AFrame: Isolating Advertisements from Mobile Applications in Android[*]

Xiao Zhang, Amit Ahlawat, and Wenliang Du
Dept. of Electrical Engineering & Computer Science, Syracuse University
Syracuse, New York, USA

ABSTRACT

Android uses a permission-based security model to restrict applications from accessing private data and privileged resources. However, the permissions are assigned at the application level, so even untrusted third-party libraries, such as advertisement, once incorporated, can share the same privileges as the entire application, leading to over-privileged problems.

We present AFrame, a developer friendly method to isolate untrusted third-party code from the host applications. The isolation achieved by AFrame covers not only the process/permission isolation, but also the display and input isolation. Our AFrame framework is implemented through a minimal change to the existing Android code base; our evaluation results demonstrate that it is effective in isolating the privileges of untrusted third-party code from applications with reasonable performance overhead.

1. INTRODUCTION

Advertising in mobile systems is cooperation between mobile advertising networks—such as Google's AdMob [4]—and application developers. Typically, mobile advertising networks distribute a Software Development Kit (SDK) library, and developers can simply incorporate one or several SDKs into their applications. Once incorporated, the SDK code will take care of the communication, advertisement refreshing, and look-and-feel customization. The advertising component is executed independently from the host application, except that they are bound together visually. When an application is installed, both the advertisement and the original application will have the same privilege, as they are running in the same process, inseparable by the system. This may cause problems.

In some situations, applications may need fewer permissions than advertisements. To allow advertisements to run, applications have to request more permissions than what they actually need, leading to over-privileged applications. In other situations, applications may need more permissions than advertisements; when users grant the permissions to the applications, they also grant the same permissions to the advertising code, leading to over-privileged advertisements. Several ideas, such as AdDroid [30] and AdSplit [33], have been proposed to solve these problems.

The integration of advertisements and applications has a special characteristic: most advertisements do not interact with their hosting applications, i.e., advertisements and applications essentially execute in mutual isolation [9, 18, 24, 31]. This special characteristic enables us to totally isolate them from host applications, i.e., running each in a separate process, and even with a different user ID. If such isolation can be achieved, we can directly use the access control system in Android and its underlying Linux to enforce privilege restriction, by assigning different permissions to different user IDs.

The isolation approach was first attempted by AdSplit, but it provides an emulated solution. In AdSplit, an advertisement and its host application are split into two different activities (and can thus be executed in two different processes), with the advertisement activity being put beneath the application activity. Using the transparency technique, AdSplit allows users to see the advertisement through the transparent region in the application activity. However, the use of the transparency technique can be problematic. First, transparency has been widely used by the ClickJacking attack [22, 28, 32, 35] and its variations [25]. Using it for every application (with advertisement) may make the attack detection and countermeasures difficult. Second, transparency poses a significant overhead in drawing, as it requires multiple layers of drawing surfaces to be combined [5]. On the other hand, AdSplit changes the current mobile advertising architecture by requiring a stub library inside each application to package up requests from advertisement activities and pass them onto their newly introduced advertisement service [33]. Such a stub library is responsible for supporting all the same APIs from the original advertising SDK, but transferring them into IPC function calls. Even though the stub library is straightforward to implement, as they argued in the paper, it is quite challenging, if not impossible, to cover all the existing APIs from the current advertising SDKs. Automated tools mitigate the challenges to some extent, but their commercial implementation will require significant testing effort and introduce new corner cases [33], which lowers the possibility of adopting AdSplit.

We would like to design a non-emulated solution without

[*]This work is supported in part by NSF grants No. 1017771, No. 1318814, and by a Google research award.

ACSAC '13 Dec. 9-13, 2013, New Orleans, Louisiana USA
Copyright 2013 ACM 978-1-4503-2015-3/13/12 ...$15.00.

using the transparency technique or changing the original advertising architecture, where advertisements and applications are placed on the same drawing surface, but they are executed in different processes. Our inspiration comes from browser's `iframe`, which allows a web page to embed another web page, and these two pages are isolated if they come from different origins. Iframe is widely adopted, because of its isolation and easy-to-use properties. We would like to create a similar "frame" in activity, allowing an activity to embed another "activity". From the user perspective, these two activities look like one because they seamlessly appear on the same window and behave like one unit. However, from the system perspective, they actually run in two different processes with different user IDs. We call such a frame *AFrame* ("Activity Frame").

AFrame achieves the process/permission isolation that is also achieved by AdSplit, but it goes beyond that by providing display and input isolation. For display isolation, each of those two activities occupies a part of the screen and one cannot tamper with the display of the other. For input isolation, the host application does not have any information about the user interaction targeted at the AFrame region and vice versa. No activity can inject an event to the other activity either. Another important advantage of AFrame is the fact that using AFrame, developers can use the advertising SDK like before, all they need to do is to tell Android to load the advertising code in a special region. In AdSplit, developers need to replace the original advertising library with their own stub library and override all the public methods. They also need to construct a standard Android IPC message in order for the stub library to communicate with the AdSplit advertising service. Although this process can be automated to some extent, such changes make it difficult to extend the AdSplit to isolate other third-party components. Using AFrame, developers can simply place advertisements and any untrusted third-party component in an AFrame region with very minimal efforts.

In this paper, we present how AFrame is designed and implemented. Although AFrame is motivated by the problems with advertisement, our design is not dependent on advertisement. AFrame is a generic solution that can be used to restrict the privilege of third-party code. In our evaluation, we demonstrate the applications of AFrame, as well as presenting its performance on Android system and applications.

2. OVERVIEW OF AFRAME

The main objective of AFrame is to isolate untrusted third-party code, such as advertising code, from its hosting application. If the code does not have a user interface (UI), the isolation can be easily achieved. What makes the problem interesting is the UI.

In Android, an application's window that interacts with users is called an *activity*, and its corresponding Java class must be a subclass of the *Activity* class. Our idea is to embed another activity inside an activity. From the user perspective, these two activities look like one because they seamlessly appear on the same window and behave like one unit. However, from the system perspective, these are two activities, running in two different processes with different user IDs. Similar to `iframe`, we call such a frame *AFrame* ("Activity Frame"), and the hosting application's frame *main frame*. We call the "activity" inside AFrame the *AFrame activity*, and the one hosting the AFrame the *main activity*. AFrame is like a typical *View* component; it occupies a rectangle area in the main activity. Inside this area runs another process, the *Aframe process*. The process that runs the main activity is called the *main process*. Figure 1 gives an example of an activity with an embedded AFrame.

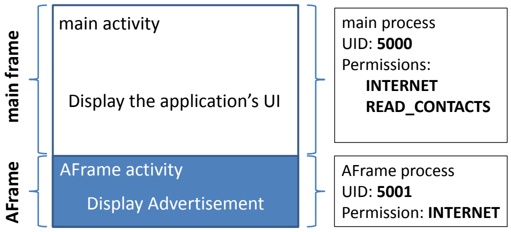


Figure 1: AFrame Example

Before discussing the design of AFrame, we would like to discuss our design objectives from three perspectives: user, developer, and system.

