FLEXDROID: Enforcing In-App Privilege Separation in Android

Jaebaek Seo[*], Daehyeok Kim[*], Donghyun Cho[*], Taesoo Kim[†], Insik Shin[*‡]
[*]School of Computing, KAIST
[†]College of Computing, Georgia Institute of Technology
{jaebaek, dhkim7, bigeyeguy, ishin}@kaist.ac.kr, taesoo@gatech.edu

***Abstract*—Mobile applications are increasingly integrating third-party libraries to provide various features, such as advertising, analytics, social networking, and more. Unfortunately, such integration with third-party libraries comes with the cost of potential privacy violations of users, because Android always grants a full set of permissions to third-party libraries as their host applications. Unintended accesses to users' private data are underestimated threats to users' privacy, as complex and often obfuscated third-party libraries make it hard for application developers to estimate the correct behaviors of third-party libraries. More critically, a wide adoption of native code (JNI) and dynamic code executions such as Java reflection or dynamic code reloading, makes it even harder to apply state-of-the-art security analysis.**

In this work, we propose FLEXDROID, a new Android security model and isolation mechanism, that provides dynamic, fine-grained access control for third-party libraries. With FLEXDROID, application developers not only can gain a full control of third-party libraries (e.g., which permissions to grant or not), but also can specify how to make them behave after detecting a privacy violation (e.g., providing a mock user's information or kill). To achieve such goals, we define a new notion of principals for third-party libraries, and develop a novel security mechanism, called inter-process stack inspection that is effective to JNI as well as dynamic code execution. Our usability study shows that developers can easily adopt FLEXDROID's policy to their existing applications. Finally, our evaluation shows that FLEXDROID can effectively restrict the permissions of third-party libraries with negligible overheads.

I. INTRODUCTION

Mobile application (or app for short) developers are becoming increasingly dependent on third-party libraries. For example, almost 50% of free apps embed advertisement libraries (also known as ad libraries) provided by ad companies to enable in-app advertising [32]. Many other third-party libraries are also used by app developers to provide various features at significantly reduced development time and cost. To name a few, such features include in-app purchases [14], UI [2], client-side cloud computing interfaces [6], game engines [17], analytics [8], and PDF view [3]. Unfortunately, third-party libraries come at costs of potential privacy violation of users.

[‡]A corresponding author.

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23485

A great deal of previous works have increasingly called attention to potential security and privacy risks posed by Android advertising libraries [22, 32, 40]. Many ad libraries access privacy-sensitive information even without notification to users or application developers. Our analysis of 100,000 Android apps reveals that in addition to ad libraries, various other third-party libraries (e.g., Facebook, Flurry, RevMob, Paypal) covertly utilize Android APIs to access privacy-sensitive resources such as `GET_ACCOUNTS`, `READ_PHONE_STATE`, or `READ_CALENDAR` without mentioning them properly in their Developer's Guides. The current Android platform provides coarse-grained controls for regulating whether third-party libraries access private information, allowing them to operate with the same permissions as their host apps. For example, if an app has the `GET_ACCOUNT` permission to access a user's online account information (e.g., Gmail and Facebook IDs), app developers have no way of disallowing third-party libraries to access such account information. As a result, they must blindly trust that third-party libraries will properly respect access to privacy-related information of app users.

This paper presents FLEXDROID, an extension to the Android permission system that allows app developers to control access to a user's private information by third-party libraries. Our primary goal is to enable in-app privilege separation among a host application and one or more third-party libraries, while running all in the same process and so, the same UID privilege. To this end, FLEXDROID provides an interface, as a part of the app manifest, for app developers to specify a set of different permissions granted to each third-party library. Upon any request for a user's information, FLEXDROID seeks to identify the principal of the currently running code (either an app or third-party libraries) via our new security mechanism, called *inter-process stack inspection*. Depending on the identified principal, FLEXDROID allows or denies the request by dynamically adjusting the app's permissions according to the pre-specified permissions in the app's manifest.

Since FLEXDROID assumes that third-party libraries are potentially malicious, a key challenge is to draw clear and trustworthy boundaries between the host app and their third-party libraries at runtime. This becomes particularly challenging as many third-party libraries utilize various dynamic features of the Java language including native methods (JNI), Java reflection, and dynamic class loading. If such dynamic code execution is not considered carefully, virtually all adversarial third-party libraries can bypass the proposed security mechanism. From our analysis of 100,000 Android apps, 72% of 295 third-party libraries are found to rely on dynamic code execution. Moreover, host apps and third-party libraries involve complex control- and data-flow dependencies through diverse features, such as class inheritance and callback methods. Unlike existing solutions that rely on static analysis [22, 32, 40] or cross-app privilege separation [37, 39, 45], the proposed stack-based inspection technique not only can faithfully identify the module of third-party libraries but also can regulate them at runtime without limiting the use of widely-adopted dynamic code execution.

Experiments with our prototype on Android 4.4.4 show that FLEXDROID has a high degree of usability and compatibility; app developers can easily apply FLEXDROID's policy to isolate existing third-party libraries. Our experimental results also indicate that FLEXDROID incurs negligible overheads. Experiments with an open-source K-9 email app show that FLEXDROID adds 1.13-1.55 % overheads in launching the application and sending an email, compared with stock Android.

We make three contributions as follows:

- We report several new findings from our analysis of 100,000 real-world Android apps and 20 popular third-party libraries (see §III-B). For example, 72% of 295 third-party libraries employ the dynamic code execution using various Java language features, and 17% of them rely on JNI.
- FLEXDROID extends Android's permission system by providing in-app privilege separation for a wide range of apps, while placing no limit on the use of native code and reflection and requiring no modification to the code except the manifest.
- To the best of our knowledge, FLEXDROID is the first system that adopts a hardware-based fault isolation using the ARM Domain to sandbox third-party libraries in Android apps. We describe our engineering experience in implementing the hardware-based fault isolation and conduct experiments using real-world Android apps.

II. RELATED WORK

Detecting in-app security/privacy risks. Security and privacy issues in in-app advertising have recently attracted considerable attention. Several studies [22, 32, 40] examine Android advertising libraries through static analysis. Their findings indicate that many in-app ad libraries collect privacy-sensitive information [32] even without mentioning the use of privacy-related permissions in their documentation [40], while such negative behaviors may be growing over time [22]. Livshits et al. [35] propose an automated approach to identify and place missing permission prompts where third-party libraries may potentially misuse permissions. A few studies employ dynamic analysis to disclose potential risks [21, 25]. Brahmastra [21] is an automation tool to test the potential vulnerability of third-party libraries embedded into mobile apps, beyond the reach of GUI-based testing tools. MAdFraud [25] adopts a dynamic analysis to detect fake ad clicks by host applications. In our work, we analyzed 20 popular third-party libraries in depth to understand how they covertly request privacy-sensitive APIs (e.g., SMS, calendar, location, etc) and how each of them rely on dynamic features of the Java language (see Table I).

Protecting sensitive data against privacy-unaware third-party libraries. Several works have introduced protection mechanisms against permission-abusing third-party libraries. All existing approaches, except one [43], share the principle of separating the privilege of third-party libraries from the host applications by running them in separate processes with the goal of isolating a specific type of third-party libraries unlike FLEXDROID. AdSplit [39] and AFrame [45] isolate ad libraries by running them as separate applications with limited permissions, and NativeGuard [41] takes a similar approach for native third-party libraries (written in C/C++) such as FFmpeg [7]. AdDroid [37] places advertising functionality into a new Android system service to separate it from host apps. LayerCake [38] isolates user interface (UI) libraries from its host app to support secure third-party UI embedding on Android. On the other hand, Compac [43] is the closest to our work. Like us, it provides in-app privilege separation for host app and third-party libraries while both running in the same process. However, Compac's approach is not applicable to third-party libraries relying on JNI or dynamic code execution, which almost all of the popular third-party libraries rely on (see Table I). FLEXDROID not only enables isolation of any third-party libraries but also allows app developers to choose how each library behaves upon their privacy violation.

Cross-app privacy leaks. A great deal of previous studies aim to mitigate the confused deputy problem, inter-application or inter-component permission leaks by either checking IPC call chains or by monitoring the run-time communication between apps [23, 24, 26, 29, 30]. TaintDroid [27] and DroidScope [44] employ efficient taint tracking to monitor real-time data leakage.

III. MOTIVATION

Each Android app runs in its own sandbox, an isolated process with an application-specific UID allocated at the installation time. In order to get access to sensitive resources (i.e., contact, location, SMS, camera) outside the app sandbox, an app must specify proper permissions in the manifest, `AndroidManifest.xml`. The Android permission model only offers an "all-or-nothing" installation option for users to accept all the permissions requested for installation or simply refuse to install to the app. More critically, the app is able to keep accessing the granted resources all the time, no matter what components or modules of the app are requesting access to.

An Android app often contains third-party libraries to offer advanced functionality. In the Android security architecture, all the modules in the same app have exactly the same permissions, resulting in *overprivileged* third-party libraries. That is, some third-party libraries can have more privilege than what they need, while being malicious (intentionally or not). As a result, a malicious third-party library may exploit the app's `SEND_SMS` permission to send premium-rate SMS messages at the users' cost (i.e., £5 for each message [9]). This could make app developers vulnerable to lawsuits since they are considered liable for all the behaviors of their apps.

A. Potential Attack Scenarios

We consider three possible attack scenarios where overprivileged third-party libraries access privacy-sensitive resources regardless of the app developers' intention.

