# FineDroid: Enforcing Permissions with System-wide Application Execution Context

Yuan Zhang[1,2], Min Yang[1,2], Guofei Gu[3], and Hao Chen[4]

[1] School of Computer Science, Fudan University
[2] Shanghai Key Laboratory of Data Science, Fudan University,
{yuanxzhang,m_yang}@fudan.edu.cn
[3] SUCCESS Lab, Teax A&M University, guofei@cse.tamu.edu
[4] University of California, Davis, chen@ucdavis.edu

**Abstract.** To protect sensitive resources from unauthorized use, modern mobile systems, such as Android and iOS, design a permission-based access control model. However, current model could not enforce *fine-grained* control over the dynamic permission use contexts, causing two severe security problems. First, any code package in an application could use the granted permissions, inducing attackers to embed malicious payloads into benign apps. Second, the permissions granted to a benign application may be utilized by an attacker through vulnerable application interactions. Although ad hoc solutions have been proposed, none could systematically solve these two issues within a unified framework.
This paper presents the first such framework to provide context-sensitive permission enforcement that regulates permission use policies according to system-wide application contexts, which cover both *intra-application context* and *inter-application context*. We build a prototype system on Android, named *FineDroid*, to track such context during the application execution. To flexibly regulate context-sensitive permission rules, FineDroid features a policy framework that could express generic application contexts. We demonstrate the benefits of FineDroid by instantiating several security extensions based on the policy framework, for two potential users: administrators and developers. Furthermore, FineDroid is showed to introduce a minor overhead.

**Key words:** permission enforcement, application context, policy framework

## 1 Introduction

Modern mobile systems such as Android, iOS design a permission-based access control model to protect sensitive resources from unauthorized use. In this model, the accesses to protected resources without granted permissions would be denied by the permission enforcement system. Ideally, the permission model should prevent malicious applications from abusing sensitive resources. However, the current permission model could not enforce a fine-grained control over permission use contexts (in this paper, when we say context we mean the application execution context). As a result, malicious entities could easily abuse permissions, leading to the explosion of Android malware these years [9] and the numerous reported application vulnerabilities [21, 36].

Since Android has been expanding its market share rapidly as the most popular mobile platform [5], this paper mainly focuses on the permission model of Android. Currently, the coarse-grained permission enforcement mechanism is limited in the following two aspects.

- **Intra-application Context Insensitive.** Current permission model treats each application as a separate principal and permissions are granted at the granularity of application, thus all the code packages in the application could access the protected resources with the same granted permissions. In fact, not all the code packages in a single application come from a same origin.
- **Inter-application Context Insensitive.** Application interaction is a common characteristic of mobile applications. However, this new characteristic is transparent to the current coarse-grained permission enforcement mechanism, exposing a new attack surface, i.e., the permissions granted to a vulnerable application may be abused by an attacker application via inter-application communication.

Given these problems, plenty of extensions have been proposed to refine the Android permission model. Dr. Android and Mr. Hide framework [23] provides fine-grained semantics for serval permissions by adding a mediation layer. SEAndroid [33] hardens the permission enforcement system by introducing SELinux extensions to the Android middleware. FlaskDroid [15] extends the scope of current permission system by regulating resource accesses in Linux kernel and Android framework together within a unified policy language. Context-aware permission models [17, 26, 30, 32] are proposed to support different permission policies according to external contexts, such as location, time of the day. However, these works still could not address the two limitations described above. There are also some work dedicated to reduce the risk of inter-application communication [12–14, 18, 20, 26] or to isolate untrusted components inside an application [27, 31, 35, 39]. However, none could achieve unified and flexible control according to the system-wide application context.

In this paper, we seek to fill the gap by bringing context-sensitive permission enforcement. We design a prototype, called FineDroid to provide fine-grained permission control over the application context. For example, if app `A` is allowed to use `SEND_SMS` permission in the context `C`, when app `A` requests `SEND_SMS` permission in another context `C'`, it would be treated as a different request of `SEND_SMS` permission. In FineDroid, we consider both the *intra-application context* which represents the internal execution context of an application, and the *inter-application context* which reflects the IPC context of interacted applications. It is non-trivial to track such context in Android. FineDroid designs several techniques to automatically track such contexts along with the application execution. To ease the administration of permission control policies, FineDroid also features a policy framework which is general enough to express the rules for handling permission requests in a context-sensitive manner.

To demonstrate the benefits of FineDroid, we create two security extensions for administrators and developers. First, since permission leak vulnerability [20, 21, 24, 36] is very common and dangerous, we show how administrators could benefit from our system in transparently fixing these vulnerabilities without modifying vulnerable applications. Second, we provide application developers with the ability of restricting untrusted third-party SDK by declaring fine-grained permission specifications in the manifest file. All these security extensions can be easily built using policies.

We evaluate the effectiveness of our framework by measuring the effectiveness of the developed security extensions. For administrators, we show that FineDroid can easily fix permission leak vulnerabilities with context-sensitive permission control policies, and the policies could even be automatically generated by a vulnerability detector. For developers, we show that just several policies are enough to restrict the permissions that could be used by untrusted SDKs. It is worth noting that our system is not limited to support these two extensions. In addition, our system is showed to introduce minor performance overhead (less than 2%).

In this paper, we make the following contributions.

- We propose context-sensitive permission enforcement to deal with severe security problems of mobile systems. Considering the characteristics of mobile applications, it is important and necessary to take the application context into account when regulating permission requests.
- We design a novel context tracking technique to track *intra-application context* and *inter-application context* during the application execution.
- We design a new policy framework to flexibly and generally regulate permission requests with respect to the fine-grained application context.
- We demonstrate two security extensions based on the context-sensitive permission enforcement system, by just writing policies and sometimes a small number of auxiliary code.
- We evaluate the security benefits gained by the two security extensions and report the performance overhead.

