

RICE UNIVERSITY

Separating Smartphone advertising from applications

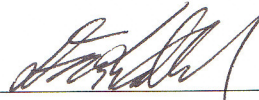
by

Shashi Shekhar

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

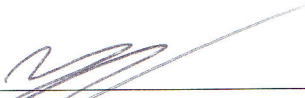
APPROVED, THESIS COMMITTEE:



Dr. Dan S. Wallach (Chair),
Associate Professor,
Computer Science



Dr. Alan L. Cox,
Associate Professor,
Computer Science



Dr. Lin Zhong
Associate Professor,
Electrical and Computer Engineering

HOUSTON, TEXAS

APRIL, 2012

Separating Smartphone advertising from applications

Shashi Shekhar

Abstract

A wide variety of smartphone applications today rely on third-party advertising services, which provide libraries that are linked into the hosting application. This situation is undesirable for both the application author and the advertiser. Advertising libraries require additional permissions, resulting in additional permission requests to users. Likewise, a malicious application could simulate the behavior of the advertising library, forging the user's interaction and effectively stealing money from the advertiser.

This thesis describes *AdSplit*, where we extended Android to allow an application and its advertising to run as separate processes, under separate user-ids, eliminating the need for applications to request permissions on behalf of their advertising libraries. We also leverage mechanisms from QUIRE to allow the remote server to validate the authenticity of client-side behavior. In this thesis, we quantify the degree of permission bloat caused by advertising, with a study of thousands of downloaded apps. *AdSplit* automatically recompiles apps to extract their ad services, and we measure minimal runtime overhead. We also observe that most ad libraries just embed an HTML widget within and describe how *AdSplit* can be designed with this in mind to avoid any need for ads to have native code.

Contents

Abstract	ii
List of Illustrations	vi
1 Introduction	1
2 Mobile Advertisements	4
2.1 Ad libraries analysis	5
2.1.1 How many ad libraries?	5
2.2 Permissions required	6
2.3 Permission bloat	7
2.3.1 Results	7
3 Design and Implementation	9
3.1 Advertisement security on the web	9
3.2 Design objectives	10
3.3 Android Background	11
3.3.1 Activities in Android	11
3.3.2 View	11
3.3.3 Activity stack	11
3.4 Advertisement system overview	12
3.5 Advertisement pairing	14
3.6 Process separation	15
3.7 Permission separation	18
3.8 Advertisement lifecycle management	18

3.9	Enabling screen sharing	19
3.9.1	AdView	20
3.9.2	Stacking advertisement and app	20
3.9.3	Layout information	22
3.10	Handling user input	23
3.11	Authenticated user input	23
4	Evaluation	26
4.1	Effect on UI responsiveness	26
4.2	Memory and CPU overhead	27
5	Advertisement separation	30
5.1	Separation for legacy apps	30
5.2	Deployment challenges	32
6	Alternative design	34
6.1	HTML Ads	34
6.1.1	WebView for ads	35
6.1.2	Specification	35
6.1.3	Handling user input	36
6.1.4	Open issues	37
6.2	Implementation	37
6.2.1	Performance	37
7	Mobile advertisements: policy enforcement	38
7.1	Policy enforcement	38
7.1.1	Advertisement blocking	38
7.1.2	Permissions and privacy	40

8 Related Work	41
8.1 Mobile Ads	41
8.2 Web security	42
8.3 JavaScript sandboxes	42
8.4 Advertisement privacy	42
8.5 Smart phone platform security	43
8.5.1 Privilege escalation	43
8.5.2 Dynamic taint analysis on Android	44
9 Future Work	46
Bibliography	48

Illustrations

2.1	Number of apps with ad libraries installed.	5
2.2	Distribution of types of permissions reduced when advertisements are separated from applications.	8
3.1	Block diagram of advertisement system showing communication between components.	13
3.2	Screen sharing between host and advertisement apps.	14
3.3	Changes to the activity launch mechanism to start advertisement process.	17
3.4	Activity launch and window creation without advertisement activity.	21
3.5	Activity launch and window creation with advertisement activity.	22
3.6	Motion event delivery to the advertisement activity.	24
4.1	Layout query time vs view depth of host activity (average of 10K runs).	29
5.1	Automated separation of advertisement libraries from their host applications.	31
6.1	Block diagram of advertisement system with HTML ads.	36

Chapter 1

Introduction

In the recent years smartphones have become increasingly popular. This popularity is partly due to their ease of use and partly due to versatile and innovative functionality that can be supported by modern smartphones. A modern smartphone not only supports high speed access to the Internet but also has significant processing power. Further smartphones provide some unique hardware input devices like GPS, accelerometer, cameras, microphones which provide access to local information about user environment. The access to Internet combined with the access to local information allows smartphones to support new types of applications, which can intelligently utilize the local context to customize user experience and provide innovative functionality.

A large part of functionality of smartphones is implemented through different applications (“apps”) developed by third party developers. Whether on the iPhone or Android platforms, apps often come in two flavors: a free version, with embedded advertising, and a pay version without. Both models have been successful in the marketplace. To pick one example, the popular Angry Birds game at one point brought in roughly equal revenue from paid downloads on Apple iOS devices and from advertising-supported free downloads on Android devices [1]. They now offer advertising-supported free downloads on both platforms.

We cannot predict whether free or paid apps will dominate in the years to come, but advertising-supported applications will certainly remain prominent. Already, a cottage industry of companies offer advertising services for smartphone application developers.

Today, these services are simply pre-compiled code libraries, linked and shipped together with the application. This means that a remote advertising server has no way to

validate a request it receives from a user legitimately clicking on an advertisement. A malicious application could easily forge these messages, generating revenue for its developer while hiding the advertisement displays in their entirety. To create a clear trust boundary, advertisers would benefit from running separately from their host applications.

In Android, applications must request permission at install time for any sensitive privileges they wish to exercise [2]. Such privileges include access to the Internet, access to coarse or fine location information, or even access to see what other apps are installed on the phone. Advertisers want this information to better profile users and thus target ads at them; in return, advertisers may pay more money to the developers of hosting applications. Consequently, many applications which require no particular permissions, by themselves, suffer *permission bloat*—being forced to request the privileges required by their advertising libraries in addition to any of their own needed privileges. Since users might be scared away by detailed permission requests, application developers would also benefit if ads could be hosted in separate applications, which might then make their own privilege requests.

Besides reducing permission bloat, separating applications from their advertisements creates better fault isolation. If the ad system fails or runs slowly, the host application should be able to carry on without inconveniencing the user. Addressing these needs requires developing a suitable software architecture, with OS assistance to make it robust.

Most of the security problems related to mobile advertisements arise due to mixing of two distinct security domains, this intermingling of security domains is a known problem on the web, where mashups originating from different websites are common. There has been several proposed solutions for alleviating security problems related to mashups on web, like separating various browser components in separate processes [3, 4], constructing browser-based multi-principal operating systems [5, 6]. Many of these proposed mechanisms can be adapted to be used in smartphones and can serve as an inspiration for the design of a secure mobile ad system.

One common mechanism by which web pages isolate advertisements from their content is by placing ads in an *iframe* [7]. The content hosted in an *iframe* is isolated from the host-

ing webpage and browsers allow only specific cross frame interactions [8, 9]. For mobile advertisements there is no primitive like an *iframe* to provide isolation. Engineering something like a full fledged *iframe* for mobile ads will be quite complex since the *iframe* itself is a powerful primitive and even allows recursive embedding. In fact due to complexity of embedding and cross frame interactions between *iframe* and hosting application there has been quite a few attacks directed to exploit embedded iframes [10, 11, 12]. While having a full fledged isolation primitive like *iframe* may be nice, the goal of this thesis is to engineer a low complexity isolation mechanism specifically for mobile ads.

The contribution of this thesis are as follows:

- *Apps and mobile advertisement analysis*: We survey a large number of actual applications and advertisement libraries used by them. We study the the specific permissions used by advertisements and empirically estimate the extent of *permission-bloat* problem.
- *Design of advertisement system*: We engineer a mechanism to separate advertisements from their hosting applications. This essentially allows applications and advertisements to run in separate application domains. We address the unique design and implementation challenges involved in enabling such a separation in an efficient manner.
- *Applying separation to existing ad libraries*: We show how our mechanism can be used by existing advertisement libraries and present a design and implementation of an automated system to separate advertisements.
- *Alternative design for mobile ads*: We propose an alternative advertisement framework that allows mobile advertisements to have a more web-like architecture, obviating the need for native code libraries.
- *Policy control of ads*: Lastly, we discuss how advertisement separation enables policy based control on mobile advertisements.

Chapter 2

Mobile Advertisements

The smartphone applications often come in two flavors: a free version, with embedded advertising, and a pay version without. Both models have been successful in the marketplace. To pick one example, the popular Angry Birds game at one point brought in roughly equal revenue from paid downloads on Apple iOS devices and from advertising-supported free downloads on Android devices [1]. They now offer advertising-supported free downloads on both platforms.

The need to monetize freely distributed smartphone applications has given rise to many different ad provider networks and libraries. The companies competing for business in the mobile ad world range from established web ad providers like Google's AdMob to a variety of dedicated smartphone advertising firms. Many of these advertisement providers use personalized advertisements and serve targeted ads to improve the effectiveness of their advertising. For serving targeted advertisements these ad libraries access private information and thus may require extra permissions to run. For example, many ad libraries request the location permission to serve geolocation based ads, any application that includes these ad libraries must also request the location permission even when application does not need it.

In order to maximize revenue many app developers include multiple ad libraries in their apps. Additionally, with so many different options for serving mobile ads there is a new trend of *advertisement aggregators* that allow an application developer to include multiple ad libraries with their app and have the aggregator choose which ads to display in order to maximize profits for the developer. Inclusion of multiple ad libraries only exacerbates the permission bloat since the application must request the super set of all permissions needed by the included ad libraries.

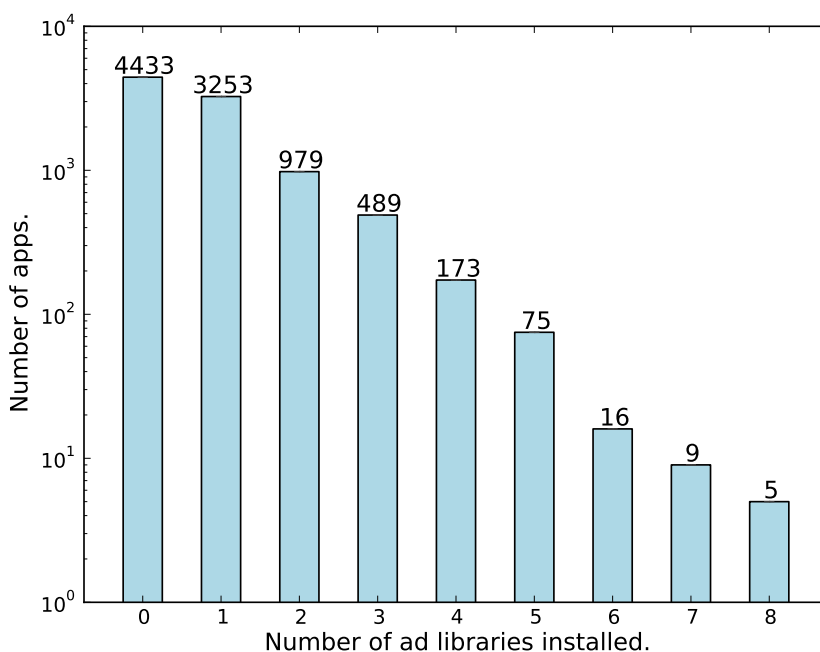


Figure 2.1 : Number of apps with ad libraries installed.