Libraries abusing undocumented permissions. App developers know what permissions a library will use based on its documentation (i.e., Developer's Guide), specifying its must-have permissions and optionally required permissions. Due

Name	Category	Accounts	Phone information	Internet	Read SMS	Write SMS	Read Calendar	Write Calendar	Write Settings	Get Tasks	Read Bookmark	Record Audio	Location	Class Loading	Reflection	Callback	Class Inheritance	JNI
Facebook	Social	·	×	O	·	·	·	·	×	·	·	·	×	✓	✓	✓	✓	✓
Flurry	Analytics	·	×	O	·	·	·	·	·	·	·	·	O	✓	✓	·	✓	·
RevMob	Advertising	×	△	O	·	·	·	·	·	·	·	·	·	·	·	✓	✓	✓
Chartboost	Advertising	·	×	O	·	·	·	·	·	·	·	·	·	✓	✓	·	✓	✓
InMobi	Advertising	·	·	O	×	×	△	△	·	·	·	·	△	✓	✓	✓	✓	·
Millennialmedia	Advertising	·	·	O	·	·	·	·	·	·	·	O	×	✓	✓	·	✓	✓
Paypal	Billing	·	×	O	·	·	·	·	·	·	·	·	×	✓	·	·	✓	·
Umeng	Analytics	·	O	O	·	·	·	·	×	×	·	·	×	✓	✓	·	✓	✓
AppLovin	Advertising	△	O	O	·	·	·	·	·	·	·	·	·	✓	✓	✓	✓	·
Pushwoosh	Notification	·	O	O	·	·	·	·	·	·	·	·	×	·	✓	·	✓	·
Tapjoy	Advertising	·	O	O	·	·	×	×	·	·	·	·	×	✓	✓	✓	✓	·
AppFlood	Advertising	·	△	O	·	·	·	·	·	·	·	·	△	·	✓	·	·	·
OpenFeint	Social	O	O	O	·	·	·	·	·	·	·	·	×	·	✓	✓	✓	·
Airpush	Advertising	×	△	O	·	·	·	·	·	·	×	·	×	·	✓	·	✓	·
Youmi	Advertising	·	O	O	·	·	·	·	·	×	·	·	×	·	✓	·	✓	·
Cauly	Advertising	·	·	O	·	·	·	·	·	×	·	·	△	·	·	✓	✓	·
Socialize	Social	·	△	O	·	·	·	·	·	·	·	·	△	✓	✓	·	✓	·
Domob	Advertising	·	O	O	·	·	·	·	·	·	·	·	×	✓	✓	·	✓	·
Leadbolt	Advertising	×	△	O	·	·	×	×	·	·	·	·	△	✓	✓	✓	✓	·
MobFox	Advertising	·	×	O	·	·	·	·	·	·	·	·	△	✓	✓	·	✓	·

TABLE I: Characteristics of third-party libraries. Columns 3-14 show the permissions potentially used by apps (O: required permission, △: optional permission, ×: undocumented permission). The rest of columns are related to runtime behavior and dependency with host apps.

to the lack of in-app privilege separation, however, libraries can take free rides to access private resources even without corresponding permissions documented properly (i.e., *undocumented permissions*) when its host app has the permissions. If the library dynamically checks whether it has certain permissions or catches a security exception, it can abuse those permissions without noticing users.

Contaminated libraries. Although the original library is legitimate, an adversary is able to rewrite its binary or source code and redistribute it. When an app mistakenly uses such *contaminated libraries*, the users who install the app are in danger of severe privacy leaks, and may suffer monetary damages. In 2013, a malware called Uten.A was repackaged and distributed while disguising itself as Umeng SDK library, a mobile analytic platform [28]. Some legitimate gaming apps available in the Google Play used this malware, and many users installed those apps. Devices affected by the malware are silently subscribed to a premium-rate SMS service, and SMS messages are sent to the service at the users' expense.

Vulnerable libraries. A third-party library may execute a class or JavaScript code[1] downloaded from Internet at runtime. If the mobile device is connected to an unsafe network and the library does not encrypt the code, an attacker can replace it with a malicious code. Such a malicious code can exploit the host app's permission to leak personal information (e.g., A malicious advertisement written in JavaScript can read the device's IMEI code if the ad library exposes the `getDeivceId` method to JavaScript using `addJavascriptInterface`).

[1] A JavaScript code can run on a WebView-based framework via `JavascriptInterface` provided by third-party libraries. Open Rich Media Mobile Advertising (ORMMA) [13] is an example of such frameworks.

B. Real-world Findings

In order to look at how our attack scenarios are pervasive in the wild, we investigate the characteristics of third-party libraries used in Android applications. Here is a brief summary of our findings:

1) 17 of 20 popular third-party libraries use undocumented permissions.
2) 72% of 295 third-party libraries rely on dynamic code execution.
3) 17% of 295 third-party libraries use JNI.

Methodologies. We randomly collect the 100,000 Android apps from the *Playdrone* dataset [42], an archive of Android Application Package (APK) files downloaded from the Google play store. To perform a static analysis for applications, we dissect each APK file using *apktool* [4] which translates the Android app's *.dex* code into the corresponding *.smali* code. In our analysis, we particularly focus the following characteristics of each library which are relevant to our threat model.

Permissions exploited by third-party libraries. We chose 16 different Android permissions that allow a third-party library to access privacy-sensitive information such as device ID, SMS, contacts, and a device's current location. Since a third-party library accesses such information via the Android's APIs or content provider, we need their mapping to the corresponding necessary permission. We extended the findings from PScout [19] that provides the mapping between Android's native API, the content provider's URI, and permissions. In addition, we manually examined and compared the required permissions that we could find from the developer's guide of each third-party library, and the exploited permissions that they could covertly use when embedded by an app.

Dynamic code execution. As previously mentioned in §III, third-party libraries deeply rely on dynamic features of the Java programming language, including runtime class loading, Java reflection, and multi-threading. To identify whether such techniques are used by third-party libraries, we build code-level signatures of techniques and apply them to the smali code of apps.

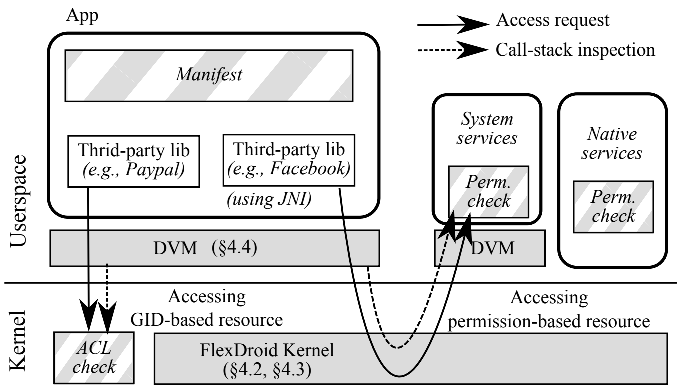
Interaction between a library and its host application. In addition to the reliance of dynamic code features, third-party libraries interact with their host apps in various ways, which makes it hard to analyze and disambiguate the boundaries between third-party libraries and their apps. These techniques include callback, class inheritance and JNI and make FLEXDROID distinct from previous works such as AdDroid [37] and AdSplit [39]. To examine such use cases, for each application, we check whether a host app inherits classes provided by third-party libraries, whether it uses any callback method, and whether third-party libraries embed JNI.

Unlike previous works [32, 40] focusing on specific third-party libraries (i.e., ad), our investigation is not limited to ad libraries, but includes social, billing, analytics and more. In addition, our investigation covers dynamic execution patterns used by third-party libraries, which serves as a primary motivation of our work.

Summary of results. Table I summarizes the results of our findings for third-party libraries used in 100,000 Android apps. We extracted the top 20 popular libraries which use at least one permission out of 16 permissions mentioned above. Note that in the result, we include a third-party library only if it provides a clear documentation on necessary permissions and a method of integration.

We found that some libraries could attempt to use permissions which are not documented in their developer's guide (marked as ×). For instance, ad libraries such as RevMob, Airpush, and Leadbolt potentially utilize host app's `GET_ACCOUNTS` permission while its developer does not mention the permissions as required or optional. With the `GET_ACCOUNTS` permission, a library can obtain a user's online account information on the phone such as Gmail, Facebook, and Dropbox.

In addition, our findings show that most libraries can make dynamic execution paths with dynamic class Loading, Java reflection, and Java thread. All the listed 20 libraries use at least one technique, and 16 libraries utilize all three techniques. Moreover, it turns out that host apps and third-party libraries have strong dependencies. Specifically, most of the ad libraries including Flurry, AppLovin, and Tapjoy need to obtain the host app's context to show and manipulate advertisements. We also found that some libraries use class inheritance, perhaps for simple integration. For instance, Parse and Chartboost recommend app developers to inherit the provided class for simplifying the integration process. In addition, we found that



App
Access request
Call-stack inspection
Manifest
Thrid-party lib (e.g., Paypal)
Third-party lib (e.g., Facebook)
(using JNI)
System services
Perm. check
Native services
Perm. check
Userspace
DVM (§4.4)
DVM
Kernel
Accessing GID-based resource
Accessing permission-based resource
ACL check
FlexDroid Kernel (§4.2, §4.3)

Fig. 1: Overview of FLEXDROID's design. The gray boxes represent FLEXDROID's modification of the existing Android components and the gray-stripped boxes represent new components that FLEXDROID introduced for fine-grained permission checking for third-party libraries. In FLEXDROID, all requests to resources will be checked based on the fine-grained module (host app or third-party libraries) and the app-specified manifest.

