includes the time for extracting the Blade class from the JAR, and loading it into the Dalvik Virtual Machine. We note that this delay happens only when the app is started, and not on Activity transitions, since Blades can reside in the app memory across activities.

Figure 6 plots a graph of the load time with an increasing number of Blades. We notice a linearly increasing trend with a rising number of Blades, as expected. Note, however, that even with 20 Blades, the load time is still only 120 ms. An active Mozilla study [7] suggests that the average load time for Android apps is of the order of a few seconds. With an addition of the order of tens of milliseconds, BladeDroid does not cause significant

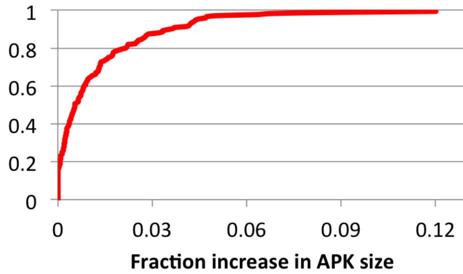


Figure 5: CDF of the fraction increase in APK size, of 176 Blade-enabled apps. We see that 90% of all apps have a size increase of less than 4%.

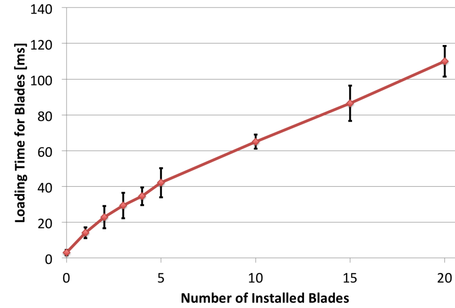


Figure 6: Plot of Blade loading time [ms] with varying number of installed Blades, averaged over 10 runs. This process happens on very application start.

overhead. The authors did not perceive an unusual lag in a Blade-enabled app, even with 20 Blades loaded.

We note that the Blades themselves contain arbitrary code, which runs within the context of the apps, and thus different Blades themselves could have could have an impact on latency, performance, and energy consumption of apps.

5 Related Work

BladeDroid applies many ideas from web and desktop extension frameworks to the mobile context. We also draw on a set of instrumentation techniques in the mobile space to transparently extend applications.

5.1 Web Application User Scripts

All major browsers provide extension APIs [2, 4]. These APIs allow code to interact with websites programmatically. GreaseMonkey [6] for Firefox popularized this concept, allowing users to inject JavaScript on pages and share their developed scripts. AdBlock Plus[3] has become one of the most popular browser extensions, preventing websites from displaying advertisements. Unlike these systems, BladeDroid extends applications without platform support or a high-level semantic language.

5.2 Mobile Platform Instrumentation

Mobile instrumentation frameworks exist, rewriting applications to provide additional facilities. However, these systems all require changing applications at compile-time. SIF, the Selective Instrumentation Framework [10], performs instrumentation based analyses built from a short description in a custom Java-like language. This description is set at compile time, allowing for more dramatic changes to application logic than Blades. I-ARM-DROID [8] allows users to specify a security policy for Android API methods, and then implements this policy through rewriting. BladeDroid is the first system that we are aware of that allows users to extend applications without OS support or reinstallation.

[3] https://adblockplus.org

6 Conclusion

We have proposed BladeDroid, a system which enables user-side programmability for traditionally unprogrammable mobile apps, using *Blades*. We proposed an API against which these Blades can be written, and a novel combination of bytecode rewriting and dynamic class loading that implements this interface on pre-existing apps in the Play Store. We developed a BladeManager app allowing users to easily install and remove Blades. Preliminary results showed that the overhead of BladeDroid is minimal, and is vastly outweighed by the possibilities of writing arbitrary mobile app extensions. With BladeDroid, we have looked forwards to a rich ecosystem of Android extensions.

7 References

[1] Android Activity. http://developer.android.com/reference/android/app/Activity.html.
[2] Chrome Extensions. http://developer.chrome.com/extensions/index.html.
[3] Dalvik. http://source.android.com/devices/tech/dalvik/.
[4] Firefox Extensions. https://developer.mozilla.org/Add-ons.
[5] Google Play Store. https://play.google.com/.
[6] GreaseMonkey. http://www.greasespot.net.
[7] Mozilla Eideticker Project. http://eideticker.mozilla.org.
[8] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for Reference Monitors for Android Applications. In *MoST*, 2012.
[9] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *ICSE*, 2013.
[10] S. Hao, D. Li, W. G. Halfond, and R. Govindan. SIF: A Selective Instrumentation Framework for Mobile Applications. In *MobiSys*, 2013.
[11] L. Ravindranath, J. Padhye, S. Agarwal, R. Mahajan, I. Obermiller, and S. Shayandeh. Appinsight: Mobile app performance monitoring in the wild. In *OSDI*, 2012.
[12] R. Valle-Rai, P. Co, E. Gagnon, L. J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *IBM CASC*, 1999.

APPENDIX

```java
public class AdsBlocker extends AbstractBlade {

  private static HashSet<String> adViews = new HashSet<String>(Arrays.asList(
      "com.google.ads.AdView",
      "com.google.android.gms.ads.AdView",
      "com.mopub.mobileads.MoPubView"));

  public void onCreate(Activity activity, Bundle savedInstanceState) {
    hideAllAdViews(activity.findViewById(android.R.id.content));
  }

  private void hideAllAdViews(View inputView) {
    ViewGroup viewgroup = (ViewGroup) inputView;
    int childCount = viewgroup.getChildCount();
    for (int i = 0; i < childCount; i++) {
      View v = viewgroup.getChildAt(i);
      try {
        String viewname = v.getClass().getName();
        if (adViews.contains(viewname)) {
          v.setVisibility(View.INVISIBLE);
          continue;
        }
      } catch (Exception e) {
      }
      if (v instanceof ViewGroup) {
        hideAllAdViews(v);
      }
    }
  }
}
```

Listing 2: Code for the Ad Blocker Blade. On activity creation, it finds views with IDs of known advertisements, and hides them from view.