2.1 Ad libraries analysis

While we're not particularly interested in advertising market share, we want to understand how these ad libraries behave. What permissions do they require? And how many apps would be operating with fewer permissions, if only their advertisement systems didn't require them? To address these questions, we downloaded approximately 10,000 free apps from the Android Market and the Amazon App Store and built tools to analyze them.

2.1.1 How many ad libraries?

Fig 2.1 shows the distribution of number of advertisement libraries used by apps in our sample. From the distribution it is clear that many app developers choose to include multiple ad libraries in their application in order to maximize profits. Of the apps that use advertisements, about 35% include two or more advertising libraries.

2.2 Permissions required

Ad Library	Internet	Network State	Read Phone State	Write External Storage	Coarse Location	Call Phone
AdMob [13]	✓	✓			○	
Greystripe [14]	✓	✓	✓			
Millennial Media [15]	✓	✓	✓	✓		
InMobi [16]	✓	○			○	○
MobClix [17]	✓	○	✓			
TapJoy [18]	✓	✓	✓	✓		
JumpTap [19]	✓	✓	✓		○	

✓ (required), ○ (optional)

Table 2.1 : Different advertising libraries require different permissions.

We found that some ad libraries need more permissions than those mentioned in the documentation, also the set of permissions may change with the version of the ad library. Table 2.1 shows some of the required and optional permission sets for a number of popular Android ad libraries. The permissions listed as optional are not required to use the ad library but may be requested in order to improve the quality of advertisements; for example, some ad libraries will use location information to customize ads. A developer using such a library has the choice of including location-targeted ads or not. Presumably, better targeted ads will bring greater revenue to the application developer.

2.3 Permission bloat

In Android, an application requests a set of permissions at the time it's installed. Those permissions must suffice for all of the app's needs and for the needs of its advertising library. We decided to measure how many of the permissions requested are used *exclusively* by the advertising library (i.e., if the advertising library were removed, the permission would be unnecessary).

This analysis required decompiling our apps into dex format [20] using the android-apktool [21]. For each app, we then extracted a list of all API calls made. Since advertising libraries have package names that are easy to distinguish, it's straightforward to separate their API calls from the main application. To map the list of API calls to the necessary permissions, we use the data gathered by Felt et. al [2]. This allows us to compute the minimal set of permissions required by an application, with and without its advertisement libraries. We then compare this against the formal list of permissions that each app requests from the system.

There may be cases where an app speculatively attempts to use an API call that requires a permission that was never granted, or there may be dead code that exercises a permission, but will never actually run. Our analysis will err on the side of believing that an application requires a permission that, in fact, it never uses. This means that our estimates of permission bloat are strictly a lower bound on the actual volume of permissions that are requested only to support the needs of the advertising libraries.

2.3.1 Results

Our results, shown in Fig. 2.2 are quite striking. 15% of apps requesting Internet permissions are doing it for the sole benefit of their advertising libraries. 26% of apps requesting coarse location permissions are doing it for the sole benefit of their advertising libraries. 47% of apps requesting permission to get a list of the tasks running on the phone (the ad libraries use this to check if the application hosting the advertisement is in foreground) are doing it for the sole benefit of their advertising libraries. About 6.3% apps of the total

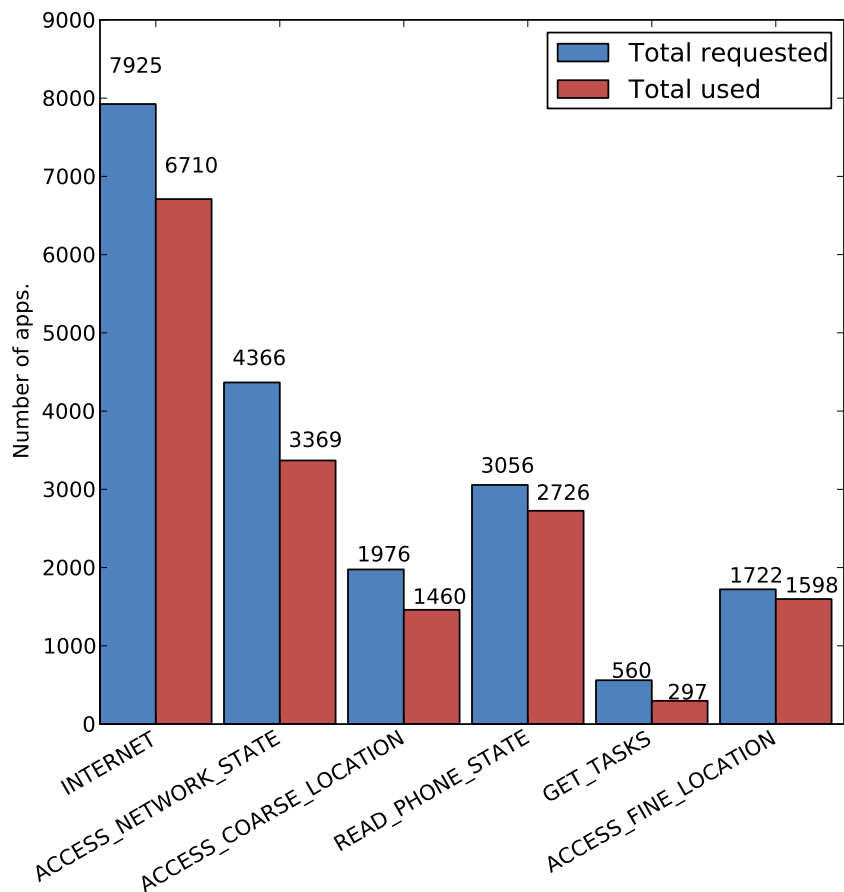


Figure 2.2 : Distribution of types of permissions reduced when advertisements are separated from applications.

9430 apps can run without any permissions once advertisements are removed. These results suggest that any architecture that separates advertisements from applications will be able to significantly reduce permission bloat.

Chapter 3

Design and Implementation

Advertisement services have been around since the very beginnings of the web. Consequently, these services have adapted to use a wide variety of technologies that should be able to influence our *AdSplit* design.

3.1 Advertisement security on the web

Fundamentally, a web page with a third-party advertisement falls under the rubric of a *mashup*, where multiple web servers are involved in the presentation of a single web page.

Many web pages isolate advertisements from content by placing ads in an *iframe* [7]. The content hosted in an *iframe* is isolated from the hosting webpage and browsers allow only specific cross frame interactions [8, 9], protecting the advertisement against intrusions from the host page. Another valuable property of the *iframe* is that it allows an external web server to distinguish requests coming from the advertisement from requests that might be otherwise forged. Standard web security mechanisms assist with this; browsers enforce the *same origin policy*, restricting the host web page from making arbitrary connections to the advertiser. Defenses against cross site request forgery, like the Origin header [22], further aid advertisers in detecting fraudulent clicks.

Adapting these ideas to a smartphone requires significant design changes. Most notably, it's common for Android applications to request the privilege to make arbitrary Internet connections. There is nothing equivalent to the same origin policy, and consequently no way for a remote server to have sufficient context, from any given click request it receives, to determine whether that click is legitimate or fraudulent. This requires *AdSplit* to include several new mechanisms.

3.2 Design objectives

The first and most prominent design decision of *AdSplit* is to separate a host application from its advertisements. This separation has a number of ramifications:

Specification for advertisements. Currently the ad libraries are compiled and linked with their corresponding host application. If advertisements are separate, then the host activities must contain the description of which advertisements to use. We introduced a method by which the host activity can specify the particular ad libraries to be used.

Permission separation. *AdSplit* allows advertisements and host applications to have distinct and independent permission sets.

Process separation. *AdSplit* advertisements run in separate processes, isolated from the host application.

Lifecycle management. Advertisements only need to run when the host application is running, otherwise they can be safely killed; similarly once the host application starts running, the associated advertisement process must also start running. Our system manages the lifecycle of advertisements.

Screen sharing. Advertisements are displayed inside host activity, so if advertisements are separated there should be a way to share screen real estate between advertisements and host application. *AdSplit* includes a mechanism for sharing screen real estate.

Authenticated user input. Advertisements generate revenue for their host applications; this revenue is typically dependent on the amount of user interaction with the advertisement. The host application can try to forge user input and generate fraudulent revenue, hence the advertisements should have a way to determine that any input events received from host application are genuine. *AdSplit* includes a method by which advertising applications can validate user input, validate that they are being

displayed on-screen, and pass that verification, in an unforgeable fashion, to their remote server.

The addition of all the above mechanisms effectively make Android advertisement aware. Before describing our implementation we discuss the relevant Android background in the next section.

3.3 Android Background

3.3.1 Activities in Android

In Android an activity is the primary mechanism for user interaction. An activity is a single, focused thing which a user can do. Each activity has associated UI window which is displayed to the user. When a user starts an application it typically launches an activity which is displayed to the user.

3.3.2 View

A view is the basic block of user interface. A view itself is a rectangular area on the screen which can have event handling code associated with it. Each activity window contains a hierarchy of views and view containers. When a window is drawn to screen, each of the individual visible views in the window are also drawn.

Every view has its associated layout information like height, width associated with it. The layout information is used for drawing the view, also for locating the target of a user input when user touches the screen.

3.3.3 Activity stack

Activities in Android are maintained on a stack. The purpose of keeping activities in a stack is to facilitate switching between activities as well as allowing user to return to previous activity by pressing the back button.

This switching between activities as well as other related functions to activity lifecycle are performed by the *ActivityManager* service.

When an activity is started the *ActivityManager* creates appropriate data structures for the activity, schedules the creation of process for activity and puts activity related information on a stack. After the activity is started *ActivityManager* schedules the activity to become visible by putting the activity in a visibility queue. The window and display associated with an activity is performed by another system component called the *WindowManager*. The *WindowManager* maintains the z-order list of windows, these windows have associations with activities. The *ActivityManager* informs the *WindowManager* about changes to activity configuration and performs appropriate calls for changing the z-order of application windows.

Since we want to factor out the advertising code into a separate process / activity, this will require a variety of changes to the activity management to ensure that the user experience is unchanged (for details about changes see section 3.6).