17.1% of 295 third-party libraries use native code through JNI (see Table II). These tight integration and dynamic behavior of off-the-shelf libraries make the state-of-the-art analysis or enforcement challenging. In FLEXDROID, we attempt to provide a practical yet strong security mechanism to isolate such third-party libraries.

Technique	Out of 295 libs
Class Loading	27.9%
Reflection	49.6%
Class Inheritance	71.5%
JNI	17.1%

TABLE II: Java language techniques used in third-party libraries.

C. Threat Model

FLEXDROID assumes a strong adversary: third-party libraries are potentially malicious, their code and logic are not directly visible to app developers (e.g., obfuscated), and they might use dynamic features of the Java language. However, app developers explicitly know what third-party libraries are for (that is why app developers want to embed them in the first place). Given a high-level functional description (e.g., ad or analytics) and perhaps a manifest provided by a third-party library, app developers should be able to have enough freedom to adjust the manifest and seamlessly integrate them without compromising usability.

IV. FLEXDROID DESIGN

A. Overview

FLEXDROID targets at a new permission system that adjusts the permissions of Android apps dynamically so as to enforce fine-grained access controls for untrusted application modules (i.e, third-party libraries).

A module is a collection of code. FLEXDROID uses the module as the unit of trust, while an entire app (or a process)

```
<flexdroid android:name="com.third.party.library"
           android:mockOnException="true">
  <allow android:permission="android.permission.READ_CONTACTS" />
  <allow android:permission="android.permission.READ_EXTERNAL_STORAGE" />
  <allow android:permission="android.permission.INTERNET" />
</flexdroid>
```

Fig. 2: An example rule in an app's manifest that allows a third-party library to access contacts, sdcard and the Internet.

is the unit of trust in the Android's permission system. In FLEXDROID, the boundary of a module aligns with either a class or a method. Key features of FLEXDROID are as follows (see Figure 1). (1) App developers specify a set of permissions for each individual module in the manifest. (2) Upon each request for resource access, FLEXDROID identifies the context of execution (i.e., a call chain of modules leading to the current execution) by inspecting the Dalvik call stack. (3) FLEXDROID then determines whether to accept or decline the request according to the permissions commonly granted to the modules.

Achieving the above features for a wide range of Android apps presents several challenges:

Secure inter-process stack inspection. FLEXDROID relies on call stack trace to mediate access from untrusted modules, while conducting access control in a separate address space from the modules. Thus, FLEXDROID should provide a mechanism to utilize the call stack trace across the process boundary (see §IV-B).

Integrity of stack principal. Preserving the integrity of Dalvik call stack and stack tracer is crucial for correctness. It is, however, very challenging with the use of native code via Java Native Interface (JNI). Malicious third-party libraries may use native code to tamper with Dalvik's data memory, for example, to counterfeit call stack frames directly. Therefore, it is critical to provide a tamper-resistant memory protection mechanism for accuracy of FLEXDROID permission system (see §IV-C).

Handling dynamic code execution. There are various ways of dynamic code execution in Java, which could blur the boundary of modules. For instance, Java reflection can be used to enable dynamic code generation and execution across classes, and code can be executed on a new thread with Java thread creation. These make it very complicated to identify module boundaries clearly. Since FLEXDROID aims to support accurate fine-grained access control for individual modules, it should be able to adjust the permissions of modules dynamically when the modules are modified at run time for execution (see §IV-D).

In addition to addressing the above challenges, FLEXDROID also aims to offer a high level of usability. To this end, FLEXDROID provides developers with programming interfaces, in the form of simple XML manifest rules, to restrict third-party libraries' privileges. Figure 2 shows an example rule in FLEXDROID's policy. Using the `flexdroid` tag, developers can specify a third-party library of interest and configure that library's privileges with the `allow` tag. Inspired by [20, 33], FLEXDROID also provides a `mockOnException` attribute to enable developers to choose whether FLEXDROID should offers mock data (e.g., fake IMEI code) upon a request for an unauthorized resource.

B. Secure Inter-Process Stack Inspection

The Android permission system conducts access control outside the process boundary of an app of interest. This is mainly because a significant portion of Android apps make use of native code for various reasons. If permission checking is performed inside the Dalvik virtual machine, just as in the traditional JVM's security architecture [31], native code can circumvent permission checking through low-level system calls or tampering with the Dalvik's data memory. Thus, Android performs permission checking in a separate address space to protect memory tampering or in the kernel to secure the use of low-level system calls, and FLEXDROID does so too.

This entails a secure inter-process stack inspection mechanism for FLEXDROID. FLEXDROID requires extra information (i.e, Dalvik call trace) to understand the current execution context for fine-grained access control. Thus, for each app, FLEXDROID creates a single special-purpose thread, called *stack tracer*, that sends Dalvik call trace data to the system's permission checker upon request. It is important to note that a malicious third-party library may pretend to be a stack tracer to send a fake call trace data. To protect against such a forgery, FLEXDROID provides a secure communication channel, *stack transmission channel*, between individual stack tracer threads and the permission checkers. FLEXDROID restricts each app to create only one trusted stack tracer, and ignores all attempts to use the stack transmission channel except those by the stack tracer.

To guarantee the authenticity of stack tracer and its enrollment for the stack transmission channel, FLEXDROID adds a unique stack tracer into each app and the stack tracer registers itself to the channel at the app's initialization time (e.g., in Android, right after Zygote forks the app process). Since the initialization occurs before the execution of the app's code, a malicious library code cannot create a thread to pretend to be a stack tracer. Notice that the malicious code could attempt to tamper with memory used by the stack tracer thread at runtime to counterfeit data and control or to raise faults. This is impossible because of our in-app memory protection mechanism. (See §IV-C for details of the in-app memory protection.)

The detailed procedure of inter-process stack inspection differs according to the type of requested resources. Android resources can broadly fall into two categories: user-space and kernel-space, depending on which components are responsible for access control. Android permission system conducts permission checking for user-space resources at the framework layer and for kernel-space resource in the kernel, respectively.

User-space resources. Apps access user-space resources through the interfaces provided by Android system services. Such resources include system resources (GPS, camera, etc.) and app components (activity, service, content provider, and broadcast receiver). Apps and system service processes, which are user processes, communicate through the Binder IPC (Inter Process Communication) mechanism. A system service, called *Package Manager* (PM), is involved in the permission checking of user-space resources, while it maintains various kinds of information (i.e., a set of permissions granted) related to the application packages installed on the device.

In FLEXDROID, a typical control flow to access user-space resources is as follows. Like Android, when an app requests to access a resource (e.g., location) to a corresponding system service (e.g., Location Manager), the system service process queries the *Package Manager* (PM) to see whether the app has proper permission. FLEXDROID provides inter-process stack inspection to conduct access control at the granularity of a module. Upon a request from PM, the stack tracer of the app passes the Dalvik call trace to PM via the secure stack transmission channel mentioned above. PM then looks through all the modules involved in the current access request to find out the commonly granted permissions among them.

Kernel-space resources. Android apps access kernel-space resources (Internet, external storage, Bluetooth, etc.) via system calls. In Android, unlike the user-space resource case, PM does not conduct permission checking for kernel-space resources. Instead, at the initialization of an app, PM passes a set of permissions (granted to the app) to the kernel. The kernel then performs permission checking itself, using Linux's Access Control Lists [1] [2]. FLEXDROID enforces the kernel to conduct inter-process stack inspection through the stack transmission channel during permission checking. Additionally, in FLEXDROID, upon app installation, PM sends to the kernel a set of granted permissions to each module in the app so as to avoid expensive user-kernel communication later on.

Note that the above process of inter-process stack inspection for user- and kernel-space resources works in a synchronous manner. That is, after requesting the access to resources, the thread is suspended so that the thread's context will remain consistent until the end of the process.

FLEXDROID extracts information of a caller method from the corresponding stack frame in its Dalvik call stack and the permission set of each modules. The information contains the sequence of method call, the principal of module, and access permission on the memory region of the call stack, as depicted in Table III. For instance, a stack frame in Table III illustrates that `com.malicious.library.WebCodeRunner.run` called the `com.ImgLib.takePicture` method which invokes the `takePicture` method of Android's Camera Class. Here we assume that `com.ImgLib.takePicture` is a JNI wrapper method to take a photo. Such a JNI method can maliciously manipulate the call stack through a memory tampering attack. To prevent such an attack, we introduce the design of Dalvik memory protection in §IV-C.

[2] In Android, each kernel-space resource is mapped to a unique GID.

	P	M	Call stack trace
↓	A	✗	android.app.Activity.onCreate
	A	✗	com.example.userapp.MainActivity.onCreate
↓	L	✗	com.malicious.library.WebCodeRunner.run
	L	✓	com.ImgLib.takePicture (JNI wrapper)
	L	✓	android.hardware.Camera.takePicture

P: Principal M: Potential modification
A: Host application L: Third party library

TABLE III: A snapshot of an app's call stack: application (A) invokes a JNI library (L) to take a picture. Since FLEXDROID protects the app's call stack when executing the library, the JNI library cannot counterfeit its principal to bypass FLEXDROID's rules.

C. *Ensuring Dalvik Stack Integrity against Native Code*

Android supports the Java Native Interface (JNI) and allows developers to implement parts of an app or library to incorporate native libraries. With JNI, a developer can re-use existing libraries written in native languages or improve an app's performance.