2.1 From the User Perspective

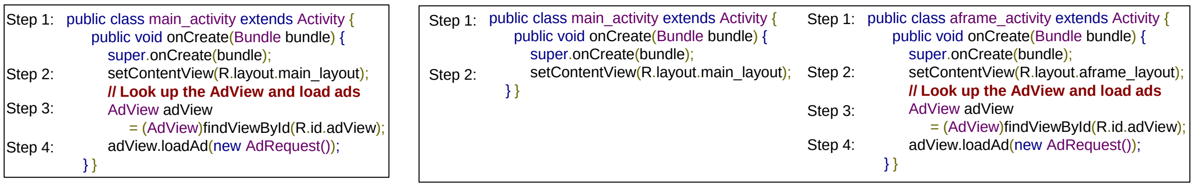
AFrame should be transparent to end users, who should feel that they are interacting with one activity, not multiple activities. This implies that the lifecycle states and the visibility of AFrame activities and the main activity should be perfectly synchronized. When one goes to the background, the other cannot stay on the foreground; when the main process is killed by user navigation or system, AFrame processes should be killed as well. It should be noted, when AFrame is killed, we do not necessarily need to kill the main activity, especially if AFrame is used to display advertisement; therefore, even if the advertisement crashes due to a bug or exception, only the AFrame process will terminate, the main activity can continue without the advertisement.

2.2 From the Developer Perspective

AFrame should also be transparent to advertising SDK developers, who may not even be aware of the existence of AFrame. No additional modifications or components are required by AFrame in the current or future SDK releases. However, to use AFrame, application developers do need to make minimal changes to their code. In this rest of this subsection, we would like to focus on how developers can use AFrame to reduce the risks caused by third-party code. We use AdMob as an example of third-party code.

In Android, to include AdMob in an application, four steps are typically involved (see Figure 2(a)). Step 1 defines an activity; Step 2 configures the UI based on a layout file specified by the developer. If there is no advertisement, Steps 1 and 2 are sufficient. To include an advertisement, two more steps are added: Step 3 finds the UI component that will be used for displaying the advertisement, and Step 4 loads the advertisement.

To use AFrame, developers need to split the original activity into two: one is the main activity, and the other is the AFrame activity. Their code is depicted in Figure 2(b). For the main activity, only the first two steps are needed, because there is no need to load the advertising code. For the AFrame activity, in addition to the essential Steps 1 and 2, it needs to load the advertisement using Steps 3 and 4. The code for the AFrame activity is quite similar to the code for the original main activity, except that the layout files used are different.



```
Step 1:  public class main_activity extends Activity {
             public void onCreate(Bundle bundle) {
                 super.onCreate(bundle);
Step 2:          setContentView(R.layout.main_layout);
                 // Look up the AdView and load ads
Step 3:          AdView adView
                     = (AdView)findViewById(R.id.adView);
Step 4:          adView.loadAd(new AdRequest());
         } }
```

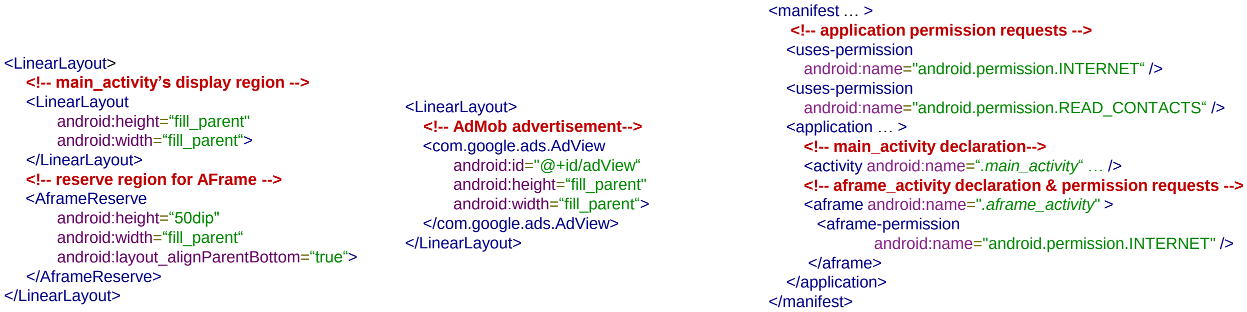
(a) Without AFrame

```
Step 1:  public class main_activity extends Activity {
             public void onCreate(Bundle bundle) {
                 super.onCreate(bundle);
Step 2:          setContentView(R.layout.main_layout);
         } }
```

```
Step 1:  public class aframe_activity extends Activity {
             public void onCreate(Bundle bundle) {
                 super.onCreate(bundle);
Step 2:          setContentView(R.layout.aframe_layout);
                 // Look up the AdView and load ads
Step 3:          AdView adView
                     = (AdView)findViewById(R.id.adView);
Step 4:          adView.loadAd(new AdRequest());
         } }
```

(b) With AFrame

Figure 2: Activity Code Without AFrame and With AFrame



```
<LinearLayout>
    <!-- main_activity's display region -->
    <LinearLayout
        android:height="fill_parent"
        android:width="fill_parent">
    </LinearLayout>
    <!-- reserve region for AFrame -->
    <AframeReserve
        android:height="50dip"
        android:width="fill_parent"
        android:layout_alignParentBottom="true">
    </AframeReserve>
</LinearLayout>
```

(a) main_activity Layout

```
<LinearLayout>
    <!-- AdMob advertisement-->
    <com.google.ads.AdView
        android:id="@+id/adView"
        android:height="fill_parent"
        android:width="fill_parent">
    </com.google.ads.AdView>
</LinearLayout>
```

(b) aframe_activity Layout

```
<manifest ... >
   <!-- application permission requests -->
  <uses-permission
    android:name="android.permission.INTERNET" />
  <uses-permission
    android:name="android.permission.READ_CONTACTS" />
  <application ... >
    <!-- main_activity declaration-->
    <activity android:name=".main_activity" ... />
    <!-- aframe_activity declaration & permission requests -->
    <aframe android:name=".aframe_activity" >
      <aframe-permission
               android:name="android.permission.INTERNET" />
     </aframe>
  </application>
</manifest>
```

(c) Application Manifest

Figure 3: Development Details of AFrame Example

The User Interface. The visual structure for user interface (UI) is called *layout* in Android. Layout elements can be declared in two ways: (1) declare UI elements in XML, and (2) instantiate layout elements at runtime. Our implementation supports both of them, i.e., defining the content of AFrame region statically in the AFrame layout file or dynamically in its activity code. We use the static case as an example in this section.

In Android, an activity has a layout file; in our design, it has two layout files: one for the main activity, and the other for the AFrame activity. In the main activity's layout file (Figure 3(a)), it reserves a place for AFrame using our introduced tag called `AframeReserve`. In Figure 3(a), the main activity reserves the bottom 50dip region for AFrame to display the advertisement.

The main activity only specifies an empty container for AFrame, without specifying any visual layout inside AFrame. The AFrame's layout is specified in a separate layout file (Figure 3(b)). Because we only put AdMob inside the AFrame, the entire layout only consists of AdView, which is the visual component used by AdMob.