3.4 Advertisement system overview

An app using *AdSplit* will require the collaboration of three major components: the host activity, the advertisement activity, and the advertisement service. The host activity is the app that the user wants to run, whether a game, a utility, or whatever else. It then “hosts” the advertisement activity, which displays the advertisement. There is a one-to-one mapping between host activity and advertisement activity instances. The Unix processes behind these activities have distinct user-ids and distinct permissions granted to them. To coordinate these two activities, we have a central advertisement service. The ad service is responsible for delivering UI events to the ad activity. It also verifies that the ad activity is being properly displayed and that the UI clicks aren’t forged. Figure 3.1 shows the interaction between different components of the ad system.

AdSplit builds on QUIRE [23], which prototyped a feature shown in Fig. 3.2, allowing the host and advertisement activities to share the screen together. First the window for

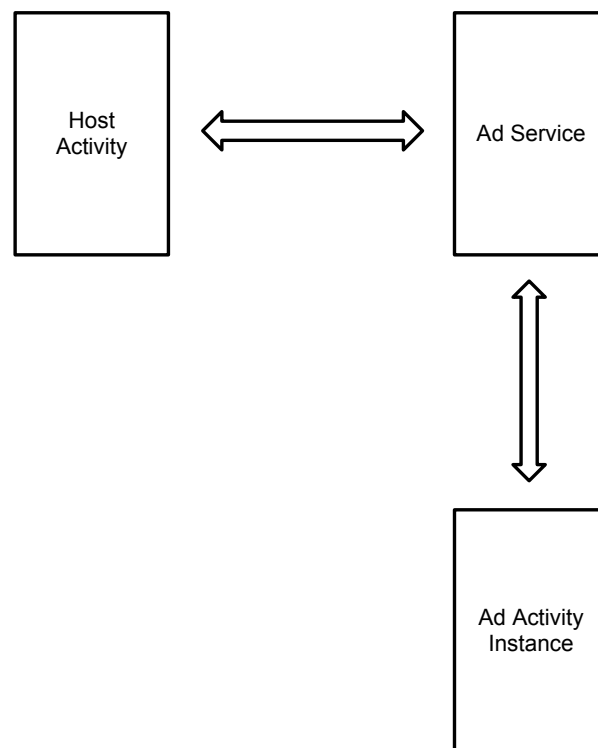


Figure 3.1 : Block diagram of advertisement system showing communication between components.

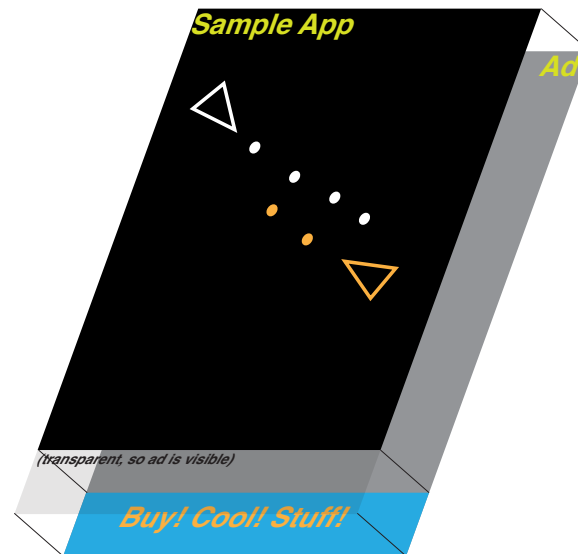


Figure 3.2 : Screen sharing between host and advertisement apps.

advertisement activity is layered just below the host activity window. The host activity window contains transparent regions where advertisement will be displayed. The advertisement activity has advertisements present at places such that they are visible through the transparent regions for the host activity. This effectively seems like advertisements are visible through holes in the host activity window. More concrete details about the above mechanism can be found in section 3.9.

3.5 Advertisement pairing

In *AdSplit*, we wish to take existing Android applications and separate out their advertising to follow the model described above. We first must explain the variety of different ways in which an existing application might arrange for an advertisement to be displayed. We will use Google's AdMob system as a running example. Other advertisement systems behave similarly, at least with respect to displaying banner ads. (For simplicity of discussion, we ignore full-screen interstitial ads.)

With current Android applications, if a developer wants to include an advertisement

from AdMob in an activity of her application, she imports the AdMob library, and then either declares an *AdMob.AdView* in the XML layout, or she generates an instance of *AdMob.AdView* and inserts it directly into the view hierarchy. This works without issue since all AdMob classes are loaded alongside the hosting application; they are separated only by having different package names.

Once we separate advertisements from applications, neither of these techniques will work, since the code isn't there any more. We first need a new mechanism. Later, in Section 5.1, we will describe how *AdSplit* does this transformation automatically.

We added a new element *AppFrame* to activity description in the application manifest (*AndroidManifest.xml*). The added element specifies the class and package names of the advertisement activity and the advertisement service. We made changes to package management and parsing mechanisms of Android for storing this information as part of activity metadata maintained by Android. This information about advertisement activities is used by the *ActivityManager* service, to manage the lifecycle of advertisements and setup correct communication links between different advertisement components.

For illustrating how advertisements are specified in *AdSplit*, let's use the familiar example of AdMob and how AdMob ads using *AdSplit* will be specified. In case of *AdSplit*, the developer declares an *AppFrame* element specifying the advertisement activity which contains AdMob, say *AdMobSeparatedActivity* and corresponding *AdMobService*. These activity and service are installed separately from application. Now at the time of activity launch the associated advertisement activity is queried and an instance *AdMobSeparatedActivity* is launched, this activity also gets information about the host activity and associated service and can configure itself and setup connection with the *AdMobService*.

3.6 Process separation

For enabling isolation the advertisement activity should run in a different process. One complication which arises from this is that advertising libraries like AdMob were engineered to have one copy running in each process. If we create a single, global instance

of any given advertising library, it won't have been engineered to maintain the state of the many original applications which hosted it.

For example, suppose app **A** is a productivity app and app **B** is a gaming app. Both **A** and **B** include the same advertisement library say AdMob. AdMob will display different kind of ads for **A** which will be relevant for a productivity app and different type of ads for **B** more relevant for a gaming app [24]. Moreover the developer of **A** may select only a few categories of ads to be displayed in her applications. This means even though both **A** and **B** share the same ad library, the ad library displays different set of ads for both of them. Further a user can switch between **A** and **B** thus if there is only a single activity for AdMob, the activity will have to switch the set of advertisements and even save the state of some of these advertisements.

A cleaner way to support this scenario is by having separate activities running AdMob for both **A** and **B**. Hence when activity **A** is launched a new process for *AdMob-A* is created and when activity **B** is launched another process *AdMob-B* running advertisements for **B** is created. Now both **A** and **B** can run concurrently and when window for activity **A** is displayed, the window for advertisement activity *AdMob-A* will be below it and when user switches to **B**, window for *AdMob-B* will be placed below. This ensures that there is no unintentional sharing between advertisements of **A** and **B**, moreover this approach does not require any changes to be made to the AdMob library.

Consequently, our advertisement service must manage distinct advertisement applications for each host application. If ten different applications include AdMob, then there need to be ten different AdMob user-ids in the system, mapping one-to-one with the host applications. The advertisement service is then responsible for ensuring that the proper host application speaks to the proper advertising application.

This is sufficient to ensure that the existing advertising libraries can run without requiring modifications. One complication concerns Android's mechanism for sharing processes across related activities. When a new activity is launched and there is already a process associated with the user-id of the application, Android will launch the new activity in the

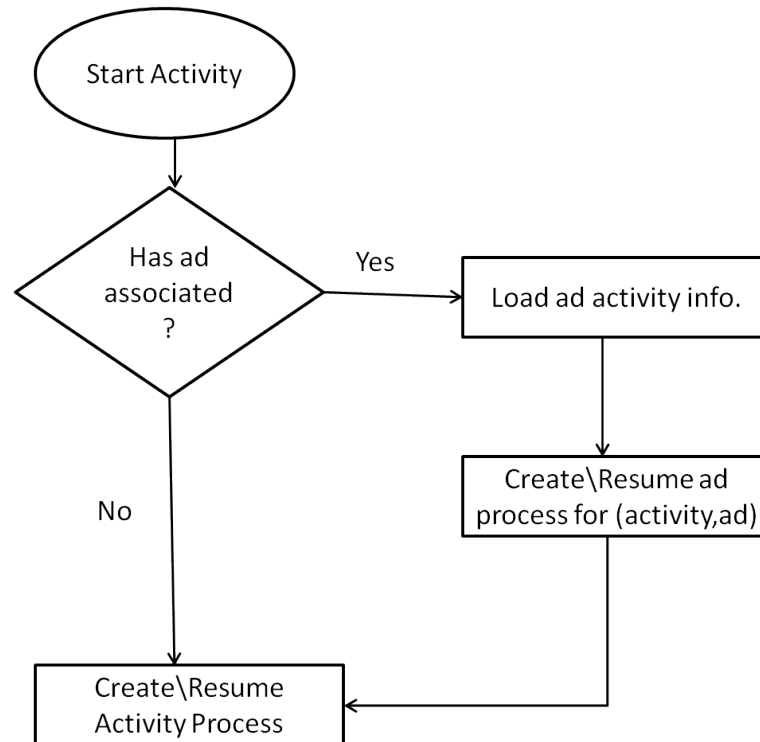


Figure 3.3 : Changes to the activity launch mechanism to start advertisement process.

same process as the old one [25]. If there is already an instance of an activity running, for example, then Android will just resume the activity and bring its activity window to the front of the stack. This is normally a feature, ensuring that there is only a single process at a time for any given application. However, for *AdSplit*, we need to ensure that advertising apps map one-to-one with hosting apps and we must ensure that their activity windows stay “glued” to their hosts’ activities. Consequently, we changed the default Android behavior such that advertisement activities are differentiated based not only by user-id, but also by the host activity. *AdSplit* thus required modest changes in how activities are launched and resumed as well as how windows are managed.

In order to have separate instance of advertisement activity per host activity, we changed the activity launch and resume mechanism, the new launch mechanism summarized in fig 3.3 associates *(host activity, advertisement activity)* tuples for the advertisement processes.

The process id associated with the advertisement activity is unique per (host,ad) tuple. So in our previous example of **A** and **B** activities. The new process for *AdMob-A* will be associated with tuple $(A, AdMob)$ so when **B** is launched, the *ActivityManager* tries to lookup process associated with $(B, AdMob)$ to launch *AdMob-B* since this tuple may not be present when a new process is created. Thus both *AdMob-A* and *AdMob-B* can coexist.

3.7 Permission separation

With Android's install-time permission system, an application requests every permission it needs at the time of its installation. As we described in Section 2.3.1, advertising libraries cause significant bloat in the permission requests made by their hosting applications. Our *AdSplit* architecture allows the advertisements to run as separate Android users with their own isolated permissions. Host applications no longer need to request permissions on behalf of their advertisement libraries.