## 2 Threat Model

This paper considers a strong threat model in which an attacker aims to gain and abuse sensitive resources stealthily. More specifically, this paper assumes an attacker could launch all kinds of *application-level* attacks, while the Linux kernel and `Android Runtime` are secure (not compromised). For the stealthiness, we mean an attacker tries to hide its identity in using permissions from the permission enforcement system. We consider these two kinds of attacks.

**Intra-application Attack.** To hide the behavior of abusing permissions, an attacker could inject malicious payloads into a benign application (either before installation or during runtime). There are several ways for an attacker to infect benign apps. First, an attacker could actively embeds malicious payloads into popular benign apps and redistributes the repackaged version via third-party application markets. Second, an attacker could exploit code injection vulnerabilities (such as Man-in-the-Middle attack with dynamic class loading [28]) to inject malicious payloads. In addition, an attacker could also publish malicious SDKs, passively waiting for developers to include [1].

**Inter-application Attack.** The prevalent application interaction in the Android programming model may also be used by attackers to stealthily use permissions. This kind of attack has been verified in several forms, such as capability leak [20, 21, 36], component hijacking [24], content leak and pollution [41]. In these attacks, the

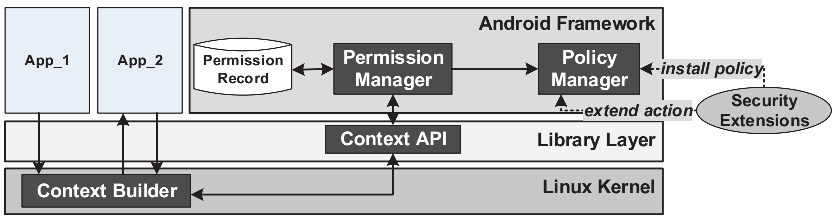

*Fig. 1:* Architecture of Context-Sensitive Permission Enforcement Framework.

permission enforcement system would see a permission request from a victim app which has a legitimate requirement for this privileged resource, while actually this permission is originally requested and utilized by an attacker app.

Note that our threat model does not consider other kinds of attacks such as privacy stealing, root exploits and colluding attacks, because they are not caused by the context-insensitive permission enforcement mechanism and have been well addressed by previous work [12, 13, 15, 19, 22].

## 3 Approach Overview

To defeat these attacks, we propose *context-sensitive* permission enforcement. The key idea is to construct a system-wide application context for each permission request and make granting decisions based on this context. Since the permission enforcement system could catch all the code packages and all the apps that participate in the permission request, an attacker could no longer stealthily abuse permissions.

The system-wide application context is composed of two parts: (1) **Intra-application Context** which represents the internal execution flow of an application, and (2) **Inter-application Context** which reflects the interaction flow among applications and system services. With these two kinds of contexts, our framework could accurately distinguish permission requests originated from different sources, thus achieving a fine-grained control over permission usage.

The overall architecture of FineDroid is presented in Figure 1. The rectangles filled with black color are new modules introduced by FineDroid. The core of our framework is the *Context Builder* module, which automatically tracks the application context along with the application execution. This module is placed in the Linux Kernel, so an attacker cannot escape from the context tracking. We also provide *Context API* at the library layer for applications and the Android framework to obtain the current application context from the *Context Builder* module.

Based on *Context API*, we design a context-sensitive permission enforcement system. To flexibly set context-sensitive permission control rules, FineDroid features a generic policy language. In FineDroid, all permission requests are intercepted by the *Permission Manager* module. To handle a permission request, the *Policy Manager* module examines all the polices in the system, and then *Permission Manager* could make a permission decision according to the action (e.g. allow or deny) specified in the match policy. Besides, our policy language is extensible for introducing new permission handling actions. To support building security extensions atop the policy framework, *Policy Manager* provides open interfaces for policy management and extension.

Next, we will detail the design of FineDroid. The application context tracking technique is presented in Section 4, and we describe the context-sensitive permission enforcement system in Section 5.

## 4 Application Context Tracking

Application context is the cornerstone of FineDroid, while it is not a primitive element yet in the Android system. Thus, we design *Context Builder* to automatically build the two kinds of application contexts. To prevent attackers from hiding their identities in the application context, we place the *Context Builder* in the Linux Kernel. However, the complexity of the Android programming model brings huge challenges in propagating application context along with the application execution. To deal with these complexities, we further introduce several techniques for context propagating. Next, we elaborate these techniques.

### 4.1 Intra-application Context Builder

*Intra-application context* is used to distinguish different execution flows inside an app. In FineDroid, the function calling context is used to abstract the internal execution context inside an app. However, it is too large to efficiently propagate and compare the complete calling context. Thus, we need to efficiently compute a birthmark for any given calling context.

**PCC as Intra Context.** We adopt a technique called *probabilistic calling context (PCC)* [11] to compute an integer birthmark based on all the functions in the flow. PCC can be efficiently calculated with a recursive expression $pcc = 3 * pcc' + cs$ where $pcc'$ is the PCC value of the caller and $cs$ is a birthmark for the current call site. By applying this expression recursively from the leaf function on the stack to the root function, we could finally obtain a PCC value as the birthmark for the whole calling context. Note that PCC calculation is deterministic which means a given calling context would always get the same PCC value. As evaluated in millions of unique calling contexts [11], PCC is efficient and accurate for bug detection and intrusion detection in deployed software. Thus, PCC is very suitable to represent the internal execution context inside an app.