The Permission Assignments. In Android, each application is granted a set of permissions during its installation. In our AFrame design, the activity running inside an AFrame has its own permission set, which can be different from the one granted to its hosting application. Application's permissions are specified by the tag `uses-permission` in the manifest file named `AndroidManifest.xml`. We introduce a new tag called `aframe`, and in this tag, we specify the configuration for an AFrame, including its associated activity and the permissions granted to it. In the example depicted in Figure 3(c), the AFrame is only granted the *INTERNET* permission, while the application has the *INTERNET* and *READ_CONTACTS* permissions.

2.3 From the System Perspective

The main objective of our design is to isolate AFrame activities from the main activity. The isolation objective consists of four concrete goals: process isolation, privilege isolation, display isolation, and input isolation.

The first goal is *process isolation*. The AFrame activity and the main activity will run in two different processes with different user IDs. This is a strong isolation, and if it can be achieved, we can leverage the underlining operating system (Android uses Linux) to protect each activity's memory space, data, files, and other resources from one another. Even if the untrusted third-party code has malicious native code, the isolation will not be broken, unless the third-party code can root the phone.

The second goal is *permission isolation*. Android's permission system is based on user ID, i.e., permissions are assigned to individual user IDs, each representing an application. Because AFrame runs as a separate process with a different user ID, we can naturally use Android's permission system to give the AFrame activity permissions that are different from those given to the application.

The third goal is *input isolation*. When users interact with an AFrame activity, the main activity should not be able to observe this interaction, and vice versa. The interaction includes the inputs occurred in the AFrame or the main frame regions, such as touch events and key strokes [1]. Moreover, AFrame activity and main activity should not be able to forge an event to one another (they can still do that to themselves).

The last goal is *display (output) isolation*. Although AFrame activities and the main activity share the same screen, and

[1] Other types of inputs, such as voice, camera, or sensor inputs, are not covered by AFrame if they are not directly related to a particular region. Side channels are not covered by AFrame either.

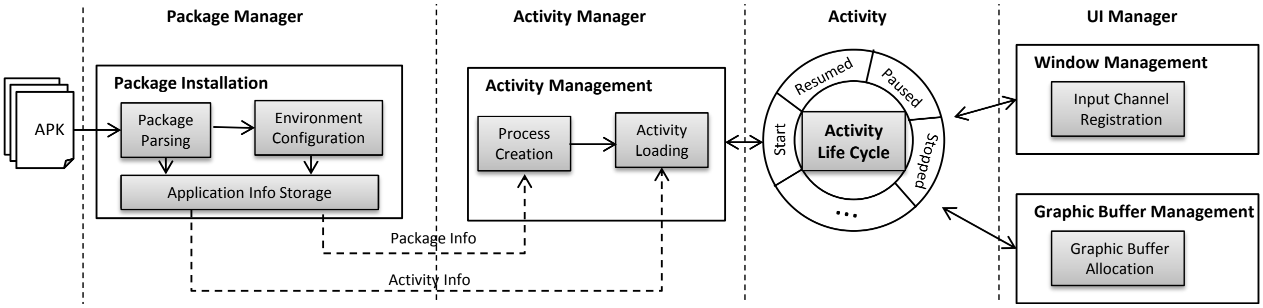


Figure 4: Activity Creation and Execution

behave like one activity, they have their own display regions. One cannot tamper with the contents displayed in another activity's region, i.e., their drawings should be restricted to their own regions on the screen.

3. DESIGN AND IMPLEMENTATION

To achieve the above goals in Android, we need to understand how exactly an activity is created, and how it interacts with the rest of the system. Based on this understanding, we identified several places where changes need to be made to support AFrame. Figure 4 depicts the process of an activity from application installation, activity launch, and finally to its standard lifecycle and interaction with the system. The gray boxes in Figure 4 are the major components that we have modified to support AFrame. In this section, we drill down to these components, and describe how AFrame is designed and implemented to achieve the four isolation goals.

3.1 Process Isolation

The objective of process isolation is to run the AFrame activity in a different process with a different user ID, so the AFrame activity is physically separated from the application's main activity. Compared to the original Android code, we need to create a new user, a new process and a new activity for the AFrame region.

Application Installation and Setup. The information about AFrame needs to be retrieved by Android when an application is installed. During installation, Android parses the application's manifest file, and retrieves the application's component information, including activities, services, broadcast receivers, and content providers. Android also creates a new user for this new application, as well as creating a private data folder for its resource storage. This is done by the Package Manager Service (PMS).

In our design, we add a new parsing module in PMS, which parses the <aframe> tag defined in the manifest file to retrieve AFrame information, including its activity class name and the permissions assigned to the AFrame. Based on the information, the modified PMS will create an additional user for the AFrame, and set up the private data folder. The procedure conducted for the AFrame is exactly the same as that for the application.

Process Creation. When an application is launched, a process needs to be created to run the application. In Android, this process creation is initiated by the Activity Manager Service (AMS). To start a process, AMS first requests the process information from PMS, including the process's user ID (UID), group ID (GID), the groups that the UID belongs to (GIDs), and the data folder (see the left part of Figure 4). After getting the information, AMS sends a process-creation request to the Zygote process, which forks a new process, and configures the new process using the process information.

To support AFrame, we slightly modified the above procedure. In addition to creating a new process for the application, AMS also retrieves the AFrame process information from PMS, if the application contains an AFrame. It then sends an additional request to Zygote, requesting it to create a new process for the AFrame. The main process and AFrame process are started simultaneously.

Activity Loading. After the process is created, Android needs to load the corresponding activity into the process, and bootstrap the activity's lifecycle, so the application can interact with users and the system. In our design, in addition to loading the application's activity to its process (already done by Android), we need to load the AFrame activity as well. Process record, activity record and runtime context are three essential resources for activity loading and for activities to run properly. The detailed explanation of these three resources can be found in appendix A. After all these types of resources are prepared, Android loads the activity under the control of AMS, following the steps depicted in Figure 5.

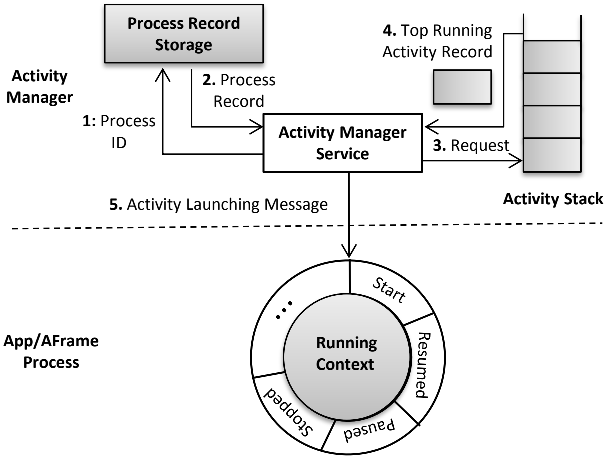


Figure 5: Android Activity Loading

We leverage the current activity implementation as much as possible to construct the above process record, activity record, and runtime context for the AFrame activity. Following this principle, we first copy all the values of the main activity, and then manually identify all the necessary places where the AFrame activity should have different values. For example, in the AFrame process record, we need to change the process name and ID. Since the AFrame component still belongs to the application, we do not need to change its application information. Similarly, for the AFrame activity record, we change the process name, data folder and its hosting process record. However, for the state of the current activity, we can keep the value since the AFrame activity always shares the same state with the main activity. Moreover, we need to create and register new communication with system services. For example, AMS needs to communicate with the AFrame activity to control the transition of its states; Window Manager Service (WMS) needs to communicate with it to set the visibility of its display.