We note that *AdSplit* makes no attempt to block a host application from explicitly delegating permissions to its advertisements. For example, the host application might obtain fine-grained location permissions (i.e., GPS coordinates with meter-level accuracy) and pass these coordinates to an advertising library which lacks any location permissions. Plenty of other Android extensions, including TaindDroid [26] and Paranoid Android [27], offer information-flow mechanisms that might be able to forbid this sort of thing if it was considered undesirable. We believe these techniques are complementary to our own, but we note that if we cannot create a hospitable environment for advertisers, they will have no incentive to run in an environment like *AdSplit*. We discuss this and other policy issues further in chapter 7.

3.8 Advertisement lifecycle management

Mobile advertisements currently are embedded in the application, so they do not require any explicit lifecycle management. When the host activity containing advertisements starts,

the advertisements which are simply a view in activity also start. Similarly when the host pauses or stops, advertisements also stop running.

Once advertisements are running in separate processes, explicit lifecycle management is needed. If a host activity starts running, the advertisement activity should also start, similarly if a host activity stops, advertisement activity must also stop. In case of *AdSplit* this lifecycle management is relatively straightforward since the information for pairing between host activity and advertisement activity is available. Thus when an activity starts (or stops) *ActivityManager* checks for any advertisements are associated with the activity and starts (or stops) these advertisements.

3.9 Enabling screen sharing

As discussed in 3.5 to insert an advertisement the developer inserts an instance of the advertisement view in the view hierarchy of the hosting activity. For example if activity **A** is using **AdMob**, the developer inserts instances of *AdMob.AdView* at places where advertisements need to be visible. These advertisement views then display advertisements.

When advertisements are separate we need a different mechanism to enable sharing of screen. This sharing of screen real estate is similar to problem which occurs in browsers where *iframe* are used for hosting content from separate domains. Many many widely used browsers however do not use different processes for iframes [28]. For iframes Gazelle [6] builds operating system principals that assign ownership of screen area, and this ownership is transferred to the iframe process through a call to the browser kernel. The browser kernel essentially maintains information about display areas and provides a syscall by which different principals can pass bitmaps for drawing.

Our goal is similar to Gazelle but instead of changing ownership of display, we share the screen through creating transparent regions for advertisements (through alpha blending). In order to enable sharing of screen between advertisement activity and host activity we introduced the mechanism by which the area for advertisement will become transparent and the advertisement activity will be visible through the transparent region. This is

illustrated in Figure 3.2. For generating the transparent region, we introduced a new type of view *AdView*. This view instead of drawing itself makes itself transparent. Further to enable the advertisement to be visible through the transparent region we stack the window of advertisement activity below the host activity. Thus if the developer of activity **A** needs to insert an instance of **AdMob** advertisement then instead of adding inserting an instance of *AdMob.AdView* she needs to insert an instance of *AdView*. This view will be transparent and the advertisement will be visible from the advertisement activity below it.

3.9.1 AdView

AdView is a special type of view which:

- is transparent.
- setups connection with the advertisement service.
- forwards any received input events to the advertisement service.

AdView thus creates the necessary holes in the host activity and forwards any received input events to the advertisement service. The advertisement service validates these events and dispatches them to the advertisement activity. The next section describes how the stacking of the advertisement and the host activity is implemented.

3.9.2 Stacking advertisement and app

In order for advertisement activity to be visible through the transparent region it should be placed just below the host activity (see Fig. 3.2). For doing that we need to change the *WindowManager* and *ActivityManager* to enable stacking of host and ad activities. Figure 3.4 shows the launch of an activity and its window creation. Figure 3.5 shows the modified launch process. In order to enable this stacking we keep track of host activity, when the host activity position is changed in the *ActivityManager* activity stack we modify the position of advertisement activity such that it is just below the host activity window. For

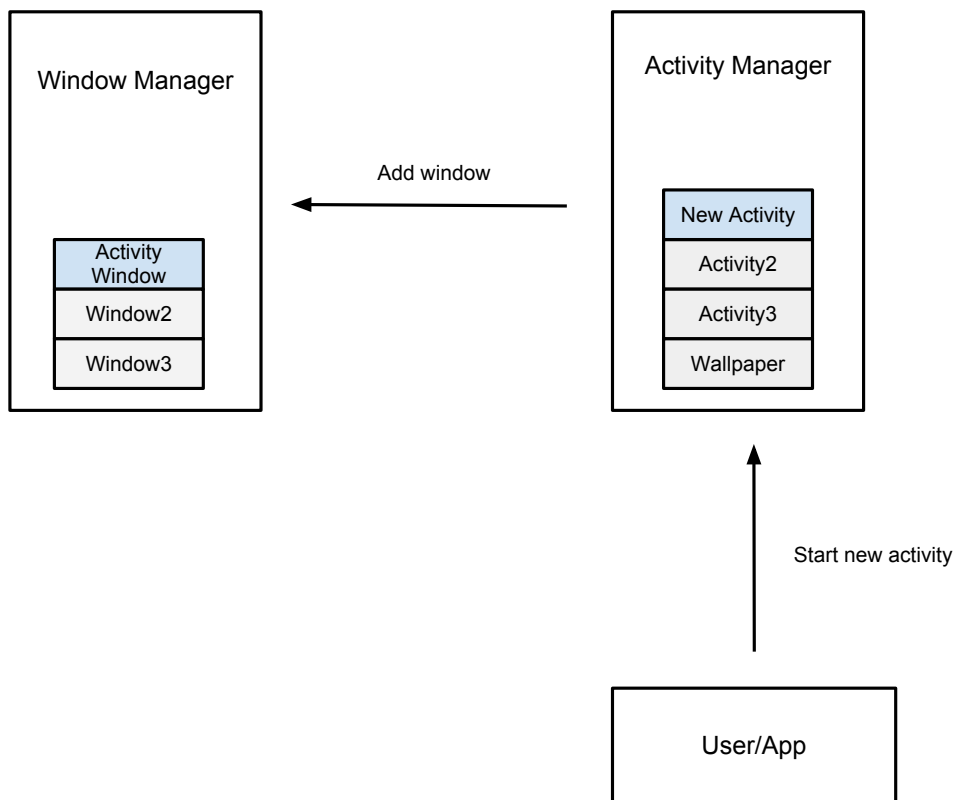


Figure 3.4 : Activity launch and window creation without advertisement activity.

example if activity **A** has the associated advertisement activity as **AdMob-A** then when **A** is launched, the *ActivityManager* checks for advertisement, it then creates a process for **AdMob-A**, further when the window for **A** is to be placed at the top, *ActivityManager* arranges the z-order of windows in *WindowManager* in such a way that **AdMob-A** window is just below **A** window. Now if user navigates to wallpaper activity (using the home button), the *ActivityManager* then removes both **A** and **AdMob-A** from the *WindowManager* list of windows. If the user relaunches the **A** activity the corresponding **AdMob-A** activity is also resumed and windows are stacked in similar manner. If **A** finishes, then *ActivityManager* also cleans up **AdMob-A** and removes the entries of advertisement activity from activity stack and run processes data structures.

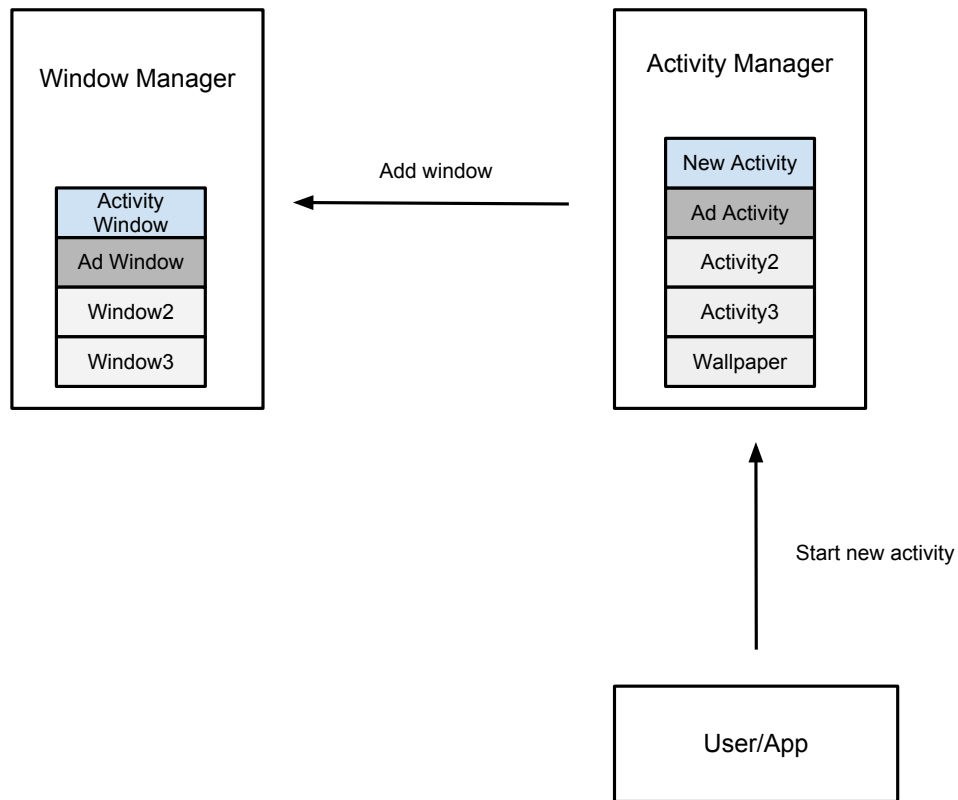


Figure 3.5 : Activity launch and window creation with advertisement activity.

3.9.3 Layout information

In order to display advertisements such that they are visible through *AdView* in the host activity, the advertisement service needs to know the layout of the hosting activity. The advertisement service can then use this information to setup advertisements at the location of *AdView* in the host activity. We modified Android to allow advertisement service to query the specific layout information from the host activity. (Layout information contains the size, position, type and visibility of views in the view hierarchy.) Our current implementation generates the representation by traversal of the host view hierarchy starting from the topmost view and generates a JSON like representation of the layout hierarchy.

The layout information queried from host is parsed by the advertisement service and

is used to generate appropriate layout for advertisement activity. For generating the layout the advertisement service first determines the size and position of *AdViews* and places the advertisement views at places where the transparent regions are present.

3.10 Handling user input

While the *AdView* solves the display sharing problem, it also needs to communicate the user input to the advertisement service to allow the user to interact with the advertisement. In order to do so when an activity is started the *AdView* sets up a connection with the advertisement service. The *AdView* implements the event handling *onTouchEvent*, *onKeyUp*, *onKeyDown* methods, these methods are called by the event system when there is a touch or key event is destined for the *AdView*. The event handling code in *AdView* simply forwards the input events to the advertisement service over the service connection setup during the initialization phase. The advertisement service is then responsible for verifying the input event and forwarding it to appropriate advertisement activity instance.