Despite its advantages, it renders the memory safety of the Java programming language obsolete, which results in security threats in Android. In FLEXDROID, such memory safety problems make it hard to guarantee the integrity of Dalvik call stack and stack tracer, as they might be corrupted or even manipulated by malicious third-party libraries.

1) Potential Attacks: We consider three potential attack scenarios where malicious JNI code might attempt to bypass FLEXDROID's security mechanism, in particular, by compromising the integrity of its principal.

1) **Compromising the stack tracer.** An attacker can effectively guess the address of a memory region (e.g., a region that stores stack traces) used by the stack tracer, and manipulate its content to counterfeit its principal.
2) **Manipulating Dalvik stack.** An attacker can directly manipulate Dalvik stacks, thereby corrupting the integrity of the call stack used for the inspection.
3) **Hijacking the control data.** Although the code segment in memory is typically read-only, an attacker can modify the read-only protection with `mprotect()` system call, and manipulate the code for a malicious purpose. Furthermore, an attacker can compromise code pointers (e.g., function pointers in heap, return addresses in stack).

2) Defenses: Since the above attacks rely on JNI's ability to access memory regions of Java code, one might think that those attacks can be prevented by making important memory regions read-only. However, such a solution does not work properly for protecting the integrity of call stack for the following two reasons.

First, it is difficult to track all memory regions which need to be protected. Although we can easily pinpoint and protect memory locations of Dalvik call stack and buffers of stack tracer, it is difficult to precisely track all function pointers within a process.

Moreover, even if we protect important memory regions by making them read-only, a multi-threaded malicious process can bypass this protection mechanism. For instance, suppose that there are two threads `T1` and `T2` within a process, and they are executing JNI and Java code, respectively. Then, `T1` is able to counterfeit the stack principal by manipulating the Dalvik call stack of `T2`; simply making the Dalvik call stack of `T2` read-only can freeze `T2`.

Thus, making specific memory locations read-only is not a suitable solution to guarantee the integrity of call stack. This motivates us to introduce a JNI sandboxing mechanism to FlexDroid. In this approach, FlexDroid prevents JNI from accessing memory regions of Java code by sandboxing JNI code.

We have three different design choices of implementing this protection mechanism.

1) **Process separation.** Process separation naturally supports the memory sandboxing. Since it lets a JNI thread run in a separate process, JNI code is not able to access the memory region of Java code directly. NativeGuard [41] applied this strategy to isolate JNI libraries in Android applications.
2) **Software Fault Isolation (SFI).** SFI restricts memory accessing by JNI through masking the operands of store and jump instructions used in JNI. Since an attacker can perform indirect attacks via static or shared libraries used by JNI, SFI also needs to confine memory access by such libraries by masking those two instructions used in such libraries. AppCage [47] applied this design in Android to prevent JNI libraries from accessing restricted APIs.
3) **Hardware Fault Isolation (HFI).** HFI leverages the memory separation mechanism supported by a processor such as the memory domain in ARM architecture. It strictly confines memory access by executing JNI in the restricted memory domain. ARMLock [46] utilizes the memory domain to implement HFI in the ARM-Linux architecture.

Each design has its own advantages and disadvantages. The process separation approach enables JNI sandboxing easily, while imposing a large amount of overhead in switching process contexts. On the other hand, unlike process separation, SFI does not incur any process context switching overhead for sandbox switching. However, it imposes runtime overhead due to extra instructions for masking store and jump instructions. Unlike SFI, HFI does not incur runtime overhead for masking operands, while adding a little overhead in switching between JNI and Java code by updating register values (i.e., Domain Access Control Register (DACR)), which is negligible.

FlexDroid takes the HFI approach to implement JNI sandboxing, as the process separation and SFI approaches incur significant overheads compared to HFI, and most Android devices based on ARM architecture natively support the concept of domain which we can leverage for our implementation. We introduce two memory domains called JNI and Java domain, which represent restricted memory regions assigned for JNI and Java code, respectively.

However, there are a couple of challenges raised in applying this design to FlexDroid. First, it is not trivial to apply the JNI sandboxing to existing JNI and shared libraries without modifying their implementation. In the default setting, when JNI attempts to access memory regions for the stack, heap, and shared libraries located at the Java domain, the domain fault occurs since only the JNI can access the JNI domain. To overcome this, FlexDroid provides the JNI domain with separate stack and heap, and loads a set of necessary shared libraries located at the JNI domain.

Another challenge is that an attacker can indirectly manipulate important memory regions through the communication channel between JNI and Java code. For instance, when JNI calls Java API, it can pass a pointer variable that points the address of Dalvik stack to Java code as an argument or as a return value. Such a pointer passing can be used to manipulate the Dalvik stack, if the pointer is set properly, bypassing the JNI sandboxing mechanism.

To prevent such attacks, FlexDroid classifies pointer variables as valid or potentially malicious pointers and disallows the latter ones to be passed to Java code via Java API. FlexDroid considers a pointer variable as valid if one of the following cases holds:

1) It points a memory address within the JNI domain
2) It points a memory address within Java domain, when the address has been returned from Java code via Java API call.

If JNI attempts to pass a pointer that does not satisfy the above conditions, the pointer can be used for a malicious purpose.

FlexDroid manages a table called Valid Address Table (VAT) that maintains a list of memory addresses which have been returned from Java code via Java API calls. When JNI calls a Java API and the API returns a pointer, FlexDroid adds the memory address that the pointer stores to the VAT. Then, when JNI passes a pointer via any Java APIs, FlexDroid checks whether the pointed address is in the JNI domain (i.e., valid case 1) or exists in VAT (i.e., valid case 2). Otherwise, the pointer is invalid and FlexDroid rejects the passing operation.

Despite the validity checking for pointers, JNI still can cause integrity or confidentiality violations with the type-confusion attack [36]. For instance, suppose that there are two Java classes called `PrivClass` which has a private integer member variable and `ListClass` which is an implementation of linked lists and has a pointer to the next element as a member variable, and a function called `set_null_to_list` that takes a `ListClass` pointer and sets the next element of the list to `null`. JNI can make the value of the private field of `PrivClass` instance to `null` by casting the pointer to the instance to `ListClass` pointer type and passing it to `set_null_to_list`.

To prevent this attack, FlexDroid verifies the type of pointer which belongs to the second condition of the valid pointer. In VAT, FlexDroid maintains a type of pointer for each address entry. With this information, FlexDroid accepts pointers only if the addresses to which they point exist in VAT and their types match the corresponding type entries in VAT.

D. Dynamic Permission Management

Dynamic code execution through various Java features (reflection, dynamic class loading, native methods, multithread-

```
package com.malicious.lib
class A
  method launch_attack
    generateClass("com.host.B")
    generateMethod("com.host.B", "malFunction")
    loadClass("com.host.B")
    C.registerCallback(new B())
  end method
end class
```

(a) Pseudocode of class `A`: registering a callback

	P	Call stack
↓	H	com.host.C.runCallback
	L	com.host.B.malFunc

(b) Call stack when the callback method is executed

Fig. 3: Example of an attack using dynamic code generation and execution

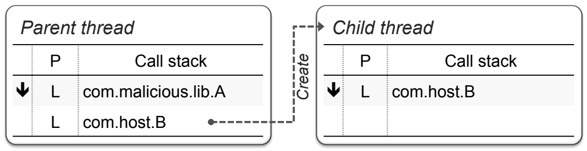


Fig. 4: Example of an attack using thread creation

ing, callback, etc.) blurs the boundaries between modules at run time. More importantly, dynamic code instrumentation via reflection makes it difficult to maintain the mapping between modules and permissions accurately. Reflection is a powerful and widely-adopted feature that allows programmers to inspect or modify any code at run time across Java classes for various benefits (e.g., logging). At the same time, reflection also opens a host of security threats. For instance, a malicious third-party library may use reflection to generate code for an existing host module, and dynamically load and run the code generated to perform harmful actions in the name of the existing host module. Or, it may even simply change its class name at run time to pretend that it is a trusted host module class.

For example, suppose a malicious library class `com.malicious.lib.A` (`A`) generates a new code that contains a harmful method. `A` then loads the new code with the name of a trusted host class `com.host.B` (`B`) and registers the harmful method of `B` as a callback method from another trusted host class `com.host.C` (`C`) (see Figure 3a). Afterwards, `A` may terminate its execution, disappearing out of the call stack trace, before `C` invokes the harmful callback method of `B` (see Figure 3b). This way, a malicious module can perform harmful actions without its name appearing in the call stack trace. This could happen as a result of the call stack not capturing the context where the callback method was registered during the execution of a callback.

Another example is related to Java thread creation, where the call stack of a new thread created does not capture the context of the parent thread. Suppose a malicious class `A` invokes a method of a trusted class `B` and `B` creates a new thread. Figure 4 shows the call stack trace of the parent thread and the new child thread. Then, the call stack trace of the new thread contains the method of `B` but no trace related to `A`. This way, the malicious class `A` may perform the confused-deputy attack to exploit the permissions of the trusted class `B`.

Note that the use of the callback does not affect the boundary since when the callback method is called, the principal of callback method can still be identified from the call stack. Besides, even in the case where the callback is intertwined with reflection and class loading, although it looks complicated, is just another case of reflection.