Once the activity record is constructed, AMS constructs an activity launching message with the activity record as the argument and delivers it to the hosting process using the activity token. The UI thread in the hosting process responds to the request by initializing the context of the current activity, loading the activity class and triggering its lifecycle. For AFrame, we construct another activity launching message based on the AFrame activity record, and deliver it to the AFrame process.

3.2 Permission Isolation

In Android, each application is assigned a unique user ID (UID) at the installation time, and each UID is associated with a set of permissions. At the runtime, Android uses the UID to find out the permissions of an application, and conduct access control based on the permissions. Since the AFrame activity's UID is different from the main activity's UID, their permissions are naturally isolated from each other. There is no need to change the access control logic of Android to support AFrame. We have only changed 10 lines of code for permission isolation, but none is on the access control logic. Still, a detailed explanation on Android's access control model is available in appendix B. At the time the application is installed, a combination of the requested permissions from main process and AFrame process will be prompted to the user for consent.

3.3 Display Isolation

We describe how the AFrame activity and the main activity can share the same screen, while not being able to tamper with each other's display region. Before that, it is essential to explain how drawing works in Android.

(A) How Drawing Works in Android

Graphic Buffer Allocation. The drawing memory required by applications is managed by a system process called *SurfaceFlinger*. To be able to draw, an application first needs to request a buffer from SurfaceFlinger that interacts with the hardware abstraction layer (HAL).

Android has two types of HAL: gralloc is used for emulator and framebuffer is used for real devices. For emulator, it uses double buffering, so HAL allocates a screen-sized graphic buffer. This acts as an off screen image and is called back buffer. The back buffer is used for drawing. When the drawing completes, this buffer is copied using block line transfer to the screen surface, called primary surface. Double buffering is used to eliminate visible draws.

On real devices, page flipping is used to eliminate tearing. In this case, HAL uses twice the screen size to allocate memory. The purpose is to use one as a back buffer and the other as the primary surface. The drawing is done on the back buffer, while the contents of the primary surface are used to display current screen contents. When the drawing is complete on the back buffer, page flip is conducted, so the back buffer becomes the primary surface, and the old primary surface is now used as the back buffer.

For both types of HAL, hardware graphics operations need physical continuous memory to manipulate. SurfaceFlinger pre-allocates a fix-sized (8M, by default) chunk of memory using Process MEMory allocator (PMEM). PMEM is typically used to manage large (1-16+MB) physically contiguous regions of memory shared between userspace and kernel drivers [7]. Upon graphic buffer requests from the application, SurfaceFlinger chooses the next available block of memory and sends the specification details back. The application takes advantage of the file descriptor, base address, offset and block size information inside the specification, and maps the same block of memory to its own process. The application process further packages the mapped buffer into a Java object, called *Canvas*, and provides drawing APIs to applications through this Java object. Due to page limit, details regarding drawing on the *Canvas* object can be found in appendix C.

Display Rendering. After the activity finishes its drawing, it informs SurfaceFlinger to do the rendering. This time, SurfaceFlinger functions like a window compositor and can combine 2D/3D surfaces from multiple applications. In Android, each activity window corresponds to a layer, and the layers are sorted in Z-order. SurfaceFlinger renders the display in two phases:

- *Visible Region Calculation:* From top to bottom, each layer takes its height and width as the initial visible region, and then subtract the region covered by the layers on top of itself.
- *Layer Composition:* From bottom to top, SurfaceFlinger copies the data inside each layer's visible region to the main surface. Main surface is a frame buffer specific for holding the layer composition result.

As Android commonly uses the standard Linux frame buffer device, HAL uses the frame buffer to generate the final image, which is sent to the frame buffer driver in the Linux kernel for displaying.

(B) Display Isolation

The AFrame activity and the main activity must share the same screen, and at the same time, their drawings should be restricted to their own regions. The combination of sharing and isolation makes the design for this part very challenging. We have considered two design choices: achieve sharing in graphic buffer allocation but isolation in canvas drawing (soft isolation), or achieve isolation in graphic buffer allocation but sharing in display rendering (hard isolation).

Soft Isolation. In this design, after SurfaceFlinger prepares a graphic buffer, it maps the same buffer memory to both the main process and the AFrame process. Essentially, both activity processes share the same buffer, and can freely draw anywhere on the screen. We need to restrict their drawings to their own regions. There are two ways to access the graphic buffer. One is through the direct memory access using native code. If the activity inside AFrame does not have its own native code or the native code brought by it is blocked, this path is blocked. The other is through the standard Canvas APIs to draw some objects in the buffer. These APIs implement a clipping mechanism to ensure that the drawing by each node on the view tree can only affect the region assigned to that node, and nothing beyond. Therefore, all we need to do is to set up the clipping region correctly for the main activity and the AFrame activity, so their drawings using the Canvas APIs are always restricted to their own designated regions.

The lightweight soft isolation comes with a cost of security, since it depends on the unlikely assumption that there is no native code execution in android advertisements. To make AFrame more secure, we choose the following hard isolation option in our implementation.

Hard Isolation. In this design, the main process and the AFrame process do not share a single drawing buffer any more. Instead, each process gets a unique graphic buffer from system, and maps that block of memory to its own process space for drawing. As a result of that, the memory space of these two processes are totally isolated and accessing the other process's drawing memory will result in a hardware exception.

After each activity finishes its drawing, two layers—main layer and AFrame layer—are used by SurfaceFlinger for display rendering. In order for SurfaceFlinger to distinguish the main layer from other layers, we explicitly send its `AframeReserve` region information to SurfaceFlinger before the graphic buffer allocation request. During the *Visible Region Calculation* phase, we identify a layer as the main layer if SurfaceFlinger contains its `AframeReserve` information. In such case, we cut `AframeReserve` region from its visible region. As a result of that, the AFrame layer below is able to keep the `AframeReserve` region as its visible region following the original *Visible Region Calculation* phase.

When SurfaceFlinger finishes the layer composition, the final image will be a combination of the drawing from main activity and AFrame activity, each activity only contributing to its designated region. Since the composition happens in SurfaceFlinger process, which is a privileged system process, there is no way for applications to tamper the designated logic, even with the native code considered.

3.4 Input Isolation

Events should be considered as application's private resources, because they affect application behavior. Normally, events are generated by user interactions, such as clicking, touching, and key strokes, but Android also allows applications to generate events programmatically [26]. While events injection to an application itself is always allowed, injecting events to another application needs the *INJECT_EVENTS* permission, which is a system-level permission that is never granted to normal applications.