3.11 Authenticated user input

AdSplit leverages mechanisms from QUIRE [23] to detect counterfeit events, thus defeating the opportunity for an Android host application to perform a click fraud attack against its advertisers. While a variety of strategies are used to defeat click fraud on the web (see, e.g., Juels et al. [29]), we need distinct mechanisms for *AdSplit*, since a smartphone is a very different environment from a web browser.

QUIRE uses an system built around HMAC-SHA1 where every process has a shared key with a system service. This allows any process to cheaply compute a “signed statement” and send it anywhere else in the system. The ultimate recipient can then ask the system service to verify the statement. QUIRE uses this on user-generated click events, before they are passed to the host activity. The host activity can then delegate a click or any other UI event, passing it to the advertising activity which will then validate it without being

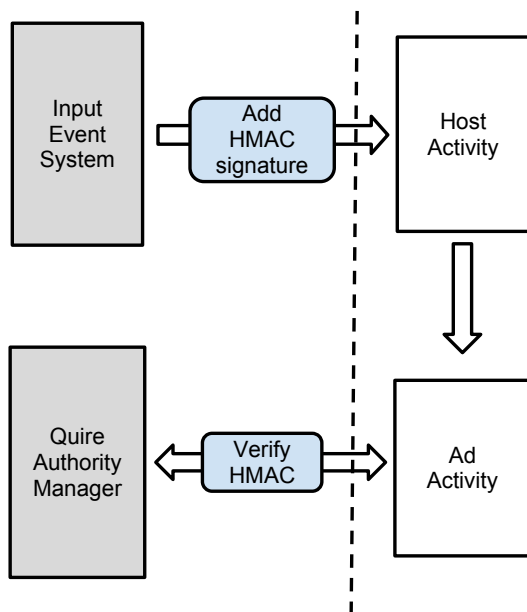


Figure 3.6 : Motion event delivery to the advertisement activity.

required to trust the host activity. The performance overhead is minimal.

QUIRE has support for making these signed statements meaningful to remote network services. Unlike the web, where we might trust a browser to speak truthfully about the context of an event (see Section 3.1), any app might potentially send any message to any network service. Instead, QUIRE provides a system service that can validate one of these messages, re-sign it using traditional public-key cryptography, and send it to a remote service over the network.

QUIRE's event delivery mechanism is summarized in Fig. 3.6. The touch event is first signed by the input system and delivered to the host activity. The stub in the host activity then forwards the touch event to advertisement service which verifies the touch event and forwards it to the advertisement activity instance. This could then be passed to another system service (not shown) which would resign and transmit the message as described above.

Despite QUIRE's security mechanisms, there are still several ways the host might attempt to defraud the advertiser. First, a host application might save old click events with valid signatures, potentially replaying them onto an advertisement. We thus include timestamps for advertisements to validate message freshness. Second, host may send genuine click events but move the *AdView*, we prevent this kind of tampering by allowing advertisement service to query layout information about the host activity. Third, a host application might attempt to hide the advertising. Android already includes mechanism for an activity to sort out its visibility to the screen [30] (touch events may include a flag that indicates the window is obscured); our advertising service uses these to ensure that the ad was being displayed at the time the click occurred.

It's also conceivable that the host application could simply drop input events rather than passing them to the advertising application. This is not a concern because the host application has no incentive to do this. The host only makes money from clicks that go through, not from clicks that are denied. (Advertising generally works on two different business models: payment per impression and payment per click. In our *AdSplit* efforts, we're focused on per-click payments, but the same QUIRE authenticated RPC mechanisms could be used in per-impression systems, with the advertisement service making remotely verifiable statements about the state of the screen.)

Chapter 4

Evaluation

In order to evaluate the performance overhead of our system we performed our experiments on a standard Android developer phone, the Nexus One, which has a 1GHz ARM core (a Qualcomm QSD 8250), 512MB of RAM, and 512MB of internal Flash storage. We conducted our experiments with the phone displaying the home screen and running the normal set of applications that spawn at start up. We replaced the default “live wallpaper” with a static image to eliminate its background CPU load. All of our benchmarks are measured using the Android Open Source Project’s (AOSP) Android 2.3 (“Gingerbread”) plus the relevant portions of QUIRE, as discussed earlier.

Our performance analysis focuses on the effect of *AdSplit* on user interface responsiveness as well as the extra CPU and memory overhead.

4.1 Effect on UI responsiveness

We performed benchmarking to determine the overhead of *AdSplit* on touch event throughput. By default Android has a 60 event per second hard coded limit; for our experiments we removed this limit. Table 4.1 shows the event throughput in terms of number of touch events per second. (The overhead added by our system is due to passing touch events from the host activity to the advertisement activity. There is also additional overhead due to the additional traversal of the view hierarchy in the advertisement activity.) We can see the our system can still support about 183 events per second which is well above the default limit of 60. Furthermore, the Nexus One is much slower than current-generation Android hardware. CPU overhead, even in this extreme case, appears to be a non-issue.

Stock Android	<i>AdSplit</i>	Ratio
229.96	183.12	0.796

Table 4.1 : Comparison of click throughput (Events/sec), averaged over 1 million events.

4.2 Memory and CPU overhead

Measuring memory overhead on Android is complicated since Android optimizes memory usage by sharing read-only data for common libraries. Consequently, if an activity has several copies of a UI widget, the effective overhead of adding a new instance of the same widget is low. Every advertisement library that we examined displays advertisements by embedding a *WebView*. A *WebView* is an instance of web browser. When the host activity already has a *WebView* instance, a fairly common practice, and it includes an advertisement, then most of the code for the advertisement *WebView* will be shared, yielding a relatively low additional overhead for the advertisement. (In our experiments we found out that multiple *WebViews* in the same activity will share their cookies, which means that an advertisement can steal cookies from any other *WebViews* in the activity.)

Consequently, in order to determine the actual memory overhead of separating advertisements from their host applications, we need to differentiate between the cases when host activities contain an instance of *WebView* and when they don't. We did our measurements by running the AdMob library, both inside the application and in a separate advertisement activity. To measure memory overhead we used procrank [31], which tells us the proportional set size (Pss) and unique set size (Uss). Pss is the amount of memory shared with other processes, divided equally among the processes who share it. Uss is the amount of memory used uniquely by the one process. Table 4.2 lists our results for the memory measurements.

In interpreting our results we are primarily concerned with the sum of Pss and Uss. From the table, we see that starting with a simply activity without any *WebView* (due to AdMob or its own), consumes about 3.9 MB. This increases to about 9 MB if the activity

Activity setup	Memory Overhead (MB)			
	Host Activity		Ad Activity	
	Pss	Uss	Pss	Uss
Without Ad or WebView	2.46	1.44	-	-
Only WebView	5.52	3.30	-	-
Only AdMob	9.67	6.58	-	-
WebView and AdMob	9.82	6.73	-	-
AdMob with <i>AdSplit</i>	2.46	1.56	9.55	6.56
WebView and AdMob with <i>AdSplit</i>	5.15	3.35	9.29	6.58

Table 4.2 : Memory overhead for host and advertisement activities with different system configurations.

has a *WebView*. Having AdMob loaded and displaying an advertisement takes about 16.3 MB of memory. When an activity has both *WebView* and AdMob, the total memory used is only about 16.5 MB, demonstrating the efficiency of Android's memory sharing.

With AdMob in a separate process, we expect to pay additional costs for Android to manage two separate activities, two separate processes, and so forth. The total memory cost in this configuration, with AdMob running in *AdSplit* and no other *WebView*, is about 20.2MB, roughly a 4 MB increase relative to AdMob running locally. Furthermore, when a separate *WebView* is running in the host activity, there is no longer an opportunity to share the cost of that *WebView*. The total memory use in this scenario is 24.4MB, or roughly an 8 MB increase relative to hosting AdMob locally. We expect we would see similar overheads with other advertising libraries.

The CPU overhead is same as the overhead of additional Dalvik virtual machine on Android, in fact since the advertisement activities run in background they run with lower priority and can be safely killed without any issues.

As briefly discussed in Section 3.11 we allow advertisement service to query layout

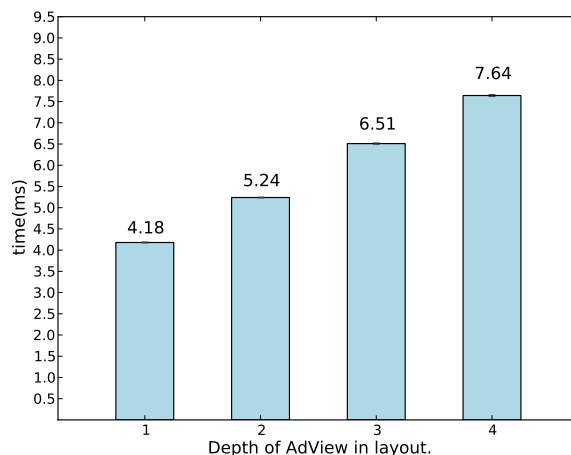


Figure 4.1 : Layout query time vs view depth of host activity (average of 10K runs).

information (type, position and transparency of views) about host activity to prevent UI rearrangement attacks. In order to evaluate the overhead of layout information queries we experimented with different view configurations for host activities and varied the depth of *AdView* in the view hierarchy. Fig. 4.1 shows how the query overhead varies with view depth. The additional depth seems to add about constant (1ms) overhead, which is non-trivial, but we expect these queries to only run once per click.

In summary, while *AdSplit* does introduce a marginal amount of additional memory and CPU cost, these will have negligible impact in practice.

Chapter 5

Advertisement separation

5.1 Separation for legacy apps

The amount of permissions requested by mobile apps and lack of information about how they are used has been a cause of concern (see, e.g., the U.S. government’s Federal Trade Commission study of privacy disclosures for children’s smartphone apps [32]). To some extent, the potential for information leakage is driven by advertisement permission bloat, so separating out the ad systems and treating them distinctly is a valuable goal.

As we showed in Section 2.3.1, a significant number of current apps with embedded advertising libraries would immediately benefit from *AdSplit*, reducing the permission bloat necessary to host embedded ads. This section describes a proof-of-concept implementation that can automatically rewrite an Android application to use *AdSplit*. Something like this could be deployed in an app store or even directly on the smartphone itself.