**Call Site Birthmark.** Since all Java code in an Android app is packed into a single DEX file, we use the relative offset of a call site in the DEX file as the birthmark of the call site ($cs$ value). While at the first glance this solution may encounter problems with native code execution, it turns out that this solution could still calculate a PCC value for the Java functions invoked before the native code because native code could only be invoked from Java functions through Java Native Interface. It is worth noting that our solution does not need to calculate a PCC value for every function invocation. Instead, it just needs to compute PCC values for a small portion of calling contexts inside an application that may participate in a permission request, such as application interaction.

**Implementation Issue.** Since Java functions are executed in a dedicated Java stack by Dalvik virtual machine, *Context Builder* which lies in the Linux Kernel cannot

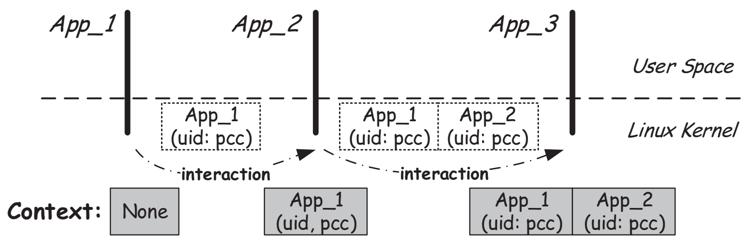

*Fig. 2:* Binder IPC Context Building.

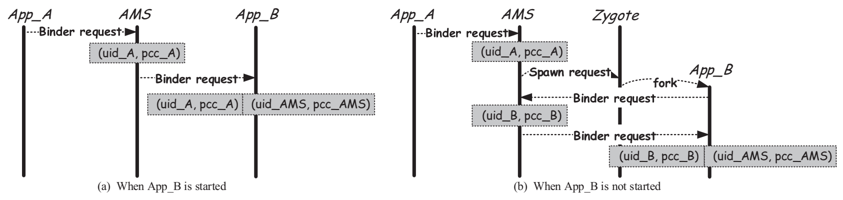

*Fig. 3:* Binder IPC Propagating Diagram in Component Interaction from App_A to App_B.

recognize the user-space Java stack. To solve this problem, we instrument Dalvik virtual machine to register the base address of Java stack to the kernel whenever a Java thread is spawned. Thus, when *Context Builder* needs to calculate the PCC value for the current context, it could traverse all the Java functions in the execution flow by reconstructing the calling stack with the base Java stack address.

## 4.2 Inter-application Context Builder

*Inter-application context* reflects the IPC context among interacted applications. Since Binder IPC is the only way for an application to interact with other applications and system services, *Context Builder* extends Binder kernel module to keep the whole IPC call chain for every IPC invocation. As showed in Figure 2, the extended Binder driver allocates an array for each thread to record the application context in handling Binder communication. During each Binder IPC interaction, the driver would append caller's identity into caller's application context, and propagate it to the callee application as the callee's application context. The caller's identity is composed of two parts: assigned UID of the caller application and PCC value for the *intra-application context* inside the caller application when this interaction occurs.

## 4.3 Context Propagating

Due to some unique features of Android, the built system-wide application context would be lost during normal execution. Thus, FineDroid further retrofits the `Android Runtime` which manages the application execution to propagate application context during the following interaction behaviors.

**Component-level Propagating.** Component interaction is prevalent in Android apps. To initiate a component interaction, an application (named as `A` ) first needs to send an Intent to the ActivitManagerService (referenced as AMS for short), then AMS would choose a target application (named as B ) and route the Intent to B. Figure 3 (a) illustrates this process. Since the invocations from A to AMS and from AMS to B are all proceeded with Binder IPC, app B would get the application context as [$(uid_A, pcc_A)$, $(uid_{AMS}, pcc_{AMS})$] when receiving this Intent. During the component interaction, AMS plays as a mediator between the sender and the receiver. However, from the application context propagated to app B, AMS looks like a participator which is contrary to its actual role.

The problem would be even worse when the target application B has not been launched at the time of Intent delivery. Figure 3 (b) illustrates this scenario. When app B is chosen as the callee of this component interaction and AMS finds that app B has not been started. AMS would delay the Intent routing and notify Zygote (which is the application incubator in Android) to spawn a new process for app B. When B has been started, it would notify AMS and AMS would send the delayed Intent to B. The problem is that the Intent delivery from AMS to app B is performed in the context of receiving the start notification of app B, so the application context propagated to app B is [$(uid_B, pcc_B)$, $(uid_{AMS}, pcc_{AMS})$]. This problem is caused by that the application context for sending the Intent from app A to AMS has not been recovered in delivering the Intent from AMS to app B.

To solve the two problems, we design Intent-based component interaction tracking. The basic idea is that, we instrument AMS to annotate each Intent object with the sender's context, thus the context is propagated to the receiver together with the Intent object. When Android Runtime in the receiver application gets the Intent object from AMS, it first recovers the application context recorded in the Intent object and then triggers the invocation of the target component. Thus, the target component can be executed with the right application context. Note that the application context recovery in the receiver application is guaranteed by our instrumented Android Runtime, thus it could not be escaped.

**Thread-level Propagating.** In each Android Runtime, there is a main thread to handle the component interactions with the system and dispatch UI events (so this thread is also known as UI thread). To reduce the latency of main thread in processing events, developers are advised to delegate time-consuming operations to worker threads. Android designs Message [4], Handler [3], AsyncTask [2] interfaces for developers to facilitate such workload migration and synchronization. However, since thread interaction is not proceeded via Binder IPC, the application context would be lost in the worker thread.

We design two countermeasures to propagate application contexts among thread interactions. First, during *thread creation*, we instrument the thread creation and initialization logic to propagate the application context of the creator thread to the new created thread and then recover the application context before the created thread is ready to run. Second, for *thread interaction*, we consider the message-based interaction mechanism in Android. Before a message is sent to a thread, the application context of the current thread is annotated to the Message object. Then before the target thread handles the Message, its application context is restored according to the one encapsulated in the Message object. It is worth noting that, our thread-level context tracking is transparently performed by our instrumented `Android Runtime`. Thus, this kind of tracking is mandatory without relying on any modification to the applications or cooperation with developers.