How Event Dispatching Works in Android. When a new activity is started, it sends a request to the WindowManager system service to register an input channel with the system. This channel will be used by the system to send events to the activity process. WindowManager forwards the request to InputManager, which responds to the request and establishes the input channel with the new activity. WindowManager manages the z-order of activity windows, and synchronizes this information with InputManager. This way, they both know which activity is on the top and is currently receiving focus from user interaction. When InputManager receives an event from the device driver or applications, after checking the *INJECT_EVENTS* permission, it delivers the event to the activity on the top through the established input channel with the activity. Once the activity receives the event, it further dispatches the event down its view tree until some view object consumes the event. The event will be discarded if no one can handle it.

Input Isolation. The original Android system assumes that only one activity is on the top and receives input focus. With AFrame, this is not true anymore, so we need to make corresponding changes to the system. We extend the original input-channel registration process, so InputManager also knows which display region belongs to the main activity or the AFrame activity. We also add an additional *Decision Maker* module to InputManager to enforce the isolation, as shown in Figure 6.

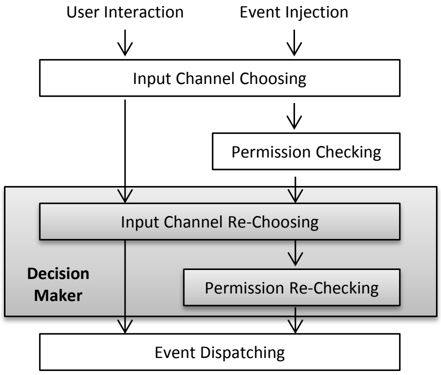
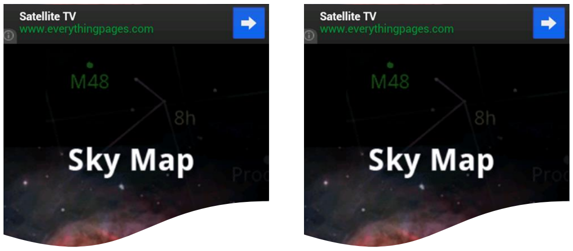


Figure 6: Android's Event Dispatching System

When InputManager receives an input event from the system, it first decides which window is on the top, and then decides which activity (main or AFrame) should get the input. Because InputManager knows the region information and the event location, it can choose the correct input channel and dispatch the event to the intended activity process.

The above change is not sufficient. In Android, an activity can generate a user event; this event will be sent to InputManager, which will identify the target input channel, check the *INJECT_EVENTS* permission and then dispatch it. Our input channel re-choosing mechanism may change the target channel, and therefore, violate the input isolation. For example, the main activity can inject an event to itself but target the AFrame activity region. This event injection will be allowed since it targets at its own process. However, the input channel re-choosing mechanism redirects the event to another input channel, which is in the AFrame process, resulting in a violation of input isolation.

To achieve input isolation, we add an additional access control in InputManager. Before the events are actually dispatched to the targeted input channel, we check again whether the UID of the sender process is the same as the UID of the target process. If these two UIDs are the same, event dispatching is allowed; therefore, like the original An-



(a) With AFrame (b) Without AFrame

Figure 7: Sky Map with Advertisement

droid, an activity can generate an event for itself. However, if the two UIDs are not the same, the event will not be dispatched, unless the event generator has the system-level *INJECT_EVENTS* permission.

3.5 LifeCycle Synchronization

AFrame activity and main activity are running in different processes, but they have to function like one activity; namely, they have to "stick" together in terms of visibility, behavior and existence.

The activity lifecycle begins with instantiation, ends with destruction, and includes many states in between. All the state transitions of activities within an application are managed by AMS. When an activity changes its state, the appropriate lifecycle event method is called, notifying the activity of the impending state change and allowing it to execute developer's code in order to adapt to that change. After the activity finishes its transition, it notifies the AMS, so the system can react to the activity's new state.

Synchronization between an AFrame activity and its host activity requires additional communication between AMS and the application. Whenever AMS receives the notification from the main activity, it also notifies the AFrame activity of the same impending state change, so the AFrame activity can go to the same state transition.

4. EVALUATION AND CASE STUDIES

To evaluate AFrame, we focus on two aspects: effectiveness and performance. For effectiveness, we use advertisement as a case study to demonstrate how AFrame can be effectively used to isolate the privilege of untrusted components from that of the main application. For performance, we measure the system-level and application-level overheads caused by AFrame.

4.1 Isolating Third-Party Advertisements

In this experiment, we show how AFrame can protect applications against untrusted third-party advertisements. Because of the need to modify the existing applications, we use an open-source application, Sky Map [6], as the hosting application. According to the statistics on Google Play, this very popular app has been downloaded for more than 10 million times. Donated by Google, Sky Map does not include any advertisement. For our demonstration purpose, we manually modified its source code and incorporated the AdMob advertising SDK. Figure 7 shows what the application looks like. From the figure, we can see that there is no visual difference between the one using AFrame and the one without AFrame, i.e., AFrame is transparent to users.

Privilege Isolation. Sky Map requires the following permissions: *WRITE_EXTERNAL_STORAGE*, *INTERNET*, *READ_PHONE_STATE*, *WRITE_SETTING*, *WAKE_LOCK*, *ACCESS_NETWORK_STATE*, *ACCESS_FINE_LOCATION*, and *READ_EXTERNAL_STORAGE*. Several of these permissions are related to user's private data. Once granted, they can be used by any component of the application, including the advertisement. Therefore, the third-party advertisement can collect the user's location data, phone state, and read the data stored on the SDCard.

With AFrame, we can easily limit the privilege of the advertising component, without affecting the original application. To achieve that, in the Sky Map's manifest file, we declare the advertisement activity in an `aframe` tag. We assign only two permissions—required by AdMob—to this AFrame region, including *ACCESS_NETWORK_STATE* and *INTERNET*. The detailed specification is shown below:

```
<aframe android:name=".activities.AdsActivity">
  <aframe-permission
      android:name="android.permission.INTERNET"/>
  <aframe-permission
      android:name="android.permission.ACCESS_NETWORK_STATE"/>
</aframe>
```

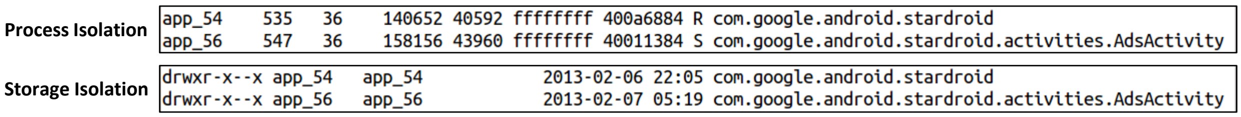
As Figure 8 shows, when the advertisement activity is started, it is loaded into a different process with a different user ID (see Figure 8). Moreover, the AFrame activity and the main activity have their own data folders, and the permissions on these folders ensure that one activity cannot access the other activity's data.

To further demonstrate the privilege restriction on the advertisement process, we intentionally add a piece of malicious code in the advertisement activity and try to access the user's location. As Sky Map runs, we receive a security exception from the advertisement process (Figure 9), indicating that the developer has not declared the *ACCESS_FINE_LOCATION* or *ACCESS_COARSE_LOCATION* permission. Without using AFrame, this access will be successful, as the Sky Map app has these permissions.