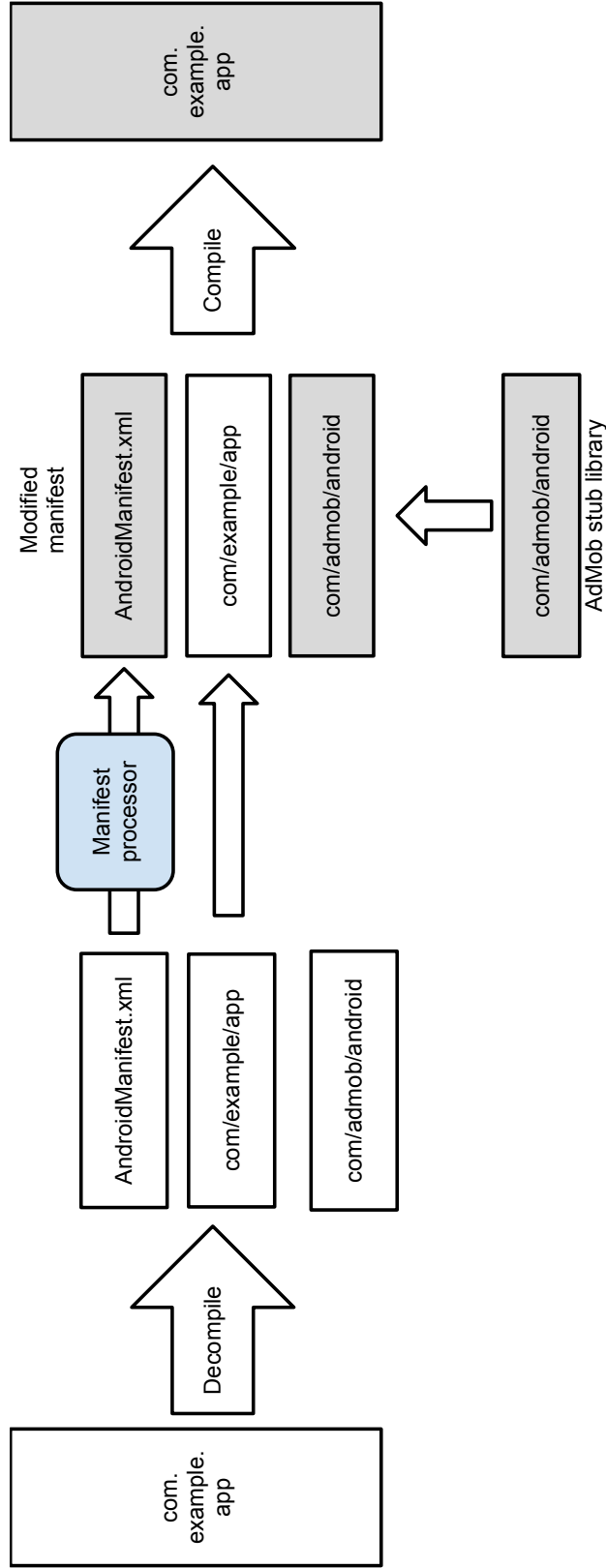


Figure 5.1 : Automated separation of advertisement libraries from their host applications.

Figure 5.1 sketches the rewriting process. First the application is decompiled using android-apktool, converting dex bytecode into smali files. (Smali is to dex bytecode what assembly language is to binary machine code; smali is the human-readable version.) Because smali files are organized into directories based on their package names, it's trivial to distinguish the advertisement libraries from their hosting applications. All we have to do is delete the advertisement code and drop in a stub library, supporting the same API, which calls out to the *AdSplit* advertisement service. We also analyze the permissions required without the advertisement present (see Section 2.3.1) and remove permission requests which are no longer necessary, and edit the manifest appropriately.

For our proof of concept, we decided to focus our attention on AdMob. Our techniques would easily generalize to support other advertising libraries, if desired. (Although we believe we have a better solution, described next in Section 6.1.)

Our stub library was straightforward to implement. We manually implemented a handful of public methods from the AdMob library, whereafter we constructed a standard Android IPC message to send to the *AdSplit* advertising service. It worked.

While it would be tempting to use automated tools to translate an entire API in one go, any commercial implementation would require significant testing and, inevitably, there would be corner cases where the automated tool didn't quite do the right thing. Instead, since there are a fairly small number of advertising vendors, we imagine that each one would best be supported by hand-written code, perhaps even supplied directly by the vendor in collaboration with an app store.

5.2 Deployment challenges

Unfortunately, there are a number of significant problems that would stand in the way of an automated rewriting architecture becoming the preferred method of deploying *AdSplit*.

Ad installation. When advertisements exist as distinct applications in the Android ecosystem, they will need to be installed somehow. We're hesitant to give the host ap-

plication the necessary privileges to install a third-party advertising application. Perhaps an application could declare that it had a dependency on a third-party app, and the main installer could hide this complexity from the user, in much the same way that common Linux package installers will follow dependencies as part of the installation process for any given target.

Ad permissions. Even if we can get the ad libraries installed, we have the challenge of understanding what permissions to grant them. Particularly when many advertising libraries know how to make optional use of a permission, such as measuring the smartphone's location if it's allowed, how should we decide if the advertisement application has those permissions? Must we install multiple instances of the advertising application based on the different subsets of permissions that it might be granted by the host application? Alternatively, should we go with a one-size-fits-all policy akin to the web's same-origin-policy? What's the proper "origin" for an application that was installed from an app store? Unfortunately, there is no good solution here, particularly not without generating complex user interfaces to manage these security policies.

Similarly, what should we do about permissions that many users will find to be sensitive, such as learning their fine-grained location, their phone number, or their address book? Again, the obvious solutions involve creating dialog boxes and/or system settings that users must interact with, which few user will understand, and which advertisers and application authors will all hate.

Ad unloading. Like any Android application, an advertisement application must be prepared to be killed at any time—a consequence of Android's resource management system. This could have some destabilizing consequences if the hosting application is trying to communicate with its advertisement and the ad is killed. Also, what happens if a user wants to uninstall an advertising application? Should that be forbidden unless every host application which uses it is also uninstalled?

Chapter 6

Alternative design

While struggling with the shortcomings outlined in Section 5.2 with the installation and permissions of advertising applications, we hit upon an alternative approach that uses the same *AdSplit* architecture. The solution is to expand on something that advertising libraries are already doing: HTML.

6.1 HTML Ads

If a customer want to purchase advertising on smartphones, they probably want to specify their advertisements the same way they do for the web: as plain text, images, or perhaps as a “rich” ad using JavaScript. Needless to say, a wide variety of tools are available to create and manage such ads, and mobile advertising providers want to make it easy for ads to appear on any platform (iPhone, Android, etc.) without requiring heroic effort from their customer.

Consequently, all of the advertising libraries we examined simply include a `WebView` within themselves. All of the native Android code is really nothing more than a wrapper around a `WebView`. Based on this insight, we suggest that *AdSplit* will be easiest to deploy by providing a single advertising application, build into the Android core distribution, that satisfies the typical needs of Android advertising vendors.

Installation becomes a non-issue, since the only advertiser-provided content in the system is HTML, JavaScript, and/or images. We still use the rest of the *AdSplit* architecture, running the `WebView` with a separate user-id, in a separate process and activity, ensuring that a malicious application cannot tamper with the advertisements it hosts. We still have the *AdSplit* advertisement service, leveraging QUIRE, to validate user events before passing

them onto the WebView. We only need to extend the WebView's outbound HTTP transactions to include QUIRE RPC signatures, allowing the remote advertising server to have confidence in the provenance of its advertising clicks.

Security permissions are more straightforward. The same-origin-policy, standard across all the web, applies perfectly to HTML *AdSplit*. Since the Android WebView is built on the same Webkit browser as the standalone "Browser" application, it has the same security machinery to enforce the same-origin-policy. Thus permissions are granted to an origin rather than the native code ad library. Further the user can optionally revoke the permission at run time.

6.1.1 WebView for ads

Keeping all this in mind we prototyped a new form of WebView specifically targetted for HTML ads : the AdWebView. The AdWebView is a way to host HTML ads in a constrained manner. We introduced two advertisement specific permissions which can be controlled by the user. These permissions control whether ads can make internet connections or use HTML5 geoLocation api.

Figure 6.1 shows the communication between different components with ads hosted in the AdWebView.

When an ad inside AdWebView requests to load a url or performs call to HTML5 geolocation api, the AdWebView performs a permission check to verify if the associated advertisement origin has the needed advertisement permission. These advertisement permissions can be managed by the user.

The AdWebView is by itself hosted in Ad Activity which has internet and location permissions. This activity simply serves as the container for the AdWebViews.

6.1.2 Specification

For specification our prototype needs two pieces of information.

- The initial URL for the advertisement to load: *initURL*.

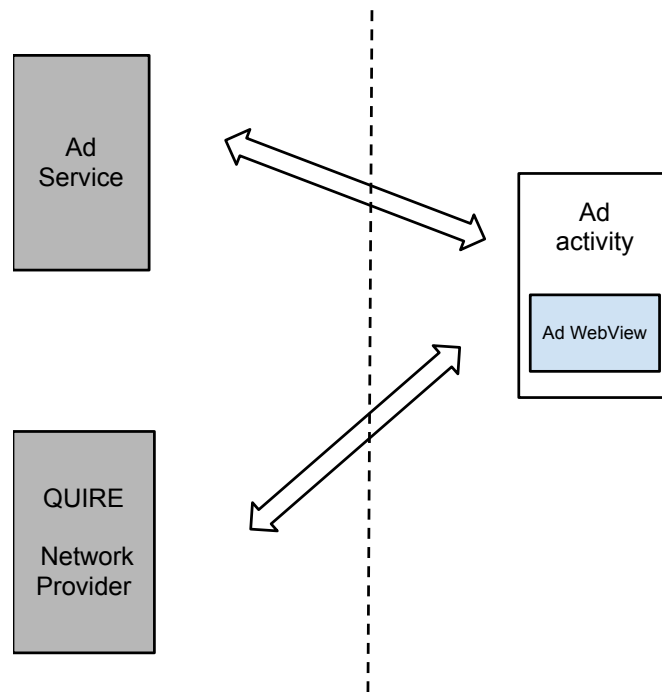


Figure 6.1 : Block diagram of advertisement system with HTML ads.

- An event handler URL for sending authenticated RPC when user interaction happens: *onUserInteraction*.

These two properties are specified by developer when she inserts the AdWebView in the view hierarchy.

6.1.3 Handling user input

The AdWebView on initialization requests to load the *initURL*, before making a network request there is a permission check to ensure that origin for *initURL* actually has internet permission. If the permission check is successful then the advertisement is fetched and loaded.

When a user interacts with the advertisement, the AdWebView receives an event, it forwards the even type information, along with relevant details(the coordinates, duration etc) to the *onUserInteraction* URL. This outgoing request contains an **X-Quire-Event** header

which contains the serialized call chain of the event. Thus a remote server can verify the call chain to validate if user interaction is legitimate.

6.1.4 Open issues

About the only open policy question is whether we should allow *AdSplit* HTML advertisements to maintain long-term tracking cookies or whether we should disable any persistent state. Certainly, persistent cookies are a standard practice for web advertising, so they seem like a reasonable feature to support here as well. *AdWebView*, by default, doesn't support persistent cookies, but it would be trivial to add.

6.2 Implementation

We built an advertising application that embeds *AdWebView* widget, as discussed above. The host application in this case specifies the URL of the advertisement server to be loaded in the *AdWebView* at initialization. We were successfully in downloading and running advertisements from our sample advertisement server.

6.2.1 Performance

Memory and performance overheads are indistinguishable from our AdMob experiments. Both versions host a *WebView* in a separate process, and it's the same HTML/JavaScript content running inside the *WebView*.