**Event-level Propagating.** Callbacks are commonly used in Android to monitor system events. A typical use case is UI event handling. However, the event-based programming model also brings problems to application context tracking, because a callback may be executed in a future time by a thread which would have a different context to the one when the callback is registered. To deal with this problem, FineDroid annotates each callback with the application context when it is registered and recover the application context from the callback before it is triggered for execution. From Android documentation, we find more than 100 APIs that would register callbacks. We instrument each API to embed the registered callback into a wrapper which automatically records and recovers the context to/from the callback. Since only Android APIs are instrumented, this technique is also enforced transparently to the app.

## 5 Context-Sensitive Permission System

Based on the constructed system-wide application context, permission requests in FineDroid could be handled separately according to the concrete application context. To ease the regulation of permissions requests, FineDroid features a policy framework. Next, this framework is introduced in two parts.

### 5.1 Permission Manager

*Permission Manager* first needs to intercept all permission requests. As introduced in [35,40], two kinds of permission requests are intercepted: For KEPs (Kernel Enforced Permissions), we instrument the UID/GID isolation modules in the Linux Kernel to intercept all KEP permission requests and redirect them to *Permission Manager* in the Android framework for handling; For AEPs (Android Enforced Permissions), we instrument *PermissionController* service to redirect all permission requests to the *Permission Manager*.

To handle a permission request, *Permission Manager* first queries *Policy Manager* to select a policy which best matches the current application context. If no policy matches, *Permission Manager* would fall back to the original permission enforcement mode. In the original mode, permission requests are handled by querying the *Permission Record* (see Figure 1) to grant all the permissions declared in the application manifest file. When a matched policy is selected for the current permission request, *Permission Manager* just needs to follow the action (e.g. allow or deny) specified in the policy.

### 5.2 Policy Framework

FineDroid designs a declarative policy language to express the rules for handling permission requests in a context-sensitive manner. Basically, it states the handling action for a permission request from an app within a specified application context. Our policy is structured in XML format, with the following tags. (A sample policy can be found in Figure 4.)

– **policy** tag. It is the root tag for specifying a policy. Three attributes are required to designate the handling action (*action* attribute) when an app (*app* attribute) requests some permission (*permission* attribute). The expected application context for this policy can be figured by either a *context* attribute or child tags described below.
– **uid-selector** tag. It describes the composition relationship of several **uid-context** child tags. The *selector* attribute is mandatory to describe the composition relationship among the child tags. It supports 5 kinds of selectors: *"contains"*, *"startwith"*, *"endwith"*, *"strictcontains"* and *"fullymatch"*.
– **uid-context** tag. It describes context information for a single application participated in the inter-application communication. The *uid* attribute is required to specify the identity of the application. Package name can also be used as the identity of the application. If the value of *uid* attribute begins with "∧", it represents any application except the one specified by the *uid* attribute. The intra-application context of the application can be described by either the *pcc* attribute using the exact PCC value of the application, or detailed function call context information using a child **pcc-selector** tag.
– **pcc-selector** tag. It describes the composition relationship of several **method-sig** child tags. Just like **uid-selector** tag, it requires a *selector* attribute which also supports 5 selectors.
– **method-sig** tag. It describes the signature for a method invoked in the calling context. Three attributes can be used for description: *className*, *methodName*, and *methodProto*.
– **or**, **and**, **not** tag. They describe the logic relationships among child tags. They are used to depict complex contexts which may be difficult to expressed only with **uid-selector** and **pcc-selector**. Meanwhile, these tags can be nested together.

Besides, the policy language supports using "*" as the wild card character in some attributes, such as *context* attribute in **policy** tag, *pcc* attribute in **uid-context** tag.

**Policy Matching.** To test whether a policy could match a permission request, *Policy Manager* first checks the requested permission and the requestor application. When both attributes match, *Policy Manager* further compares the application context. The application context matching is relatively slow, so we use a cache to remember the context matching results. If multiple policies are found to match, *Policy Manager* would select the one that express the most fine-grained application context. *Policy Manager* also supports adding and removing policies to/from the system, as well as registering new action types to extend the policy language. The next section will show how these policies can be used to refine current permission model.

## 6 Security Extensions

To demonstrate the effectiveness of context-sensitive permission enforcement, we create security extensions for administrators and developers. All these extensions are built upon the interfaces exposed by *Policy Manager*, without modifying other FineDroid modules.

```
<policy action="deny" app="com.android.mms" permission="SEND_SMS" >
  <uid-selector selector="strictcontains" >
    <uid-context uid="^com.android.mms" pcc="*" />
    <uid-context uid="com.android.mms" />
      <pcc-selector selector="contains" >
        <method-sig className="com.android.mms.transaction.SmsReceiver"
                    methodName="beginStartingService" />
      </pcc-selector>
    </uid-context>
  </uid-selector>
</policy>
```

*Fig. 4:* Policy to fix `SEND_SMS` permission leak in *SmsReceiver*.

## 6.1 For Administrator: Fixing Permission Leak Vulnerability

In the Android programming model, if a public component is not protected well, it may be misused to perform privileged actions by an attacker application. As demonstrated in [21, 24, 36], many high-risk permissions, such as `SEND_SMS`, `RECORD_AUDIO` are found to be leaked in pre-installed apps and third-party apps. Next, we introduce how to use FineDroid to prevent permission leaks. Note that we do not want to prevent all kinds of component hijacking vulnerabilities, such as information leaks.