We have handled the above exception in the advertisement activity code, so the Sky Map application does not crash and can continue to run smoothly. If we remove the exception handling code, the AFrame process will crash. Because AFrame runs in a different process, the main process does not need to terminate when the AFrame process crashes. Whether the application should be required to terminate or not depends on our synchronization policy set by the developer. It should be noted that without using AFrame, if the advertisement crashes, the entire application will crash.

Compatibility with various advertising libraries. We repeat our above experiment with other advertising libraries. Five popular advertising libraries are selected based on the statistics provided by the third-party Android market AppBrain [3]. They all work with AFrame, and the user experiences are similar to that of AdMob. Due to page limitation, we do not include screenshots for this evaluation. Table 1 summarizes our results.

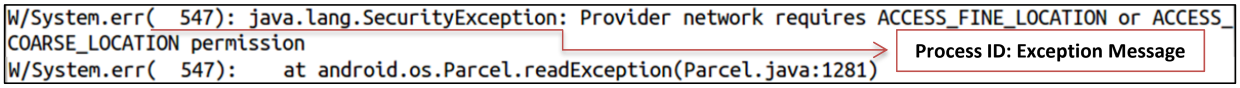
Other Applications. We further test AFrame with other open-source applications, including Easy Random Numbers, Easy Graphic Paper and Skillful Lines [14]. All of them were released in 2012, with the AdMob library already added. Each of the selected applications was modified to move the AdMob code into AFrame. This involves adding a new activity for AFrame, reserving space for AFrame in the original activity layout file, moving the layout for advertisement to



```
app_54     535   36    140652 40592 ffffffff 400a6884 R com.google.android.stardroid
app_56     547   36    158156 43960 ffffffff 40011384 S com.google.android.stardroid.activities.AdsActivity
```

```
drwxr-x--x app_54   app_54           2013-02-06 22:05 com.google.android.stardroid
drwxr-x--x app_56   app_56           2013-02-07 05:19 com.google.android.stardroid.activities.AdsActivity
```

Figure 8: Process and folder isolation between main activity and AFrame activity with different UIDs



```
W/System.err(  547): java.lang.SecurityException: Provider network requires ACCESS_FINE_LOCATION or ACCESS_
COARSE_LOCATION permission
W/System.err(  547):    at android.os.Parcel.readException(Parcel.java:1281)
```

Figure 9: Security exception caused by not having the location permissions in the AFrame process

	% of apps	% of installs	Compatibility
AdMob	32.34	44.17	Yes
Millenial Media	3.73	16.67	Yes
TapJoy	1.24	11.85	Yes
InMobi	2.62	12.54	Yes
Greystripe	0.39	1.38	Yes
Mopub	0.71	5.15	Yes

Table 1: AFrame Compatibility with Ad Libraries

the AFrame layout file, and specifying permissions for the AFrame in the manifest file. This modification is quite simple, and takes about 10 minutes for each activity. All applications run successfully on Android, and the observations are similar to those obtained from our Sky Map experiment.

4.2 Performance: System Overhead

To evaluate the performance of AFrame, we use the following environment setup: Our source code is based on Ice Cream Sandwich(ics): *android-4.0.3_r1.2*, and we run our modified Android on Samsung Galaxy Nexus I9250 with the following specifications: CPU (Dual-core 1.2 GHz Cortex-A9), RAM (1 GB), Display (4.65, 720*1280), GPU (PowerVR SGX540), Internal Storage (16 GB), and no SDCard.

To measure the overhead imposed on the system, we use a popular Android benchmarking tool: AnTuTu [2]. It runs a series of tests and provides a score report on various metrics, including performance on memory access, integer & floating-point operations, graphics, database I/O, and SDCard access (read/write). The higher the score is, the better. The results (average scores of multiple runs) are provided in Table 2. Since our testing phone does not have an SDCard and AFrame does not have any modification that affects SDCard, we remove the SDCard score from the final results.

AuTuTu	Original SDK	AFrame SDK	Difference
Total Score	7302.90	7250.00	-0.71%
Memory Access	1413.15	1410.35	-0.20%
Integer Operation	1697.30	1689.90	-0.44%
Float Operation	1292.25	1286.80	-0.42%
2-D Graphic	460.20	451.80	-1.83%
3-D Graphic	1860.25	1833.15	-1.46%
Database Operation	312.00	311.75	-0.08%

Table 2: Benchmark Scores

From the AnTuTu benchmark, it can be observed that AFrame imposes no significant overhead on memory accesses, CPU, and database. It does impose a slight overhead on 2D graphics (1.83%) and 3D graphics (1.46%). AFrame imposes these overheads because it performs additional checks before a canvas is created for drawing and AFrame also performs checks for layer composition before the final image is sent to the frame buffer driver in Linux kernel for displaying.

4.3 Performance: Application Overhead

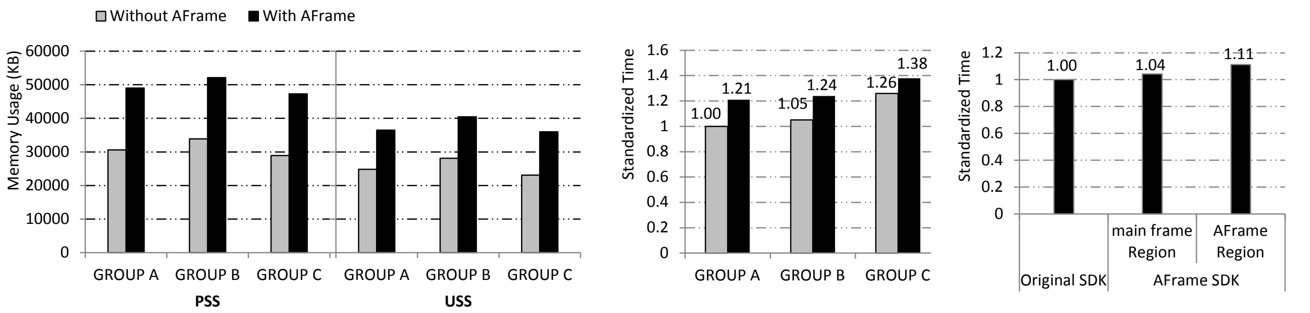
We also need to evaluate the performance impact of AFrame on applications. We measure the following: memory consumption, time to start the application, and event dispatching. To conduct the tests, we wrote six testing applications, which are divided into three groups. Each group has two applications with identical configuration, except that one uses AFrame and the other does not. The detailed specifications are shown in Table 3.

Groups	In-Group Difference	AdMob	WebView	Webpage
A	No AFrame	No	Yes	Blank
	With AFrame	No	Yes	Blank
B	No AFrame	No	Yes	Google
	With AFrame	No	Yes	Google
C	No AFrame	Yes	No	N/A
	With AFrame	Yes	No	N/A

Table 3: Sample Applications for Overhead Evaluation

Memory Overhead. To measure the memory overhead we use a tool called procrank [1]. This tool measures the memory usage by an application in terms of PSS (Proportional Set Size) and USS (Unique Set Size). PSS divides the size of shared memory equally among all the processes that share it, and count each process's contribution to the overall memory. The other measure, USS, only counts the amount of memory used uniquely by a process. For applications containing AFrame, the memory usage is the sum of the usage of the application process and the AFrame process. We first run each application individually and our measurements are depicted in Figure 10(a).