As such, the boundaries between modules become unclear with dynamic code generation and execution. If we keep a static mapping between modules and permissions, it is unlikely to prevent untrusted modules from exploiting dynamic code execution to help or perform malicious actions. Thus, FLEXDROID enforces dynamic permission management for modules to resolve the problem. A basic idea behind the management is that a runtime instance of a module is assigned a set of permissions granted at app installation, except two cases of dynamic class loading and thread creation. As explained above, a malicious module can make use of reflection or dynamic code generation along with dynamic class loading to take harmful actions without leaving any trace on the call stack (see Figure 3b). In addition, Java thread creation is another feature that malicious modules can exploit for similar attacks, as shown in Figure 4. Hence, in those two cases, FLEXDROID sets the permissions of a runtime instance of a module with respect to not only its installation-time permissions but also the context at runtime.

For ease of presentation, let us define some terms before presenting the rules for dynamic permission management. Each module M has a set of installation-time permissions (denoted by $P_I(M)$). Let $\langle M, T\rangle$ indicate a runtime instance of module M running on thread T and $P(\langle M, T\rangle)$ represent its permissions. For each thread T, its permission $P_\tau(T)$ is defined as the intersection of the permissions of all modules on the call stack of T, i.e.,

$$P_\tau(T) = \bigcap_{\forall M_i \in CS(T)} P(\langle M_i, T\rangle),$$

where $CS(T)$ is a set of modules on the call stack of thread T.

When a new runtime instance $\langle M, T\rangle$ is created, FLEXDROID determines its permissions depending on how it is created in the following way:

- If $\langle M, T\rangle$ is created via dynamic class loading, its permissions are set to the intersection of its installation-time permissions and the thread's permissions. $P(\langle M, T\rangle) = P_I(M) \cap P_\tau(T)$.
- If $\langle M, T\rangle$ is created via thread creation, its permission is determined as the intersection of its installation-time permissions and the parent thread's permissions. $P(\langle M, T\rangle) = P_I(M) \cap P_\tau(T')$, where a thread $\mathtt{T}'$ is a parent thread of $\mathtt{T}$.
- Otherwise, $P(\langle M, T\rangle) = P_I(M)$.

A similar reasoning to the above rules was used to address the confused-deputy attack [24, 26, 29].

V. IMPLEMENTATION

We implemented a prototype of FLEXDROID on Android 4.4.4 (KitKat) and Linux kernel 3.4.0. We have modified various components of the Android framework, the Dalvik

	#Files	Insertion	Deletion
Kernel	28	1,831	25
Android Framework	46	1,466	77
Dalvik VM	24	6,081	22
Bionic	23	2,827	70
Others	12	95	24
Total	133	12,300	218

TABLE IV: FlexDroid's component-wise complexity in terms of lines of code.

VM, the Bionic, the Linux kernel, the Java core library, the Binder library (user layer interface of Binder IPC), and the SELinux setting, which in total consists of 12,300 LoC across 133 files (see Table IV).

In the rest of this section, we first describe the detailed implementation of inter-process stack inspection. We then present how FlexDroid handles general Java techniques (i.e., JNI and reflection) that can be used to bypass our security mechanism. Next we share our experience to cope with deadlocks and optimize performance further during the implementation of FlexDroid.

A. Inter-Process Stack Inspection

There are three key components involved in the inter-process stack inspection; stack tracer, stack transmission channel, and *permission checkers* (e,g., the Android framework's Package Manager (PM) and the Linux kernel's ACL) which communicate with the stack tracer to obtain Dalvik call trace and conduct access control.

Stack tracer. In order to send the Dalvik call trace to permission checkers, FlexDroid adds a unique stack tracer into each app right after Zygote is forked at the app's initialization time. The stack tracer enrolls itself via the `ioctl` system call in the stack transmission channel. After registration, the channel stores it with its process ID (PID) as a key into the RBTree and is inactivated until a request comes from either of the permission checkers.

In detail, the stack tracer performs the stack inspection for a target thread as follows. It first holds Dalvik VM thread lock to block garbage collection. It then suspends the target thread and traces the stack frame pointer to get the call trace (utilizing `DVMFillInStackTraceRaw` which is originally used to print stack when an exception occurs). It resumes the thread and releases the Dalvik VM thread lock.

Stack transmission channel. The stack transmission channel is a special purpose device driver that is designed to handle the communication between a permission checker and a stack tracer. The communication gets started by the permission checker when it leaves a stack inspection request for a target thread of TID (Thread ID) and PID. The channel then finds and wakes up the pre-registered stack tracer for the process of PID and sends the target thread information (TID) to the stack tracer. The channel waits for a response from the stack tracer and forwards the call trace to the permission checker.

The length of Dalvik call traces are often longer than 1000 bytes, although it depends on the current depth of the call sequence. To reduce the amount of exchanging data for a inter-process stack inspection, FlexDroid assigns a unique integer key to each module specified by each app developer in the manifest. At the initial time, PM determines the mapping between a key and a module. A stack tracer receives part of the mapping related to its app, when it is created. Afterwards, a stack tracer sends only keys of the modules shown in the call trace for inter-process stack inspection.

Although the use of keys reduce the amount of data to exchange, it requires additional costs to find keys corresponding to the modules in the call trace. A naive approach is to compare the name of each module in the call trace with each module name listed in the mapping table. This approach can incur a significant amount of string comparison overheads. We avoid the overheads by caching memory addresses of modules. In other words, each stack tracer creates a table, which maps the address of a module to its key, so that the stack tracer simply compares the address of each module in the call trace to each address in the table.

There are two things worth mentioning in the implementation of the stack transmission channel. First, SELinux (integrated since Android 4.3 (Jelly Bean)) enforces a strict security policy that does not allow any user-space process to access a device driver (e.g., stack transmission channel) by default. We re-configured SELinux to allow all processes to access the channel. Then, to prevent threads that are neither a stack tracer nor a permission checker from intervening in the communication between a permission checker and a stack tracer, the stack transmission channel does not return anything when the caller is not a permission checker and a stack tracer, by checking its PID and TID. Second, since both the permission checkers and the stack tracers access the channel asynchronously through `write` and `read` system calls, it is subject to race conditions. Thereby we enforce synchronized access to the channel and employ a wake queue for waiting and waking up for channel access.

Permission checker. During access control, the permission checkers need to know authorized privileges of each module. Thus at app install time, our modified `PackageParser` parses a manifest of an app (i.e., `flexdroid` tags), and passes permissions for user-space resources to PM and those for kernel-space resources to the kernel, respectively. PM and the kernel keep them as a key-value data structure, where a key is a Java package name of each third-party library, and a value is the granted permission set for the library. Because several apps can have different permission sets for the same library, the key-value storage is allocated to each app. Note that for a finer-grained permission management, FlexDroid allows multiple sub-packages of a single library to have different permission sets.

To mediate access from third-party libraries, we modified system services to check accesses to user-space resources (Android permission model) and to kernel-space resources (GID-controlled ACL routine). For user-space resources, Android's PM uses the `checkUidPermission` method to perform permission checks based on the app's UID. Instead, FlexDroid's PM provides the `checkThreadPermission` method for access control in the granularity of the thread.

To enable this, we modified most Android system services to invoke the `checkThreadPermission` method rather than the `checkUidPermission` method. For kernel-space resources, on the other hand, we modified Linux's ACL to adopt the inter-process stack inspection.

B. JNI Sandbox

As stated in §IV-C, we implement JNI sandbox on FLEXDROID based on hardware-based fault isolation using memory domain supported by ARM architecture. The goal of our implementation is to confine memory access by JNI to the JNI sandbox. At the same time, it ensures that the JNI code should maintain their functionality without any modification. Our implementation mainly consists of a sandbox switch, custom linker, separate heap in JNI sandbox, and Java API wrapper used in JNI.

In ARM architecture, each 1MB virtual memory, called *section*, has a domain ID (from 0 to 15) as a field in its page directory entry. A domain is a collection of memory regions having the same domain ID. Each domain has two bits in the Domain Access Control Register (DACR). According to the values in these bits, attempts to access regions belonging to the domain can: 1) generate a domain fault (`00`); 2) be allowed only if the permissions set in the page table allow (`01`); 3) be allowed regardless of the permissions set in the page table (`11`) [3]. For instance, the system can disallow the code to access a specific domain by setting the corresponding its two bits to 00. Since each core has its own DACR, it is possible to control memory regions that a specific thread can access.

Sandbox switch. FLEXDROID uses domain 3 for JNI sandbox in FLEXDROID, while three domains (0 to 2) are reserved for kernel, user-space (Java domain in FLEXDROID), and device memory, respectively. To assign a domain ID to specific contiguous memory regions, we introduce a `sys_mark_domain` system call which takes the base address and size of the regions that will be used as a JNI sandbox. Assuming that JNIs and shared libraries invoke `malloc` for memory allocation, rather than `mmap`, FLEXDROID assigns 512MB to the JNI sandbox for each process using `mmap` and `sys_mark_domain`. Then, FLEXDROID allocates the stack, heap, text, and data of JNI within the 512MB region.

When a thread in an app executes Java code, FLEXDROID allows the thread to access both the Java and JNI domain by setting its DACR. When the thread executes JNI, FLEXDROID updates DACR to prevent the thread from accessing the Java domain. However, switching DACR alone will lead to a domain fault since the thread can attempt to access memory regions outside the JNI domain such as the stack, heap of the process.