**Leak Causes.** There are two possible cases for the permission leak vulnerability. The first case is that some application-private components are mistakenly made publicly accessible. This may be caused by developer's lack of security awareness or insecure code generated by IDE. To fix such kind of leak, developers just need to mark these components as private ones in the manifest file. In Android, intra-application component interaction and the inter-application component interaction share the same communication channel [16]. Thus, a single component may be designed for two purposes: *internal use* and *public use*. The second case of permission leak is that developers do not perform enough security checks when an internal component is for *public use*. However, this case is quite difficult to handle, due to two levels of security requirements in a single component.

**Our Solution.** By tracking system-wide application context, FineDroid could be used to fix permission leak vulnerability. With *inter-application context*, we could find whether a component interaction is for *internal use* or for *public use*. With *intra-application context*, we could accurately specify the vulnerable flow inside the application. Combining *intra-application context* and *inter-application context* together, we could make a policy to prevent a vulnerable flow from using permissions when it is invoked from an external application. For example, the policy in Figure 4 denies the `SEND_SMS` permission request from the app *com.android.mms* when a foreign application participates in the interaction and the internal execution state of *com.android.mms* matches a vulnerable path (specified by the *<pcc-selector>* element).

The advantages of FineDroid in preventing permission leak vulnerabilities are that it requires no modification to the system nor the vulnerable applications and the policies are quite easy to write. In Section 7.1, we would evaluate the effectiveness of FineDroid in fixing real-world permission leak vulnerabilities, and show that how the policies could be automatically generated by enhancing a permission leak vulnerability detector.

```
...
<fine-permission android:package="com.flurry.android">
  <deny android:permission="android.permission.ACCESS_FINE_LOCATION" />
  <deny android:permission="android.permission.ACCESS_COARSE_LOCATION" />
</fine-permission>
...
```

*Fig. 5:* Policy to prevent Flurry Ads from requesting location permission.

### 6.2 For Developer: Fine-grained Permission Specification

An Android application may contain many third-party code packages. For example, it is common for applications to embed an Ad library for fetching Ads, social network SDKs for publishing events, payment SDKs for financial charge, analytic SDKs for marketing. However, in this case multiple third-party SDKs from different origins (potentially with different trust levels) will share the same privileges as the host application, violating the principle of least privilege. Thus, a third-party SDK may abuse the permissions that granted to the host application. For example, a popular Ad library was found to collect text messages, contacts and call logs [1]. Unfortunately, developers have no way to restrict the permissions that are available to certain foreign packages.

**Our Solution.** By tracking *intra-application context*, FineDroid is capable of distinguishing the origins of permission requests inside an application. Thus, we could build a permission sandbox inside an application where code packages from different origins have different permission configurations. Based on the permission sandbox, developers could declare fine-grained permission specifications in the application manifest file to specify the permissions that could be used by each third-party SDK. Figure 5 shows the format of this kind of permission specification. The fine-grained permission specifications in the manifest file will be transformed to FineDroid policy by our enhanced *PackageManagerService* at the install-time and added to the *Policy Manager*. Note that application obfuscation [6] would not cause problems here, because developers could modify the manifest file after code obfuscation.

## 7 Prototype & Evaluation

We implement a prototype of FineDroid on Android 4.1.1 (Jelly Bean), running on both Google Nexus phones (Samsung I9250) and emulators. We also implement the two security extensions upon FineDroid. This section evaluates these extensions to demonstrate the effectiveness of our context-sensitive permission enforcement framework, as well as the performance overhead introduced by our framework.

### 7.1 Fixing Permission Leak Vulnerability

We evaluate the effectiveness of FineDroid in fixing permission leak vulnerabilities with two real-world vulnerabilities in Android AOSP apps: `SEND_SMS` leak [7] and `WRITE_SMS` leak [8]. These two vulnerabilities are both caused by the improper protection of public components exposed in the Mms application, which is the default message management app.

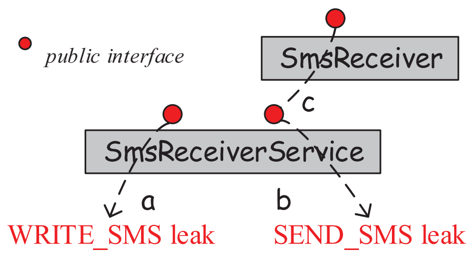

*Fig. 6:* Permission leak paths in Mms application.

**Vulnerability Analysis.** There are two vulnerable components in the Mms application: *SmsReceiverService* which is a Service component and *SmsReceiver* which is a Broadcast Receiver component. Figure 6 illustrates the exploitable paths in this application. *SmsReceiverService* is intended for only *internal use* in the Mms application, while it is mistakenly exported to the public. Through sending a well-crafted Intent to *SmsReceiverService*, an attacker can drive the Mms application to fake the receiving of arbitrary SMS messages (`WRITE_SMS` leak, path *a*) or send arbitrary SMS messages (`SEND_SMS` leak, path *b*). *SmsReceiver* is designed for both *internal use* and *public use*. However, the functionality of sending arbitrary SMS messages which should only be used by private components is not protected properly, causing it to be exported to the public (`SEND_SMS` leak, path *c*).

**Fixing the Vulnerability.** Permission leak vulnerability is typically difficult to fix manually, because it requires enforcing multiple security requirements in a single component, such as `SEND_SMS` leak (path *c* in Figure 6) in *SmsReceiver*. Besides, even if carefully fixed, it also requires the re-distribution of the new application file. Based on FineDroid, we could easily prevent permission leaks by simply writing policies to deny the permission request occurred in the exploitable path without modifying the application. Figure 4 shows an example of how to prevent `SEND_SMS` leak (path *c* in Figure 6) in *SmsReceiver*. Similarly, we could fix the vulnerability of path *a* and *b*.