We do expect AFrame to increase the memory consumption quite significantly, as it adds an additional process to each activity. This is confirmed by our results in Table 4, as AFrame adds about 59% and 49% memory overhead to PSS and VSS, respectively. However, in real world, it is quite common that multiple applications are alive at the same time, with most of them running in the background. To better emulate our daily smartphone usage, we simultaneously run the AFrame application from each group in background and the normal application from the same group in front. Due to the memory sharing mechanism on android,



(a) Memory Overhead (Individual Run Case) (b) Time-to-start Overhead (c) Event Dispatching Overhead

Figure 10: Various Overhead Introduced by AFrame

the memory overhead to PSS dramatically drops to 27%, which is only less than half of the previous overhead. Even though VSS does not count the shared memory, the corresponding overhead still decreases slightly to 45%.

	Group Index	Pss Overhead	Uss Overhead
Individual Run	A	60.00%	47.00%
	B	53.00%	43.00%
	C	63.00%	56.00%
Average		**59.00%**	**49.00%**
Group Run	A	27.00%	43.00%
	B	25.00%	46.00%
	C	30.00%	47.00%
Average		**27.00%**	**45.00%**

Table 4: Memory Overhead Comparison

It should be mentioned that we have not exploited any optimization in our current implementation. We believe that the memory overhead can be further reduced using optimization. For example, if an application has N activities, each running an advertisement in its AFrame, instead of creating N additional processes, we can host all these advertisements in a single AFrame process. We can also reduce the heap size of the AFrame process, etc.

TTS Overhead. When an application uses AFrame, the application startup procedure involves checking the presence of AFrame component and starting an additional process. We evaluate how much AFrame affects application's startup time (TTS stands for "Time To Start"). We record the system time when launching an application in ActivityManagerService.java, before the process fork arguments are sent to Zygote via Process.start(). We also record the startup ending time in reportActivityLaunchedLocked() from ActivityStack.java. The startup time for the six sample applications are shown in Figure 10(b). The results are standardized based on the startup time of the application with WebView loading a blank page. As it shows, if AFrame is used to isolate AdMob library, the TTS overhead is only 9%. However, if we use AFrame for other purposes, such as isolating WebView, the TTS overhead will be a little bit higher. In the WebView case, the overhead will reach 20% (18% for loading blank page and 21% for loading Google), but the absolute time is still acceptable, because it is similar to the time for loading an normal advertisement.

Event Dispatching Overhead. When an application uses an AFrame component, when dispatching events to the application, the system will conduct additional checks to ensure that events for the main frame are not sent to AFrame, and vice versa. We measure the overhead of this checking by calculating the difference in time taken to dispatch a single event. We wrote two sample applications for this purpose. The first application contains an activity with a single button, and it is executed in the unmodified Android system. The second application contains an activity with two buttons, one in the main activity, and the other in AFrame. This application is executed in our modified system. The evaluation measures how long it takes a touch event to be dispatched to its corresponding button. Figure 10(c) depicts the evaluation results, which show that the overhead for the main frame is quite small, and the overhead for the AFrame is about 11%. If we use AFrame to display advertisement, this overhead does not have much effect, as users do not interact with the advertisement very often [9, 18, 24, 31].

5. RELATED WORK

Restricting advertisement. Two existing proposals specifically focus on restricting the privilege of advertising code. AdDroid [30] encapsulates the advertising libraries into the Android framework to lift their trust level. To separate privileged advertisement functionality from host applications, AdDroid introduces a new advertising API and corresponding advertising permissions, allowing applications to show advertisements without requesting privacy-sensitive permissions. AdDroid changes the way developers use the advertising SDK. This kind of changes is not necessary if developers use AFrame. Another proposal is AdSplit [33], which puts the advertisement into a separate activity, with the advertisement activity being put beneath the application activity. We have compared with AdSplit in the Introduction section, and will not repeat the comparison here.

Privilege Restriction. Another line of research that is related to AFrame is the work on restricting application's privilege. Apex [27] allows a user to selectively grant permissions to applications during the installation. Saint [29] governs install-time permission assignment and their runtime use as dictated by application provider policy. Stowaway [19] determines the set of API calls that an application uses, maps those API calls to permissions, and determines whether the application is over-privileged.

Privilege escalation is an important problem in mobile systems. Several ideas have been proposed to defeat privilege-escalation attacks, including XMandDroid [10], PScount [8], DroidChecker [11], WoodPecker [21], and the work developed by Felt et al. [20].

Security Extensions. Several security extensions have been proposed to improve the Android security architecture. Kirin [17] performs lightweight app certification to prevent malware at the installation time. DroidRanger [37] applies both static and dynamic analysis to build a permission-based behavioral footprinting scheme to detect new samples of known Android malware families. It also uses a heuristics-based filtering scheme to identify certain inherent behaviors of unknown malicious families. While ComDroid [12] detects application communication vulnerabilities, DroidMOSS [36] tries to detect the app-repackaging behavior in third-party Android markets. QUIRE [15] tracks the call chain of IPCs and enforces security policies. CRePE [13] enforces fine-grained policies based on the context of the smartphone.

Privacy Protection. Another line of research on smartphone security is devoted to protecting users' private information. Several systems have been developed, including TaintDroid [16], AppFence [23], Aurasium [34], etc. This work is orthogonal to our work on AFrame, and they can be applied together with AFrame to further restrict the privilege of third-party libraries.

6. CONCLUSIONS

To totally isolate untrusted third-party libraries, such as advertisement, from their hosting applications, we propose AFrame, a developer friendly framework to achieve the necessary process isolation, permission isolation, and input/output isolation. With AFrame, untrusted third-party libraries can be isolated into a different process with a different UID. Moreover, developers still have the capability to configure the permissions they request. AFrame also ensures the integrity of each other's display and input resources. We have conducted case studies on advertising libraries and WebView component to demonstrate AFrame's effectiveness. We also measure its performance to show that the overhead is acceptable. The results indicate that AFrame is a viable solution to solve the over-privileged problem associated with third-party libraries. In our future work, we will further optimize the performance, as well as exploiting various security solutions, such as AppArmor and SELinux to further restrict the behavior of the AFrame process.

7. REFERENCES

[1] Android memory usage. http://elinux.org/Android_Memory_Usage.

[2] Antutu benchmark. https://play.google.com/store/apps/details?id=com.antutu.ABenchMark.

[3] Appbrain. http://www.appbrain.com/.

[4] Google AdMob Ads SDK. https://developers.google.com/mobile-ads-sdk/docs/.

[5] The proper use of view.setalpha(). https://plus.google.com/113735310430199015092/posts/NSgQvbfXGQN.

[6] Skymap. https://play.google.com/store/apps/details?id=com.google.android.stardroid&hl=en.

[7] Android kernel features. http://elinux.org/Android_Kernel_Features, 2011.