We define the context of sandbox as the combination of DACR, Program Counter (PC), Stack Pointer (SP), and Thread Local Storage (TLS), and we call switching those four registers *sandbox switch*. It is worth noting that we do not need to switch anything for the heap (acquired by `malloc`) and shared libraries, since addresses of functions used in JNI are determined at linking time of JNI, which are separate from the ones used in Java domain.

[3] `10` is reserved.

For sandbox switch, we add two system calls named `sys_jni_enter` and `sys_jni_exit`. Switching to JNI sandbox invokes `sys_jni_enter` to save the current DACR, PC, and SP and change them to new ones. It also updates TLS using `set_tls` system call. Switching to Java domain involves restoring the saved sandbox context. `sys_jni_enter/exit` uses `struct pt_regs*` to modify PC and SP while special ARM instructions (`MCR` and `MRC`) are used to update DACR and TLS.

While JNI is executed, it can call Java APIs through the `JNIEnv` structure to interact with the Java context. Since JNI cannot directly call Java APIs located at the Java domain, an interface between JNI and Java APIs is needed (i.e., trampoline). Also, as mentioned in §IV-C, the validity and type of a pointer passed from JNI as an argument of a Java API should be checked. To enable this, we implement wrappers for all 228 Java APIs, two system calls, `sys_java_enter` and `sys_java_exit`, and Java API handler. Each Java API wrapper invokes `sys_java_enter` system call with the marshalled arguments and the name of the API function, and then the system call activates the Java API handler. The handler validates the pointer arguments, calls the actual Java API, and invokes `sys_java_exit` system call which restores the saved JNI context. (see Java API handler below for details).

During the implementation, we found that a shared library `libjnigraphics.so` in the Android framework layer directly accesses the Java domain without calling Java APIs. It receives an integer from a Java API and casts it to a pointer to access the memory of the address. We implement trampolines for `AndroidBitmap_getInfo`, `AndroidBitmap_lockPixels`, `AndroidBitmap_unlockPixels` of `libjnigraphics.so`.

Since the SP and TLS switch need a stack in the JNI sandbox, FLEXDROID provides an on-demand 1MB stack to each thread running JNI. FLEXDROID supports 64 JNI stacks at maximum, which are managed as a pool. Since JNI stack grows upward from its bottom, FLEXDROID sets the highest address of JNI stack as the base address of JNI's TLS. A stack for a newly created thread in JNI is also allocated from this stack pool.

Right after the sandbox switch, Foreign Function Interface (FFI) is called to invoke the JNI function as JVM usually does. FLEXDROID marshals arguments in JNI's stack that will be passed to FFI.

Custom Linker. We implement a custom linker by modifying the original linker in Android. We add `dlopen_in_jni` in the linker to load JNI code to the JNI sandbox, while Dalvik VM originally loads them using `dlopen`. `dlopen` maps each shared library file (`.so`) to memory region using `mmap` system call. Since `mmap` can allocate the memory region outside the JNI sandbox, `dlopen_in_jni` should not use the `mmap`. Instead, we implement a wrapper of `mmap` to map `.so` files to the memory region inside the JNI sandbox.

The original Android linker manages information of shared libraries loaded by `dlopen` using a linked list for reusability. To reuse shared libraries loaded into the JNI sandbox, our custom linker maintains another linked list which manages information of shared libraries loaded by `dlopen_in_jni`.

Among shared libraries, the `libc` is required to be customized to support the JNI sandbox. The `libc` provides a process with important information for constructing JNI execution environment, including system environment variables and arguments of the created process. Because of this, when initializing the `libc`, FLEXDROID copies the variables and arguments from the Java domain into the JNI sandbox so that the JNI code can access that information.

Heap management in JNI sandbox. FLEXDROID needs to allocate memory space from the heap of the JNI sandbox instead of the default heap. We customize the heap management functions in the `libc` including `malloc`, `calloc`, `realloc`, `free`, and `memalign` using `mspace_malloc()` in `dlmalloc` [34], which enables us to allocate memory from specified memory regions.

Java API handler. Java API handler first unmarshals the passed arguments and verifies the validity of pointer arguments. We implement Valid Address Table (VAT) which maintains a mapping between memory addresses a pointer variable points and the type of the pointer. If the Java function returns a pointer variable, the handler updates VAT with the value and type of pointer.

It is important to note that if the type of return value is a pointer of primitive type (e.g., `char*`, `int*`), and the JNI code attempts to dereference the returned pointer, it causes a domain fault since the dereferenced value is in the Java domain. To resolve this, when a pointer of primitive type is returned, we copy the data pointed by it into a buffer inside the JNI domain and return the address of buffer instead.

In general, marshalling and unmarshalling arguments impose additional memory copy overhead. We avoid the overhead by maintaining the value of registers and stack[4] used for arguments when the calling Java API conducts the sandbox switch.

C. Dynamic Permission Management

Java reflection enables dynamic code generation and execution at runtime. A common procedure involved in dynamic code generation is to store a sequence of bytecode instructions into a Java class file. In order to execute the code generated as such, it needs to load the class file that contains the code. FLEXDROID adjusts the permissions of a module dynamically when the module is loaded. The Dalvik class loader loads a class using `loadClassFromDex()` or a method using `loadMethodFromDex()`. Upon each class or method loading, FLEXDROID looks at the Dalvik call stack to identify which module (caller) loads which other module (callee). Then, FLEXDROID restricts the permissions of the callee module as the intersection of the permissions of the caller and callee modules in order to avoid a potential attack, where the caller module abuses the permissions of the callee module for its own sake.

In addition, FLEXDROID performs dynamic permission management when a new Java thread is created. When a system call `do_fork()` is invoked to create a new kernel thread, FLEXDROID inspects the Dalvik call stack of a parent thread and adjusts the permissions of a module running on the new thread according to the rules described in §IV-D.

[4] In ARM calling convention, the first four arguments are passed through `r0-r3` registers while the rest are passed through the stack.

D. Deadlock Avoidance

There were two critical cases arose causing deadlocks that we faced in the course of developing the inter-process stack inspection.

File access from Garbage Collection. Both Garbage Collection (GC) and stack tracer need to hold the thread lock in the Dalvik VM. If a thread in the GC state accesses a file, the kernel conducts the access control and activates the stack tracer of the same process for access control. The stack tracer waits for the lock already held by the GC. At the same time, GC waits for the access control (i.e. deadlock). To resolve this, FLEXDROID keeps a waiting thread list of the lock using `stl::set`. When inter-process stack inspection of a waiting thread is requested, the stack tracer ignores the request.

Resource access without state change. We can trace the call stack of a thread only when the thread is in “suspended” state. Since a stack tracker is just a normal thread, it cannot actually suspend the target thread. Instead, it just waits until the thread suspends itself. When a thread is in the middle of access to a resource via a system call, Dalvik VM does not know it and keeps its state as “running” (i.e., not suspended). At the same time, the stack tracer just waits for the thread to suspend itself (i.e. deadlock). To avoid this, stack tracer marks the status of the thread as “suspended” when tracing its call stack.

VI. EVALUATION

In this section, we evaluate FLEXDROID by answering the three questions as follows:

1) How flexible and effective is FLEXDROID’s policy to restrict third-party libraries (§VI-A)?
2) How easy is it to adopt FLEXDROID’s policy to existing Android apps (§VI-A)?
3) How much performance overhead does FLEXDROID impose when adopted (§VI-B)?

Experimental setup. All our experiments are performed on Nexus 5 that has 2.265GHz quad-core CPU with 2GB RAM, with our prototype of FLEXDROID on Android 4.4.4 (KitKat) and Linux kernel 3.4.0.

A. Usability

In this evaluation, we examine two aspects of FLEXDROID’s usability:

1) How easily can app developers apply FLEXDROID’s policy to third-party libraries?
2) What experiences do the end-users have running FLEXDROID-protected apps?

Compatibility with existing apps. We downloaded application APK files of 32 apps, which are top apps of various categories listed in App Annie [5], from the Google Play store. In order to test the backward compatibility of FLEXDROID, we installed each app without any modification and ran it for

App Name	Package Name	Category	# of JNI libraries
Bible	com.sirma.mobile.bible.android	Book	0
Job Search	com.indeed.android.jobsearch	Business	0
ZingBox Manga	com.zingbox.manga.view	Cartoon	0
LINE Messenger	jp.naver.line.android	Communication	8
Duolingo	com.duolingo	Education	1
eBay	com.ebay.mobile	Shopping	1
Amazon Shopping	com.amazon.mShop.android.shopping	Shopping	3
Airbnb	com.airbnb.android	Trip	0
Instagram	com.instagram.android	SNS	10
TED	com.ted.android	Education	0
Subway Surf	com.kiloo.subwaysurf	Game	3
NPR News	org.npr.android.news	News	0
Flashlight	com.devuni.flashlight	Utilities	1
K-9 mail	com.fsck.k9	E-mail	1
Fitbit	com.fitbit.FitbitMobile	Health	0
Zillow Real Estate & Rentals	com.zillow.android.zillowmap	Lifestyle	0
musical.ly	com.zhiliaoapp.musically	Media & Video	7
Drugs.com	com.drugscom.app	Medical	0
Yahoo News	com.yahoo.mobile.client.android.yahoo	News & Magazines	2
Yahoo Mail	com.yahoo.mobile.client.android.mail	E-mail	1
Hola launcher	com.hola.launcher	Launcher	4
Layout from Instagram: Collage	com.instagram.layout	Photography	0
Photo Editor by Aviary	com.aviary.android.feather	Photography	2
SquareQuick	mobi.charmer.squarequick	Photography	1
Retrica	com.venticake.retrica	Photography	1
Yelp	com.yelp.android	Local & Travel	1
Pinterest	com.pinterest	Social	0

TABLE V: Compatibility test. Running popular apps on FLEXDROID without applying FLEXDROID policy.