Chapter 7

Mobile advertisements: policy enforcement

7.1 Policy enforcement

While *AdSplit* allows for and incentivizes applications to run distinct from their advertisements, there are a variety of policy and user experience issues that we must still address.

7.1.1 Advertisement blocking

Once advertisements run as distinct processes, some fraction of the Android users will see this as an opportunity to block advertisements for good. Certainly, with web browsers, extension like Adblock and Adblock Plus are incredibly popular. The Chrome web store lists these two extensions in its top six¹ with “over a million” installs of each. (Google doesn’t disclose exact numbers.)

The Firefox add-ons page offers more details, claiming that Adblock Plus is far and away the most popular Firefox extension, having been installed just over 14 million times, versus 7 million for the next most popular extension². The Mozilla Foundation estimates that 85% of their users have installed an extension [33]. Many will install an ad blocker.

To pick one example, Ars Technica, a web site popular with tech-savvy users, estimated that about 40% of its users ran ad blockers [34]. At point, it added code to display blank pages to these users in an attempt to cajole them into either paying for ad-free “premium” service, or at least configuring their ad blocker to “white list” the Ars Technica website.

Strategies such as this are perilous. Some users, faced with a broken web site, will

¹<https://chrome.google.com/webstore/category/popular>

²<https://addons.mozilla.org/en-US/firefox/extensions/?sort=users>

simply stop visiting it rather than trying to sort out why it's broken. Of course, many web sites instead employ a variety of technical tricks to get around ad blockers, ensuring their ads will still be displayed.

Given what's happening on the web, it's reasonable to expect a similar fraction of smart-phone users might want an ad blocker if it was available, with the concomitant arms race in ad block versus ad display technologies.

So long as users have not "rooted" their phones, a variety of core Android services can be relied upon by host applications to ensure that the ads they're trying to host are being properly displayed with the appropriate advertisement content. Similarly, advertising applications (or HTML ads) can make SSL connections to their remote servers, and even embed the proper remote server's public key certificate, to ensure they are downloading data from the proper source, rather than empty images from a transparent proxy.

Once a user has rooted their phone, of course, all bets are off. While it's hard to measure the total number of rooted Android phones, the CyanogenMod Android distribution, which requires a rooted phone for installation, is installed on roughly 722 thousand phones³—a tiny fraction of the hundreds of millions of Android phones reported to be in circulation [35]. Given the relatively small market share where such hacks might be possible, advertisers might be willing to cede this fraction of the market rather than do battle against it.

Consequently, for the bulk of the smartphone marketplace, *advertising apps on Android phones offer greater potential for blocking-detection and blocking-resistance than advertising on the web*, regardless of whether they are served by in-process libraries or by *AdSplit*. Given all the other benefits of *AdSplit*, we believe advertisers and application vendors would prefer *AdSplit* over the status quo.

³<http://stats.cyanogenmod.com/>

7.1.2 Permissions and privacy

Some advertisers would appear to love their ability to learn additional data about the user, including location information, address book, other apps running on the phone, and so forth. This information can help profile a user, which can help target ads. Targeted ads, in turn, are worth more money to the advertiser and thus worth more money to the hosting application. When we offer HTML style advertisements, with HTML-like security restrictions, the elegance of the solution seems to go against the higher value profiling that advertisers desire.

Leaving aside whether it's *legal* for advertisers to collect this information, we have suggested that a host application could make its own requests that violate the users' privacy and pass these into the *AdSplit* advertising app. We hope that, if we can successfully reduce apps' default requests for privileges that they don't really need, then users will be less accustomed to seeing such permission requests. When they do occur, users will push back, refusing to install the app. (Reading through the user-authored comments in the Android Market, many apps with seemingly excessive permission requirements will often have scathing comments from users, along with technical justifications posted by the app authors to explain why each permission is necessary.)

Furthermore, if advertisers ultimately prefer the *AdSplit* architecture, perhaps due to its improved resistance to click fraud and so forth, then they will be forced to make the trade-off between whether they prefer improved integrity of their advertising platform, or whether they instead want less integrity but more privacy-violating user details.

Chapter 8

Related Work

8.1 Mobile Ads

There has been some recent (concurrent) work which analyses mobile advertisements and myriad of security and privacy problems associated with them. Grace et al. [36] performed static analysis of 100 thousand Android apps and found advertisement libraries uploading sensitive information to remote ad servers. They also found that some advertisement libraries were fetching and dynamically executing code from remote ad servers. With *AdSplit* advertisement libraries run in separate processes from host applications and have limited permissions thus *AdSplit* prevents opportunities of stealing sensitive information as well as executing code in host application domain. AdDroid [37] proposes separation of advertisements similar to section 6.1 by introducing a system service for advertisements. AdDroid does not run advertisement code in separate processes instead allows advertisements to be only requested by the advertisement service. AdDroid thus does not mitigate the host application from generating fraudulent revenue but is designed to enhance privacy of host applications. Pathak et.al [38] analysed the energy spent in popular mobile apps and found that 65%-75% energy of apps is spent in third party advertisement libraries. As discussed in chapter 7 *AdSplit* facilitates implementation of a variety of policies hence to prevent advertisements from using too much energy, a suitable energy consumption policy can be specified and advertisements using a significant amount of energy can be killed safely without affecting the host application.

8.2 Web security

AdSplit considers an architecture to allow for controlled mashups of advertisements and applications on a smartphone. The web has been doing this for a while (as discussed in Section 3.1). Additionally, researchers have considered a variety of web extensions to further contain browser components in separate processes [3, 4], including constructing browser-based multi-principal operating systems [5, 6].

8.3 JavaScript sandboxes

Caja [39] and ADsafe [40] work as JavaScript sandboxes which use static and dynamic checks to safely host JavaScript code. They use a safe subset of JavaScript, eliminating dangerous primitives like `eval` or `document.write` that could allow an advertisement to take over an entire web page. Instead, advertisements are given a limited API to accomplish what they need. *AdSplit* can trivially host advertisements built against these systems, and as their APIs evolve, they could be directly supported by our `AdWebView` class. Additionally, because we run the `AdWebView` in a distinct process with its own user-id and permissions, we provide a strong barrier against advertisement misbehavior impacting the rest of the platform.

8.4 Advertisement privacy

Privad [41] and Juels et al. [42] address security issues related to privacy and targeted advertising for web ads. They use client side software that prevents behavior profiling of users and allows targeted advertisements without compromising user privacy.

AdSplit does not address privacy problems related to targeted advertisements but it provides framework for implementing various policies on advertisements.

8.5 Smart phone platform security

As mobile phone hardware and software increase in complexity the security of the code running on a mobile devices has become a major concern.

The Kirin system [43] and Security-by-Contract [44] focus on enforcing install time application permissions within the Android OS and .NET framework respectively. These approaches to mobile phone security allow a user to protect themselves by enforcing blanket restrictions on what applications may be installed or what installed applications may do, but do little to protect the user from applications that collaborate to leak data or protect applications from one another.

Saint [45] extends the functionality of the Kirin system to allow for runtime inspection of the full system permission state before launching a given application. Apex [46] presents another solution for the same problem where the user is responsible for defining run-time constraints on top of the existing Android permission system. Both of these approaches allow users to specify static policies to shield themselves from malicious applications, but don't allow apps to make dynamic policy decisions.

CRPE [47] presents a solution that attempts to artificially restrict an application's permissions based on environmental constraints such as location, noise, and time-of-day. While CRPE considers contextual information to apply dynamic policy decisions, it does not attempt to address privilege escalation attacks.

8.5.1 Privilege escalation

XManDroid [48] presents a solution for privilege escalation and collusion by restricting communication at runtime between applications where the communication could open a path leading to dangerous information flows based on Chinese Wall-style policies [?] (e.g., forbidding communication between an application with GPS privileges and an application with Internet access). While this does protect against some privilege escalation attacks, and allows for enforcing a more flexible range of policies, applications may launch denial of service attacks on other applications (e.g., connecting to an application and thus preventing

it from using its full set of permissions) and it does not allow the flexibility for an application to regain privileges which they lost due to communicating with other applications.

One feature of QUIRE that is not used in *AdSplit* is its ability to defeat confused deputy attacks, by annotating IPCs with the entire call chain. In concurrent work to QUIRE, Felt et al. present a solution to what they term “permission re-delegation” attacks against deputies on the Android system [49]. With their “IPC inspection” system, apps that receive IPC requests are poly-instantiated based on the privileges of their callers, ensuring that the callee has no greater privileges than the caller. IPC inspection addresses the same confused deputy attack as QUIRE’s “security passing” IPC annotations, however the approaches differ in how intentional deputies are handled. With IPC inspection, the OS strictly ensures that callees have reduced privileges. They have no mechanism for a callee to deliberately offer a safe interface to an otherwise dangerous primitive. Unlike QUIRE, however, IPC inspection doesn’t require apps to be recompiled or any other modifications to be made to how apps make IPC requests.

(*AdSplit* does not require QUIRE’s IPC inspection system, and thus also does not require apps to be recompiled to have the semantics described in this paper.)

More recent work has focused on kernel extensions that can observe IPC traffic, label files, and enforce a variety of policies [50, 51]. These systems can enhance the assurance of many of the above techniques by centralizing the policy specification and enforcement mechanisms.

8.5.2 Dynamic taint analysis on Android

The TaintDroid [26] and ParanoidAndroid [27] projects present dynamic taint analysis techniques to preventing runtime attacks and data leakage. These projects attempt to tag objects with metadata in order to track information flow and enable policies based on the path that data has taken through the system. TaintDroid’s approach to information flow control is to restrict the transmission of tainted data to a remote server by monitoring the outbound network connections made from the device and disallowing tainted data to flow

along the outbound channels.

AdSplit allows ads to run in separate processes but applications can still pass sensitive information to separated advertisements. TaintDroid and ParanoidAndroid can be used to detect and prevent any such flow of information. Thus they are complementary to *AdSplit*.

Chapter 9

Future Work

The work in this thesis touches on a trend that will become increasingly prevalent over the next several years: the merger of the HTML security model and the smartphone application security model. Today, HTML is rapidly evolving from its one-size-fits-all security origins to allow additional permissions, such access to location information, for specific pages that are granted those permissions by the user. HTML extensions are similarly granted varying permissions rather than having all-or-nothing access [52, 53].

On the flip side, iOS apps originally ran with full, unrestricted access to the platform, subject only to vague policies enforced by human auditors. Only access to location information was restricted. In contrast, the Android security model restricts the permissions of apps, with many popular apps running without any optional permissions at all. Despite this, Android malware is a growing problem, particularly from third-party app stores (see, e.g., [54, 55]). Clearly, there's a need for more restrictive Android security, more like the one-size-fits-all web security model.

While it's unclear exactly how web apps and smartphone apps will eventually become one thing, this thesis shows where this merger is already underway: when web content is embedded in a smartphone app. Well beyond advertising, a variety of smartphone apps take the strategy of using native code to set up one or more web views, then to the rest in HTML and JavaScript. This has several advantages: it makes it easier to support an app across many different smartphone platforms. It also allows authors to quickly update their apps, without needing to go through a third-party review process.