**Effectiveness.** We created three sample apps to exploit each vulnerable path mentioned above. The sample apps were first tested in our FineDroid prototype with no policies. The result shows that all the three apps successfully exploited the vulnerabilities in the Mms app. Then we added three policies (as showed in Figure 4) to our prototype to fix the three vulnerable paths. We also ran the same three sample apps to attack Mms again. We found that our security policies successfully prevented the permission re-delegation attacks this time, demonstrating the effectiveness of FineDroid in enforcing fine-grained permission use policies.

**Policy Generation.** The policies to fix permission-leak vulnerabilities rely on the precise understanding of vulnerable paths among component interactions. Thus the ideal scenario is to use together with an existing permission leakage vulnerability detector (such as CHEX [24]). Once a vulnerable path is detected, we can automatically generate a corresponding policy for FineDroid. Thus, the task of diagnosing vulnerable applications and writing policies can be greatly simplified. To demonstrate the feasibility of automatic policy generation to be used together with any vulnerability detector, we choose CHEX [24], a state-of-the-art tool in detecting permission leak vulnerability, in our evaluation. However, the source code of CHEX is not available, so we could not directly enhance CHEX for policy generation. Instead, the authors of CHEX provided us the output of CHEX in analyzing 20 vulnerable applications, among which 10 applications are vulnerable to `INTERNET` permission leak. By parsing the output files, we successfully extracted 414 vulnerable paths with detailed calling contexts. Based on the vulnerable paths (contexts), the automatic policy generation is quite straightforward. As showed in Figure 4, the generated policies could deny the permission request when the vulnerable path is exploited by a foreign application. Finally, for each vulnerable path detected by CHEX, a policy is automatically generated to fix it.

## 7.2 Fine-grained Permission Specification

We evaluate the effectiveness of FineDroid in providing fine-grained permission specification by restricting the privileges of untrusted Ad libraries. In this experiment, we use an application named *Stock Watch* which embeds Flurry Ads for fetching and displaying advertisements. For demonstration purpose, we assume Flurry Ads is not trusted by *Stock Watch* developers, thus the developers want to restrict the permissions that could be used by Flurry Ads. Flurry Ads requests `ACCESS_FINE_LOCATION` permission during the execution, and we assume the developers think this is quite suspicious. With FineDroid, *Stock Watch* developers could easily prohibit Flurry Ads from using `ACCESS_FINE_LOCATION` permission. As Figure 5 shows, they just need to declare a fine-grained permission specification in the manifest file. During the installation, these specifications would be transformed to policies that could be added to FineDroid. Because we do not have the source code of the *Stock Watch* application, we mimic the behavior of *Stock Watch* developers by repackaging the application file to replace the manifest file. By running the new application, we could find the `ACCESS_FINE_LOCATION` permission requests from Flurry Ads are all denied by FineDroid, and this does not affect the normal operation of the *Stock Watch* application. Similar to *Stock Watch*, we also tested another 20 applications to restrict the permissions assigned to third-party libraries, including Google Ads, Tapjoy, Millennial Media. In all these cases, FineDroid provides strong enforcement of fine-grained permission specifications. We did encounter two cases that the applications crashed due to the denial of some permissions requested from the Ads library. Instead of considering it as the fault of FineDroid, we argue that developers of the Ads library should write more robust code to handle more necessary exceptions in the future.

## 7.3 Performance Overhead

We have conducted several experiments to measure the performance overhead caused by FineDroid. The experiments are performed on Google Nexus phones.

**Overall Performance.** We first use three performance benchmarks (Caffeine-Mark3, AnTuTu, and Linpack) to measure the overall overhead introduced by Fine-Droid. The results show that almost no noticeable performance overhead is observed, with the worst overhead case at 1.99% in the Linpack benchmark.

**Permission Request Handling Performance.** Most overhead of FineDroid is introduced when handling permission requests. We implement a test app that performs 10,000 times of permission requests to measure the average performance of FineDroid in handling a single permission request. We compare the performance of unmodified Android with FineDroid in two configurations. Context tracking is disabled in *FineDroid w/o Context*, where all overhead is caused by permission interception. In *FineDroid w/ Context*, context tracking is switched on and no policy is installed on the system. Table 1 shows the results.

FineDroid introduces an overhead of 2.02 ms per request in intercepting KEP permission requests, which is undoubtedly higher than the case of unmodified Andorid because in that case KEP request can be handled in the application process without communicating with *Permission Manager* in the system process. The overhead introduced by further application context tracking is very minor (0.02 ms per request). For AEP permissions, the interception overhead is quite minor because AEP is originally enforced in the system process, while the context tracking overhead is more significant because it needs to build intra- and inter-application contexts in several processes.

| Permission Type | Original Android | FineDroid w/o Context | FineDroid w/ Context |
|---|---|---|---|
| *Socket(KEP)* | 0.14ms | 2.16ms $\Delta$2.02ms | 2.18ms $\Delta$0.02ms |
| *IMEI(AEP)* | 0.62ms | 0.69ms $\Delta$0.06ms | 1.09ms $\Delta$0.40ms |

*Table 1:* Results on handling permission requests.

**Policy Matching Performance.** To test the overhead introduced by the policy matching, we add policies to the system to grant the permissions requested by the test app. Each policy is written with the same structure as Figure 4. Table 2 shows the overhead of policy matching.

| Permission Type | FineDroid w/o Policy | FineDroid w/ Policy | Overhead |
|---|---|---|---|
| *Socket(KEP)* | 2.18ms | 3.06 ms | 0.88ms |
| *IMEI(AEP)* | 1.09ms | 1.99ms | 0.90ms |

*Table 2:* Results on policy matching.