10 minutes in both the stock Android and FLEXDROID, and checked to see if an app crashes during the execution.

Table V shows a list of apps which run as normal in FLEXDROID. Unlike those apps, 5 apps crashed during the execution. They are Waze Social GPS Map & Travel (`com.waze`), Uber (`com.ubercab`), Adobe Acrobat Reader (`com.adobe.reader`), Facebook (`com.facebook.katana`), and UC Browser (`com.UCMobile.intl`). To figure out the cause of the crash, we first disable JNI sandbox (§V-B) and then test those apps again. Since they work fine without JNI sandbox, we conclude the faults stem from the JNI sandbox.

To specify the source of each fault, we capture the fault address and the context (i.e., values of registers and stack information) of the thread at the time the fault occurs using a signal handler registered by Dalvik VM and our domain fault handler in the kernel. Moreover, we manually reverse engineer the JNI code (i.e., `*.so` files) of the crashed apps and pinpoint the locations of the faults based on the captured information. We found out that the roots of the faults are Pthread ID, `mmap()`, and `free()`.

JNI code of Waze fails in the thread safety check which compares the Pthread ID obtained by Java to the one obtained by JNI code. Since both are the return values of `pthread_self()`, they are expected to be the same, but they are different in FLEXDROID, indeed. Pthread ID is an address of a thread structure inside Android `libc`, which is accessed by Pthread APIs. The thread structure initially stays in Java domain, because Dalvik VM, `libc` and other code in Java domain need it to use Pthread APIs. When executing JNI code, FLEXDROID copies the thread structure to JNI domain so as to support Pthread APIs in JNI without domain faults[5]. Consequently, it changes the return value of `pthread_self()` and causes the failure. Separating Pthread ID from the thread structure is a way to avoid this problem, although it requires modification of JNI code.

Uber employs a JNI library SnappyDB [16], which is a key-value database for Android. It maps data stored in a file to memory pages using `mmap()`. Since the memory pages returned by `mmap()` are out of JNI domain, it generates domain faults. Another shared library that calls `mmap()` is `libbinder.so`. Fortunately, since the bulk of apps use Binder only in Java code, most apps do not crash because of `libbinder.so`. Allocating memory to each separate region depending on the caller of `mmap()` is our future work.

While executing JNI code of Adobe Acrobat Reader, `free()` which we implement for JNI domain is called a few thousand times and then incurs a fault. We are currently uncertain as to what makes this fault, however, we believe that we can solve this through engineering.

Since Facebook and UC Browser contain many JNI libraries (29 and 20, respectively), it is challenging to manually reverse engineer them and understand the roles of their instructions. Moreover, they show complicated runtime behaviors such as multi-threading. Due to this, we cannot reveal the exact reasons that cause crashes of both apps.

Usability and effectiveness. We evaluate the usability of FLEXDROID's policy and its effectiveness. We conduct experiments using a simple app that we implement and real-world apps repackaged with FLEXDROID's policy. A component of our simple app accesses several resources with corresponding permissions. We eliminate each permission by enforcing FLEXDROID's policy and verify that a security exception occurs according to the absence of permission.

[5] If FLEXDROID moves the thread structure to JNI domain instead of copying it to JNI domain, the thread structure should be shared between Java domain and JNI domain. It raises a security concern that the thread structure can be tampered by untrusted native code (i.e., JNI) in malicious purposes.

Target Third-party Library	Role	App Name	Blocked Resource
`com.google.ads.*`[†]	Ad	ZingBox Manga	Internet
`jp.naver.line.*`[‡]	Photo	LINE Messenger	Camera
`com.ebay.redlasersdk.*`	Barcode scanner	eBay	Camera
`com.facebook.*`[†]	Login	Airbnb	Internet
`com.tapjoy.*`	Ad	Subway Surf	Internet
`com.twitter.*`[†]	Login	Drugs.com	Internet
`com.android.volley.*`	HTTP	Yahoo News	Internet
`com.flurry.*`[†]	Analytics	Yahoo Mail	Internet

† Used in two or more apps
‡ A component of the app, not a third-party library

TABLE VI: Enforcing the FLEXDROID's policy against third-party libraries of real-world apps.

Use scenario	Android	FLEXDROID	Over.
Launch an application*	39.13 ms	39.73 ms	1.55%
Launch a service	3.76 ms	3.95 ms	5.22%
Download 1.3MB file	136.54 ms	139.59 ms	2.24%
Take a photo	443.01 ms	448.99 ms	1.35%
Send an email*	100.56 ms	101.70 ms	1.13%
Read 8.4MB file via JNI	88.71 ms	89.16 ms	0.51%

∗ Functionalities of open-source K-9 email app

TABLE VII: Performance overheads of real use scenario on both FLEXDROID and the stock Android. FLEXDROID imposes 0.51%-5.22% overheads depending on the type of task.

We select 8 third-party libraries from 8 real-world apps and apply FLEXDROID's policy to them. We just decompress the 8 APK files, append `flexdroid` tags to their `AndroidManifest.xml` files, and repackage them. It is worth noting that a simple modification to `AndroidManifest.xml` file is enough to enforce FLEXDROID's policy without any knowledge of the app. In other words, it is easy for app developers to adopt FLEXDROID. We manually analyze each third-party library to specify one of its roles and eliminate all its permissions using FLEXDROID's policy. As shown in Table VI, each of the 8 libraries have a problem accessing a resource. It implies that FLEXDROID's policy is effectively enforced.

Case study with a real-world app. To demonstrate how FLEXDROID works with existing apps, we used Yahoo News [18] which is a popular news app registered on Google Play. We chose this app for our case study, since the app uses various third-party analytics, social, UI, and video libraries such as Flurry, Facebook, NineOldAndroids [12], and MP4Parser [11]. Among the libraries, we particularly focus on Flurry, which accesses a device's IMEI code using undocumented `READ_PHONE_STATE` permission. Flurry prints IMEI code as a log message that we can observe.

We repackage the APK file to take away the `READ_PHONE_STATE` permission from Flurry. We also set the `android:mockOnException` attribute to `true` to provide Flurry with a fake IMEI code. We check out that Flurry outputs log messages related to the fake IMEI code. This indicates that FLEXDROID does not degrade usability in the view of a user while FLEXDROID successfully prevents the privacy-sensitive information from being leaked.

B. Performance

We evaluate the performance impact of enabling FLEXDROID's features in comparison to stock Android. The overhead of FLEXDROID mainly comes from two potential sources. One is from the inter-process stack inspection that FLEXDROID conducts upon each access request, and the other is from a sandbox switch that FLEXDROID does upon every JNI execution and Java API calls from JNI. For the inter-process stack inspection overhead, we focus on situations where an app attempts to access various resources, while measuring the delay of JNI execution and Java API call for sandbox switch. We note that FLEXDROID introduces an additional thread, named stack tracer, for each app. The stack sleeps all the time, thereby leaving negligible impacts on performance, except when its app makes access requests.

In order to minimize the effect of unrelated processes (e.g., system daemons), we turn on all cores of CPU and fix their frequencies to the maximum values for all experiments. Besides, we assign the highest priorities to all the threads running our benchmarks. We repeat each experiment case 50 times on both stock Android and FLEXDROID, and choose a median value for comparison.

We conduct macro-benchmarks to examine overheads in common use scenarios and micro-benchmarks to inspect the details.

1) Benchmarks for Common Use Scenarios: We measure the performance overheads imposed by FLEXDROID with our custom benchmarks to simulate end-user scenarios commonly seen on smartphones, as summarized in Table VII. For ease of modification and experiment, we use a popular open-source email app K-9 [10]. In the following, we explain how we measure the elapsed time for each case.

Launch an application. The application launch time measures from when `startActivity()` is called and the time the app becomes visible. This time includes relatively slow operations (e.g., process creation, IPCs between system services, and I/O operations for supporting GUI). FLEXDROID imposes an overhead of only 1.6%.

Launch a service. We build a custom Android service to measure the service launch time. Similar to the app's launch, we measure from when `startService()` is called to the time the service routine is started. FLEXDROID adds an overhead of 5.22%, which is relatively larger though the actual timespan is quite small.

Download an image. Using our custom app, we measure the elapsed time to download a 1.3MB image from the web and to store it to an SD card. Although it takes several hundreds of milliseconds, FLEXDROID adds an overhead of 2.24%, as just a few permission checks occur.

Take a photo. We measure the elapsed time for taking a photo using the camera benchmark. It measures from when the shutter button is clicked to the time the image is stored. FLEXDROID has an overhead of 1.35%, since the time includes heavy operations such as capturing raw images from camera and compressing it.

Benchmark	Android	FlexDroid	Over.
`startActivity()`	3,935 μs	4,529 μs	594 μs
`startService()`	1,221 μs	1,734 μs	513 μs
file open*	782 μs	1,657 μs	875 μs
file open (create)*	1,390 μs	2,338 μs	948 μs
file delete	745 μs	1,330 μs	585 μs
file read†	138 μs	142 μs	4 μs
file write†	1,076 μs	1,134 μs	58 μs
call JNI method	97 μs	186 μs	89 μs
call JNI method after loading libs‡	963 μs	8,436 μs	7,473 μs

$*$ Two stack inspections are required during a file open
† No stack inspection is required during file read and write
‡ This includes the process of loading (and dynamic linking) the JNI code and shared libraries needed by the JNI code

TABLE VIII: Micro-benchmarks of starting activity/service, file-related system calls and JNI method invocation. FlexDroid imposes 4-948 μs overheads depending on the number of stack inspection calls while calling JNI method, including loading and dynamic linking, increases the delay by 7,473 μs.