[8] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[9] D. BELIC. Three things to know about mobile advertising in 2011. http://www.intomobile.com/2012/02/28/3-things-know-mobile-advertising-2011-infographic/.

[10] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Technische UniversitÃČÂd't Darmstadt, 2011.

[11] P. P. Chan, L. C. Hui, and S. M. Yiu. Droidchecker: analyzing android applications for capability. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, 2012.

[12] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, 2011.

[13] M. Conti, V. T. N. Nguyen, and B. Crispo. Crepe: context-related policy enforcement for android. In *Proceedings of the 13th international conference on Information security*, 2011.

[14] H. Davis. Open source android applications. http://www.hunterdavis.com/android-apps/.

[15] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX conference on Security symposium*, 2011.

[16] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, 2010.

[17] W. Enck, M. Ongtang, and P. McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[18] A. Farahat and M. C. Bailey. How effective is targeted advertising? In *Proceedings of the 21st international conference on World Wide Web*, WWW '10, 2012.

[19] 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*, 2011.

[20] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: attacks and defenses. In *Proceedings of the 20th USENIX conference on Security symposium*, 2011.

[21] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, 2012.

[22] R. Hansen. Clickjacking. http://ha.ckers.org/blog/20080915/clickjacking/.

[23] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, 2011.

[24] P. Kazienko and M. Adamski. Adrosa Íc adaptive personalization of web advertising. *Information Sciences*, 177:2269–2295, 2007.

[25] T. Luo, X. Jin, A. Ananthanarayanan, and W. Du. Touchjacking Attacks on Web in Android, iOS, and Windows Phone. In *Proceedings of the 5th*

International Symposium on Foundations & Practice of Security, October 25–26 2012.

[26] R. Motisan. Programmatically injecting events on android. http://www.pocketmagic.net/2012/04/injecting-events-programatically-on-android/#.USRs4aU4vlK, 2012.

[27] 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*, 2010.

[28] M. Niemietz. UI Redressing: Attacks and Countermeasures Revisited. CONFidence 2011.

[29] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, 2009.

[30] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*, 2012.

[31] T. R. Report. Mobile ad click-through rate twice as high for iphone, ipad users. http://therealtimereport.com/2011/07/08/mobile-ad-click-through-rate-twice\-as-high-for-iphone-ipad-users/.

[32] J. Ruderman. Bug 154957 - iframe content background defaults to transparent. https://bugzilla.mozilla.org/show_bug.cgi?id=154957.

[33] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the 21st USENIX conference on Security symposium*, 2012.

[34] R. Xu, H. Saïdi, and R. Anderson. Aurasium: practical policy enforcement for android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, 2012.

[35] M. Zalewski. Browser security handbook. http://code.google.com/p/browsersec/wiki/Part2.

[36] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Droidmoss: Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, 2012.

[37] 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 *Proceedings of 19th Annual Network & Distributed System Security Symposium*, 2012.

APPENDIX

A. ACTIVITY LOADING PREREQUISITES

The following three types of resources are essential for activity loading and for activities to run properly.

Process Record: All about the process that is going to load the activity. It includes the battery status, application token for system to communicate with the activity, application information, process name, process ID, components history record and many other variables. After a process is created in AMS, its record will be registered to AMS for future look-ups.

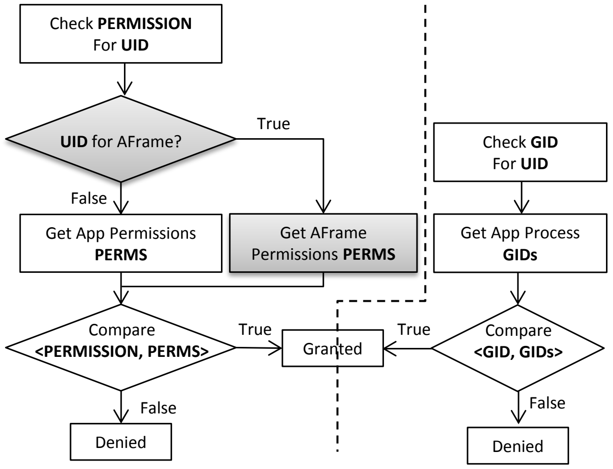
Activity Record: An entry in the history stack, representing the activity that is about to run. It is placed on ActivityStack for management. The one that we are going to load is always on the top of the stack. An activity record contains the activity's package name, component information, data directory, communication tokens with system services, an ActivityStack instance, an AMS instance, and many other types of information.

Runtime Context: An interface to global information about an application environment. It provides services for resolving resources, for accessing database and preferences, etc. Once loaded, the hosting process will initialize the running context of the activity and attach it to the activity.

First of all, AMS looks up the host process record, then it requests the top running activity record from ActivityStack. Finally, AMS communicates with the host process via the application token defined in the process record, and ask the process to load the specified activity. The host process responds to the request by loading the activity class and initializing its running context. At this point, the activity becomes "alive", and enters its standard lifecycle.

B. ANDROID ACCESS CONTROL MODEL

Android access control logic (i.e., reference monitor) are placed in two different places: in framework and kernel. Access to some resources, such as CONTACTS, NFC (Near Field Communication), and SMS, are enforced by the framework Reference Monitor (Figure 11(a)), and access to other resources, such as INTERNET, BLUETOOTH, and GPS, are enforced in the Kernel Reference Monitor using GIDs (Figure 11(b)). From the figure, we can see that Android's access control is based on UID, so if two processes have different UIDs, their privilege can be different.



(a) system resource access (b) Kernel resource access

Figure 11: Android's access control system

Framework Reference Monitor: It is in system_server process and consists of two services: Activity Manager Service(AMS) and Package Manager Service(PMS). The eventual permission checking is done in CheckUIDPermission() call in PMS. This API takes a UID and a requested permission as arguments and checks whether the requested permission is inside the permission set granted to the UID. If yes, the access is granted. Otherwise, it will be denied.

Kernel Reference Monitor: When application tries to access some kernel resources, like socket or Bluetooth, Android also needs to check it has the corresponding permissions. One way to do it is ask the PMS and AMS to push all the applications' permission information down to the kernel. However, this will introduce significant storage and performance overhead to the kernel. Android takes another approach by using the GIDs of a process to make the decision. This is more like a capability-based access control system. For example, if an application requests INTERNET permission and user grants the request, Android will assign the inet GID to the app's GID group. When the app tries to use INTERNET, kernel simply checks whether the app has the inet group ID. If it has, the app can get the socket and use the Internet. Otherwise, the request will be denied.

C. CANVAS DRAWING

The application process packages the mapped buffer into a Java object, called *Canvas*, and provides drawing APIs to applications through this Java object. When an activity is launched, the application process parses the current activity's layout file and constructs its view tree. When an area on the screen needs to be redrawn, the area is identified as the invalid region; the activity traverses the entire view tree and redraws any view node that intersects with the invalid region. Each node on the view tree represents a region with a specific size and at a specific location on the screen. Each node is given a canvas with a clipping boundary and a translation matrix. Through this matrix and clipping boundary, each node is restricted to draw inside its own region; anything drawn outside of the region will be clipped away.