We believe the performance penalty introduced by FineDroid is acceptable because permission request (as well as policy matching) do not frequently occur in practice.

## 8 Discussion

To propagate application context, FineDroid relies on `Android Runtime` instance in each application to participate. Since `Android Runtime` is a user-space module in the application process, currently FineDroid cannot guarantee its integrity. Attackers may use Java Reflection to modify `Android Runtime`'s private data structures. To prevent such attacks, we instrument Reflection APIs to prevent manipulation of the private fields which are added by FineDroid to keep application context. Because these fields are unique to FineDroid, this kind of instrumentation would not break other legitimate use of Reflection. Besides, adversaries may also use native code to attack `Android Runtime`. Recent work on isolating native code in Android system [34] could be incorporated to our system to prevent native code attack.

Undesirable data flows among multiple permission requests are not considered in this paper. Actually, by providing fine-grained permission control to raise the bar for abusing permissions, FineDroid could also be used to prevent potential risky data flows.

## 9 Related Work

**Permission System Extensions.** Aurasium [37] provides time-of-use permission granting for legacy Android apps by automatically repackaging applications to attach user-level sandboxing code. Roesner et al. [29] introduced access control gadgets (ACGs) which embed permission-granting semantics in normal user actions. Dr. Android and Mr. Hide [23] provides finer semantics for coarse-grained permissions by rewriting privileged API invocations. Apex [25] introduces partial permission granting at installation time and runtime constraints over permission requests. SEAndroid [33] combines kernel-level MAC (SELinux) with several middleware MAC extensions to the Android permissions model, which could mitigate vulnerabilities in both system and application layer. FlaskDroid [15] extends kernel-level MAC to bring mandatory access control for all resources in Linux Kernel and Android framework. While these works refine or extend current permission system in some degree, they do not enforce fine-grained control over the permission use context, which is the focus of FineDroid.

**Application Interaction Hardening.** Felt et al. [20] proposed IPC inspection to prevent permission re-delegation attacks by intersecting the permissions of all the applications in the IPC call chain. However, this strategy is too rigid to allow intentional permission re-delegations. Quire [18] provides developers with new interfaces to acquire IPC call chain. Different from FineDroid, Quire relies on AIDL instrumentation to record the IPC call chain. However, the technique has several limitations: First, it could only track the IPC call chain during the invocation of AIDL-specified methods, while some system interfaces are not specified using AIDL such as `AcvitityManagerService`; Second, it is an opt-in option for developers to use these enhanced API proxies, thus an attacker application can easily escape.

TrustDroid [14] divides apps into two isolated domains: trusted and untrusted. However, communication problems inside a single domain are not considered. XManDroid [12, 13] generally mitigates application-level privilege escalation attacks by prohibiting any application communication if the permission union of the two apps may pose a security risk. Saint [26] secures the application communication by providing developers with the ability to specify fine-grained requirements about the caller and callee. However, it could not improve the permission enforcement mechanism during the application communication.

AppSealer [38] is a tool to automatically fix component hijacking vulnerabilities by actively instrumenting vulnerable apps. Compared to AppSealer, our technique of fixing permission leak vulnerabilities does not require heavy application rewriting which is error-prone and needs redistribution of patched apps.

Similar to FineDroid, Scippa [10] also extends Binder driver and `Android Runtime` to provide IPC provenance. However, it does not cover *intra-application context* which is quite important for a unified fine-grained permission system. Moreover, the IPC context propagating technique in Scippa is quite simpler than the one designed in FineDroid which could systematically propagate IPC contexts at the level of component-interaction, thread creation/interaction, and events.

**Application Internal Isolation.** To isolate in-app Ads, a separate process is introduced by AFrame [39], AdDroid [27] and AdSplit [31] for running Ads libraries. By intersecting the permissions that can be used by different code packages in the same application, Compac [35] also provides fine-grained permission specification. However, without a systematic context tracking system and a generic policy framework, Compac could not flexibly handle permission requests that cross multiple code packages. Compared with FineDroid, these frameworks could not flexibly regulate permission use policies based on *intra-application* context.

**Context-aware Access Control.** Recent works on context-aware access control model [17,26,30,32] also regulate access control rules based on context information. Different from the notion in FineDroid, these works mostly consider the external application context such as location, time of the day.

## 10 Conclusion

This paper presents FineDroid, which brings context-sensitive permission enforcement to Android. By associating each permission request with its application context, FineDroid provides a fine-grained permission control. The application context in FineDroid covers not only *intra-application context*, but also *inter-application context*. To automatically track such application context, FineDroid designs a new seamless context tracking technique. FineDroid also features a policy framework to flexibly regulate context-sensitive permission rules. This paper further demonstrates the effectiveness of FineDroid by creating two security extensions upon FineDroid for administrators and application developers. The performance evaluation shows that the overhead introduced by FineDroid is minor.

**Acknowledgment.** We thank the anonymous reviewers for their insightful comments. Yuan Zhang and Min Yang are funded in part by the National Program on Key Basic Research (NO. 2015CB358800), the National Natural Science Foundation of China (61300027), and the Science and Technology Commission of Shanghai Municipality (13511504402, 13JC1400800 and 15511103003). Guofei Gu is supported in part by the National Science Foundation (NSF) under Grant CNS-0954096. Any opinions, findings, and conclusions expressed in this material do not necessarily reflect the views of the funding agencies.