These trends, plus the increasing functionality in HTML5, suggest that “native” apps may well be entirely supplanted by some sort of “mobile HTML” variant, not unlike

HP/Palm's WebOS, where every app is built this way¹. For this convergence of mobile and web apps to be smooth, the various security design issues for bridging the gap between native and web apps need to be tackled. Our future work will be pursuing these interesting security challenges.

¹<http://developer.palm.com/blog>

Bibliography

- [1] T. Cheshire, *In depth: How Rovio made Angry Birds a winner (and what's next)*. *Wired*, Mar. 2011. <http://www.wired.co.uk/magazine/archive/2011/04/features/how-rovio-made-angry-birds-a-winner>.
- [2] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, (Chicago, Illinois, USA), ACM, 2011.
- [3] C. Grier, S. Tang, and S. T. King, "Secure web browsing with the op web browser," in *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, IEEE Computer Society, May 2008.
- [4] C. Reis and S. D. Gribble, "Isolating web programs in modern browser architectures," in *Proceedings of the 4th ACM European conference on Computer systems*, EuroSys '09, (Nuremberg, Germany), ACM, Apr. 2009.
- [5] J. Howell, C. Jackson, H. J. Wang, and X. Fan, "MashupOS: Operating system abstractions for client mashups," in *Proceedings of the 11th USENIX Workshop on Hot Topics in Operating Systems (HotOS '07)*, pp. 1–7, 2007.
- [6] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter, "The multi-principal OS construction of the Gazelle web browser," in *Proceedings of the 18th USENIX Security Symposium*, 2009.
- [7] World Wide Web Consortium (W3C), *Frames in HTML Documents*, Nov. 2011. <http://www.w3.org/TR/REC-html40/present/frames.html#h-16.5>.

- [8] A. Barth, C. Jackson, and J. C. Mitchell, “Securing frame communication in browsers,” in *Proceedings of the 17th USENIX Security Symposium (USENIX Security 2008)*, 2008.
- [9] MSDN, *About Cross-Frame Scripting and Security*, Oct. 2011. [http://msdn.microsoft.com/en-us/library/ms533028\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ms533028(v=vs.85).aspx).
- [10] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson, “Busting frame busting: a study of clickjacking vulnerabilities at popular sites,” in *IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010. <http://seclab.stanford.edu/websec/framebusting/framebust.pdf>.
- [11] N. Provos, P. Mavrommatis, M. A. Rajab, and F. Monrose, “All your iFRAMEs point to us,” in *Proceedings of the 17th USENIX Security Symposium*, (San Jose, CA), 2008.
- [12] G. Rydstedt, E. Bursztein, and D. Boneh, “Framing attacks on smart phones and dumb routers: Tap-jacking and geo-localization,” in *Usenix Workshop on Offensive Technologies (wOOT 2010)*, 2010. <http://seclab.stanford.edu/websec/framebusting/tapjacking.pdf>.
- [13] Google Inc., *Google AdMob Ads Android Fundamentals*, Nov. 2011. <http://code.google.com/mobile/ads/docs/android/fundamentals.html>.
- [14] GreyStripe Inc., *Android - SDK Integration Overview*, Nov. 2011. <http://wiki.greystripe.com/index.php/Android#AndroidManifest.xml>.
- [15] Millennial Media, *Millennial Media Android SDK - Version 4.5.0*, Nov. 2011. <http://wiki.millennialmedia.com/index.php/Android>.
- [16] InMobi, *InMobi Android SDK - Version a300*, Nov. 2011. <http://developer.inmobi.com/wiki/index.php?title=Android>.

- [17] Mobclix, *Mobclix SDK Integration Guide Version 3.1.0*, Nov. 2011. https://developer.mobclix.com/help/advertising/sdk_api/android.
- [18] Tapjoy, *Getting Started with Publisher SDK*, Nov. 2011. <http://knowledge.tapjoy.com/integration-8-x/android/publisher/getting-started-with-offers-sdk>.
- [19] Jumptap, *Jumptap Android SDK Integration*, Nov. 2011. https://support.jumptap.com/index.php/Jumptap_Android_SDK_Integration.
- [20] Android Open Source Project., *dex - Dalvik Executable Format*, Nov. 2007. <http://source.android.com/tech/dalvik/dex-format.html>.
- [21] Google Project Hosting., *android-apktool - A tool for reengineering Android apk files*, Feb. 2012. <http://code.google.com/p/android-apktool>.
- [22] A. Barth, C. Jackson, and J. C. Mitchell, “Robust defenses for cross-site request forgery,” in *15th ACM Conference on Computer and Communications Security (CCS '08)*, (Alexandria, VA), Oct. 2008.
- [23] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, “Quire: Lightweight provenance for smart phone operating systems,” in *20th USENIX Security Symposium*, (San Francisco, CA), Aug. 2011.
- [24] AdMob, *Types of Ads That Appear on Your App or Site — Welcome to AdMob Help*, Nov. 2011. <http://helpcenter.admob.com/content/types-ads-appear-your-app-or-site>.
- [25] Android, *Processes and Threads — Android Developers*, Nov. 2011. <http://developer.android.com/guide/topics/fundamentals/processes-and-threads.html>.
- [26] W. Enck, P. Gilbert, C. Byung-gon, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, “TaintDroid: An information-flow tracking system for realtime privacy monitoring on

- smartphones,” in *Proceeding of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI '10)*, pp. 393–408, 2010.
- [27] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, “Paranoid Android: Zero-day protection for smartphones using the cloud,” in *Annual Computer Security Applications Conference (ACSAC '10)*, (Austin, TX), Dec. 2010.
- [28] Google Project Hosting, *Issue 99379 - chromium - Out of process iframes.*, Oct. 2011. <http://code.google.com/p/chromium/issues/detail?id=99379>.
- [29] A. Juels, S. Stamm, and M. Jakobsson, “Combating click fraud via premium clicks,” in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, (Boston, MA), USENIX Association, 2007.
- [30] Google, *View: Android developer reference*, Feb. 2011. <http://developer.android.com/reference/android/view/View.html#Security>.
- [31] eLinux.org, *Android Memory Usage*, Feb. 2012. http://elinux.org/Android_Memory_Usage.
- [32] Federal Trade Commission, *Mobile Privacy for Kids: Current Privacy Disclosures are Disappointing*, Feb. 2012. http://ftc.gov/os/2012/02/120216mobile_apps_kids.pdf.
- [33] Mozilla Foundation, *How Many Firefox Users Have Add-Ons Installed? 85%!*, June 2011. <http://blog.mozilla.com/addons/2011/06/21/firefox-4-add-on-users/>.
- [34] L. McGann, *How Ars Technica’s “experiment” with ad-blocking readers built on its community’s affection for the site.* Nieman Journalism Lab, Mar. 2010. <http://www.niemanlab.org/2010/03/how-ars-technica-made-the-ask-of-ad-blocking-readers/>.

- [35] M. Panzarino, *Google: About 190 Million Android Devices Activated Worldwide. That's About 576900 A Day Since May.* The Next Web, Oct. 2011. <http://thenextweb.com/google/2011/10/13/google-190-million-android-devices-activated-worldwide-thats-about->
- [36] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec '12*, (Tucson, Arizona), Apr. 2012.
- [37] P. Pearce, A. P. Felt, and D. Wagner, "Addroid: Privilege separation for applications and advertisers in android," in *7th ACM Symposium on Information, Computer and Communications Security (AsiaCCS)*, AsiaCCS '12, (Seoul, Korea), May 2012.
- [38] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof," in *Proceedings of the 7th ACM european conference on Computer Systems, EuroSys '12*, (Bern, Switzerland), Apr. 2012.
- [39] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, *Caja Safe active content in sanitized JavaScript.* Google, Dec. 2007. <http://google-caja.googlecode.com/files/caja-2007.pdf>.
- [40] ADsafe, *ADsafe*, Feb. 2012. <http://www.adsafe.org>.
- [41] S. Guha, B. Cheng, and P. Francis, "Privad: Practical privacy in online advertising," in *Proceedings of the 8th Symposium on Networked Systems Design and Implementation (NSDI)*, (Boston, MA), Mar 2011.
- [42] A. Juels, "Targeted advertising ... and privacy too," in *Proceedings of the 2001 Conference on Topics in Cryptology: The Cryptographer's Track at RSA, CT-RSA 2001*, Springer-Verlag, 2001.

- [43] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *16th ACM Conference on Computer and Communications Security (CCS '09)*, (Chicago, IL), Nov. 2009.
- [44] L. Desmet, W. Joosen, F. Massacci, P. Philippaerts, F. Piessens, I. Siahaan, and D. Vanoverberghe, "Security-by-contract on the .NET platform," *Information Security Technical Report*, vol. 13, no. 1, pp. 25–32, 2008.
- [45] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, "Semantically rich application-centric security in Android," in *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC '09)*, (Honolulu, HI), Dec. 2009.
- [46] M. Nauman, S. Khan, and X. Zhang, "Apex: extending Android permission model and enforcement with user-defined runtime constraints," in *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pp. 328–332, 2010.
- [47] M. Conti, V. T. N. Nguyen, and B. Crispo, "CRPE: Context-related policy enforcement for Android," in *Proceedings of the Thirteen Information Security Conference (ISC '10)*, (Boca Raton, FL), Oct. 2010.
- [48] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "XManDroid: A new Android evolution to mitigate privilege escalation attacks," Tech. Rep. TR-2011-04, Technische Universität Darmstadt, Apr. 2011. http://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/xmandroid.pdf.
- [49] A. P. Felt, H. J. Wang, A. Moshchuck, S. Hanna, and E. Chin, "Permission re-delegation: Attacks and defenses," in *20th Usenix Security Symposium*, (San Francisco, CA), Aug. 2011.
- [50] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on Android," in *Proc. of the 19th Network*

and Distributed System Security Symposium (NDSS 2012), (San Diego, CA), Feb. 2012.

- [51] S. Smalley, “The case for SE Android,” in *Linux Security Summit 2011*, (Santa Rosa, CA), Sept. 2011. http://selinuxproject.org/~jmorris/lss2011_slides/caseforseandroid.pdf.
- [52] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, “Protecting browsers from extension vulnerabilities,” in *Proc. of the 17th Network and Distributed System Security Symposium (NDSS 2010)*, (San Diego, CA), Feb. 2010.
- [53] L. Liu, X. Zhang, G. Yan, and S. Chen, “Chrome extensions: Threat analysis and countermeasures,” in *Proc. of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, (San Diego, CA), Feb. 2012.
- [54] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, “A survey of mobile malware in the wild,” in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM '11)*, (Chicago, Illinois), Oct. 2011.
- [55] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, “Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets,” in *Proc. of the 19th Network and Distributed System Security Symposium (NDSS 2012)*, (San Diego, CA), Feb. 2012.