Send an email. In K-9 app, when we press the `Compose` button, a background task for sending an email is asynchronously executed. Thereby we measure the email sending time as the execution time of the background task. Since the task is heavy, our system adds only 1.13% overhead.

Read a file via JNI. We invoke a native method which reads data from a 8.4MB file and measure the time to finish its execution. FlexDroid imposes an overhead of only 0.51%, which is imperceptible.

2) Micro-benchmarks: We build a custom benchmark app to measure overheads from inter-process stack inspection for accessing user-space and kernel-space resources (e.g., Activity, Service, File) and the expense of calling JNI method. For user-space resources, we measure the execution times of `startActivity()` and `startService()`. Table VIII shows the overheads that are mainly caused by inter-process stack inspection.

For kernel-space resources, we evaluate the overheads regarding file operations. Our benchmark app accesses a file located in the external storage by invoking open, read, write, and close system calls, which need `READ/WRITE_EXTERNAL_STORAGE` permissions. As shown in Table VIII, we can see that opening a file has approximately twice the delay of the one for file deletion. This is because opening a file requires executing the inter-process stack inspection twice, while the file deletion needs to execute only once. File read and write have a much lower overhead since they do not involve the inter-process stack inspection.

Our benchmark app invokes a native method that simply prints a log message. When Dalvik VM calls a JNI method for the first time, it conducts loading and dynamic linking the JNI code and shared libraries needed by the JNI code. After this, calling the same JNI method again skips the process of loading and dynamic linking to shorten the delay. It is worth mentioning that the Android linker keeps loaded shared libraries to avoid loading the same shared library again. Since the linker for JNI domain in FlexDroid loads shared libraries that are already loaded in Java domain, the first delay of calling a JNI method must be much longer. This overhead, however, does not last after the first call. Depicted in Table VIII, the first JNI method call introduces an overhead of 963 μs on stock Android, and FlexDroid incurs an additional overhead of 7,473 μs. The overhead mainly comes from additional loading and dynamic linking. Though it is very large, the later JNI method calls, fortunately, impose only 89 μs overhead. The additional expenses of the later JNI method calls stem from the sandbox switch.

VII. Discussion

Currently, FlexDroid has a weakness in terms of backward compatibility which is caused by JNI sandbox. As specified in §VI-A, Pthread ID, `mmap()`, and `free()` are known sources of faults when executing JNI code in FlexDroid. In particular, supporting `mmap()` requires changing `mmap()` system call implementation inside the kernel. It must distinguish memory pages that will be allocated in JNI domain from the one allocated in Java domain, so that FlexDroid can allocate memory to each domain depending on the caller of `mmap()` and avoid domain faults. Developing the JNI sandbox, which provides complete backward compatibility, would be a meaningful future work.

FlexDroid does not provide memory isolation between third-party libraries. What this means is that malicious third-party libraries can use native code to access or overwrite the memory of other third-party libraries running on its host app. Establishing memory isolation between third-party libraries is another significant future work.

Since most of the third-party libraries are provided with a list of necessary permissions as a plain text or a part of the manifest code, an app developer can easily enforce FlexDroid's policy with the given information.

App developers often protect apps' source code with ProGuard [15] which optimizes and obfuscates the source code. Since ProGuard obfuscates package names, which is given to FlexDroid rule, FlexDroid may not work effectively with an app protected by ProGuard. To overcome this limitation, we suggest that app developers configure ProGuard to exclude suspicious libraries, which is easily done by adding a few lines in the configuration file (proguard.config).

VIII. Conclusion

In this paper, we present FlexDroid, an extension to Android's permission system, which provides dynamic, fine-grained access control for third-party libraries. FlexDroid enables app developers to fully control capabilities of third-party libraries (e.g., permissions to be granted), and specify behaviors after unauthorized resource access attempts (e.g., quitting the app or sending dummy data). From our analysis of 100,000 Android apps, 17.1% of 295 third-party libraries are found to utilize JNI, and at least 27.9% of them are observed to use dynamic code generation. Considering that these are two key factors hindering the host app and their third-party libraries from drawing clear and trustworthy boundaries at runtime, FlexDroid provides a novel in-app privilege isolation mechanism with inter-process stack inspection, which is effective to JNI as well as dynamic code generation.

Our usability and compatibility experiments with 32 popular Android apps show that app developers can easily adopt FLEXDROID's policy to third-party libraries without any code modification except the manifest. Also, our evaluation shows that FLEXDROID successfully regulates resource access of third-party libraries with imperceptible performance overheads.

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd, David Lie, for their helpful feedback. This work was supported in part by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government(MSIP) (No.10041313, UX-oriented Mobile SW Platform). Taesoo Kim was supported in part by the NSF award (DGE-1500084), by the ONR grant (N00014-15-1-2162), and by the DARPA Transparent Computing program under contract No. DARPA-15-15-TC-FP-006.

REFERENCES

[1] acl(5) Linux man page. http://linux.die.net/man/5/acl.
[2] Actionbarsherlock Android SDK. http://actionbarsherlock.com/.
[3] Adobe Pdf Library SDK. http://www.adobe.com/devnet/pdf/library.html.
[4] android-apktool: A tool for reverse engineering Android apk files. https://code.google.com/p/android-apktool/.
[5] App annie. https://www.appannie.com.
[6] Dropbox Android SDK. https://www.dropbox.com/developers/core/sdks/android.
[7] FFMPEG. https://www.ffmpeg.org/.
[8] Google Analytics Android SDK. https://developers.google.com/analytics/devguides.
[9] Firm fined for angry birds mobile billing scam. http://ipkonfig.com/firm-fined-for-angry-birds-mobile-billing-scam.
[10] K-9 mail. https://en.wikipedia.org/wiki/K-9_Mail.
[11] Mp4parser. https://code.google.com/p/mp4parser/.
[12] Nineoldandroids. http://nineoldandroids.com/.
[13] Open Rich Media Mobile Advertising. https://code.google.com/p/ormma/.
[14] Paypal Android SDK. https://developer.paypal.com.
[15] Proguard. http://developer.android.com/tools/help/proguard.html.
[16] Snappydb. https://github.com/nhachicha/SnappyDB.
[17] Unity3d SDK. http://docs.unity3d.com/Manual/android-sdksetup.html.
[18] Yahoo news. https://play.google.com/store/apps/details?id=com.yahoo.mobile.client.android.yahoo.
[19] 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.
[20] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, 2011.
[21] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: Driving apps to test the security of third-party components. In *23rd USENIX Security Symposium*, Aug. 2014.
[22] T. Book, A. Pridgen, and D. S. Wallach. Longitudinal analysis of android ad library permissions. *arXiv preprint arXiv:1303.0857*, 2013.
[23] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, 2011. URL http://www.trust.informatik.tu-darmstadt.de/fileadmin/user_upload/Group_TRUST/PubsPDF/xmandroid.pdf.
[24] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, 2012.
[25] J. Crussell, R. Stevens, and H. Chen. Madfraud: Investigating ad fraud in android applications. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, 2014.
[26] 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*, Aug. 2011.
[27] 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.
[28] F-Secure. Mobile Threat Report Q3 2013. https://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q3_2013.pdf.
[29] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.
[30] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and enhancing androids permission system. In *17th European Symposium on Research in Computer Security*, 2012.
[31] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. 1996.
[32] M. C. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2012.
[33] 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.
[34] D. Lea. Dlmalloc, 2010. URL http://g.oswego.edu/dl/html/malloc.html.
[35] B. Livshits and J. Jung. Automatic mediation of privacy-sensitive resource access in smartphone applications. In *Presented as part of the 22nd USENIX Security Symposium*, 2013.
[36] G. McGraw and E. W. Felten. *Securing Java: Getting Down to Business with Mobile Code*. John Wiley & Sons, Inc., New York, NY, USA, 1999. ISBN 0-471-31952-X.
[37] 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.
[38] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *Presented as part of the 22nd USENIX Security Symposium*, 2013.
[39] S. Shekhar, M. Dietz, and D. S. Wallach. Adsplit: Separating smartphone advertising from applications. In *Presented as part of the 21st USENIX Security Symposium*, 2012.
[40] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in android ad libraries. In *IEEE MOST 2012*, 2012.
[41] M. Sun and G. Tan. NativeGuard: Protecting android applications from third-party native libraries. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, 2014.
[42] N. Viennot, E. Garcia, and J. Nieh. A measurement study of google play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems*, 2014.
[43] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce component-level access control in android. In *4th ACM conference on Data and application security and privacy*, 2014.
[44] L. K. Yan and H. Yin. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Presented as part of the 21st USENIX Security Symposium*, 2012.
[45] X. Zhang, A. Ahlawat, and W. Du. Aframe: Isolating advertisements from mobile applications in android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.
[46] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 558–569, 2014. ISBN 978-1-4503-2957-6.
[47] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang. Hybrid user-level sandboxing of third-party android apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, pages 19–30, 2015. ISBN 978-1-4503-3245-3.