## References

1. Ad vulna: A vulnaggressive (vulnerable & aggressive) adware threatening millions. http://www.fireeye.com/blog/technical/2013/10/ad-vulna-a-vulnaggressive-vulnerable-aggressive-adware-threatening-millions.html.
2. Android asynctask class. http://developer.android.com/reference/android/os/AsyncTask.html.
3. Android handler class. http://developer.android.com/reference/android/os/Handler.html.
4. Android message class. http://developer.android.com/reference/android/os/Message.html.
5. Android remains the leader in the smartphone operating system market. http://www.idc.com/getdoc.jsp?containerId=prUS24108913.
6. Proguard. http://developer.android.com/tools/help/proguard.html.
7. Send_sms capability leak in android open source project. http://www.csc.ncsu.edu/faculty/jiang/send_sms_leak.html.
8. Smishing vulnerability in multiple android platforms. http://www.csc.ncsu.edu/faculty/jiang/smishing.html.
9. Sophos security threat report 2013. http://www.sophos.com/en-us/security-news-trends/reports/security-threat-report/android-malware.aspx.
10. BACKES, M., BUGIEL, S., AND GERLING, S. Scippa: System-centric ipc provenance on android.
11. BOND, M. D., AND MCKINLEY, K. S. Probabilistic calling context. In *Proc. of OOPSLA '07*.
12. BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., AND SADEGHI, A.-R. Xmandroid: A new android evolution to mitigate privilege escalation attacks. In *Technical report TR-2011-04, Technische Universit?t Darmstadt* (2011).
13. BUGIEL, S., DAVI, L., DMITRIENKO, A., FISCHER, T., SADEGHI, A.-R., AND SHASTRY, B. Towards taming privilege-escalation attacks on Android. In *Proc. of NDSS '12*.
14. BUGIEL, S., DAVI, L., DMITRIENKO, A., HEUSER, S., SADEGHI, A.-R., AND SHASTRY, B. Practical and lightweight domain isolation on android. In *Proc. of SPSM '11*.
15. BUGIEL, S., HEUSER, S., AND SADEGHI, A.-R. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Proc. of USENIX Security '13*.
16. CHIN, E., FELT, A. P., GREENWOOD, K., AND WAGNER, D. Analyzing inter-application communication in android. In *Proc. of MobiSys '11*.
17. CONTI, M., NGUYEN, V. T. N., AND CRISPO, B. Crepe: Context-related policy enforcement for android. In *Proc. of ISC '10*.
18. DIETZ, M., SHEKHAR, S., PISETSKY, Y., SHU, A., AND WALLACH, D. S. Quire: lightweight provenance for smart phone operating systems. In *Proc. of Security '11*.
19. ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. N. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of OSDI'10*.
20. FELT, A. P., WANG, H. J., MOSHCHUK, A., HANNA, S., AND CHIN, E. Permission re-delegation: attacks and defenses. In *Proc. of USENIX Security '11*.
21. GRACE, M., ZHOU, Y., WANG, Z., AND JIANG, X. Systematic detection of capability leaks in stock android smartphones. In *Proc. of NDSS '12*.

22. HORNYACK, P., HAN, S., JUNG, J., SCHECHTER, S., AND WETHERALL, D. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proc. of CCS '11*.
23. JEON, J., MICINSKI, K. K., VAUGHAN, J. A., FOGEL, A., REDDY, N., FOSTER, J. S., AND MILLSTEIN, T. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proc. of SPSM '12*.
24. LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of CCS '12*.
25. NAUMAN, M., KHAN, S., AND ZHANG, X. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proc. of AsiaCCS '10*.
26. ONGTANG, M., MCLAUGHLIN, S., ENCK, W., AND MCDANIEL, P. Semantically rich application-centric security in android. In *Proc. of ACSAC '09*.
27. PEARCE, P., FELT, A. P., NUNEZ, G., AND WAGNER, D. Addroid: Privilege separation for applications and advertisers in android. In *Proc. of AsiaCCS '12*.
28. POEPLAU, S., FRATANTONIO, Y., BIANCHI, A., KRUEGEL, C., AND VIGNA, G. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proc. of NDSS'14*.
29. ROESNER, F., KOHNO, T., MOSHCHUK, A., PARNO, B., WANG, H., AND COWAN, C. User-driven access control: Rethinking permission granting in modern operating systems. In *Proc. of SP '12*.
30. ROHRER, F., ZHANG, Y., CHITKUSHEV, L., AND ZLATEVA, T. Dr baca: Dynamic role based access control for android. In *Prof. of ACSAC '13*.
31. SHEKHAR, S., DIETZ, M., AND WALLACH, D. S. Adsplit: Separating smartphone advertising from applications. In *Proc. of USENIX Security'12*.
32. SINGH, K. Practical context-aware permission control for hybrid mobile applications. In *Proc. of RAID '13*.
33. SMALLEY, S., AND CRAIG, R. Security enhanced (se) android: Bringing flexible mac to android. In *Proc. of NDSS '13*.
34. SUN, M., AND TAN, G. Nativeguard: Protecting android applications from third-party native libraries. In *Proc. of WiSec '14*.
35. WANG, Y., HARIHARAN, S., ZHAO, C., LIU, J., AND DU, W. Compac: Enforce component-level access control in android. In *Proc. of CODASPY '14*.
36. WU, L., GRACE, M., ZHOU, Y., WU, C., AND JIANG, X. The impact of vendor customizations on android security. In *Proc. of CCS '13*.
37. XU, R., SAIDI, H., AND ANDERSON, R. Aurasium: Practical policy enforcement for android applications. In *Proc. of USENIX Security'12*.
38. ZHANG, M., AND YIN, H. AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications. In *Proc. of NDSS '14*.
39. ZHANG, X., AHLAWAT, A., , AND DU, W. Aframe: Isolating advertisements from mobile applications in android. In *Proc. of ACSAC '13*.
40. ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proc. of CCS '13*.
41. ZHOU, Y., AND JIANG, X. Detecting passive content leaks and pollution in android applications. In *Proc. of NDSS '13*.