This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

Rethinking Permission Enforcement Mechanism on Mobile Systems

Yuan Zhang, Min Yang, Guofei Gu, and Hao Chen

***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 three potential users: end-users, administrators and developers. Furthermore, FineDroid is showed to introduce a minor overhead.

***Index Terms*—permission enforcement; application context; policy framework**

I. 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. Since Android has been expanding its market share rapidly as the most popular mobile platform [1], this paper mainly focuses on the permission model of Android. Ideally, the Android permission model should prevent malicious applications from abusing sensitive resources. Actually, due to some features of the Android ecosystem, malicious entities could easily abuse permissions, leading to the explosion of Android malware [2] and the numerous reported application vulnerabilities [3], [4] in the past few years.

To better understand this problem, we first study the characteristics of current Android application ecosystem.

- SDK incorporation is quite popular in mobile app development. With these SDKs, third-party apps can invoke web services to provide rich functionalities in an easier way, such as posting messages to Twitter, navigating using Google maps.
- Except Google Play, a lot of third-party application markets exist in the Android application ecosystem. Due to the lack of centralized management, an app could be repackaged, e.g., with an embedded advertisement library for profit, and redistributed through third-party markets.
- Application interaction is a built-in feature in Android programming model. It is quite effective to ease application development. For example, a lot of social apps encourage clients to upload portraits. Without application interaction, the app needs to support photo shooting and editing which is quite professional and complex. Fortunately, with application interaction, the app could delegate these functionalities to professional apps.

From the above analysis, we can find that there are many principals (such as incorporated SDKs, repacked payloads, deputy apps) that collaboratively participate in the functionalities of an app, while current application-level permission enforcement cannot precisely control the fine-grained permissions that each principal can use. Specifically, we summarize the limitations of current permission model from the following two aspects.

- **Intra-application Context Insensitive.** In existing permission model, each application is treated as a separate principal, while the above analysis indicates that the the application code may be contributed by multiple parties (app developers, SDK providers, repackaged payload, etc). Current permission enforcement mechanism is limited in simply applying a single security policy for the entire app.
- **Inter-application Context Insensitive.** Although application interaction is a common characteristic of mobile applications, it is transparent to the current coarse-grained permission enforcement mechanism, exposing a new attack surface, i.e., the permissions granted to a vulnerable

Copyright (c) 2013 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org

This material is based upon work supported 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). Yuan Zhang is sponsored by Shanghai Sailing Program (16YF1400800) and Shanghai Excellent Academic Leader Funds (16XD1400200).

Yuan Zhang, Min Yang (Corresponding Author) are with the School of Computer Science and the Shanghai Key Laboratory of Data Science, Fudan University, Shanghai, China, 201203. E-mail: {yuanxzhang, m_yang}@fudan.edu.cn; Guofei Gu is with the Department of Computer Science and Engineering, Texas A&M University, College Station, TX, 77840. E-mail: guofei@cse.tamu.edu; Hao Chen is with the Department of Computer Science, University of California, Davis, CA, 95616-5270. E-mail: chen@ucdavis.edu.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

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 [5] provides fine-grained semantics for serval permissions by adding a mediation layer. SE-Android [6] hardens the permission enforcement system by introducing SELinux extensions to the Android middleware. FlaskDroid [7] 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 [8]–[11] 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 [11]–[15] or to isolate untrusted components inside an application [16]–[19]. 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 (in this paper, when we say context we mean the application execution 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 end-users, administrators and developers. First, we provide end-users with in-context permission granting in which users could make permission granting decisions just for the occurrent application context. Second, since permission leak vulnerability [3], [4], [12], [20] is very common and dangerous, we show how administrators could benefit from our system in transparently fixing these vulnerabilities without modifying vulnerable applications. Last, 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 end-users, we show that "for this context" permission granting is the most popular choice by users, and without such in-context granting option, users would have to handle 13 times more permission prompts if they still want to make rational permission request decisions (details presented in Section VIII-A). 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 three 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.

II. Android Background

The permission model of Android has been well described in [21]. Here we introduce some most relevant background.

Application Process Model. By default, each Android application runs in an isolated process with unique UID/GID. To boost the application creation performance, each application process is spawned from a process incubator, named Zygote. When an application process is spawned, Zygote initializes the process as an `Android Runtime` instance which keeps a communication channel with the Android system. Through this channel, the Android system could manage the execution of every application.

Binder Interaction. Android designs the Binder IPC mechanism to facilitate interactions among applications and system services. During startup, `Android Runtime` spawns a specific Binder thread to handle all Binder transactions routed to this application. If one Binder thread is not enough to handle these transactions timely, `Android Runtime` would automatically spawn new Binder threads. The Binder driver plays as a router that forwards transactions from clients to servers. During each transaction, the Binder driver also annotates some meta-data such as the UID, PID of the client. Based on the transaction meta-data, Android system services could get the client identity and enforce permission checks.

Component Model. In Android programming model, apps are structured in components: `Activity`, `Service`, `Broadcast Receiver` and `Content Provider` [21]. `Intent` object is needed for component interactions to

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

describe the selection criteria for the target component and invocation arguments. For an interaction, an application needs to send an `Intent` to Android system, and the Android system would check the meta-data of the `Intent` object to determine the target component. After the target component is selected, the Android system would forward the `Intent` to the `Android Runtime` of the target application through the communication channel set up during the application startup. According to the `Intent` object, `Android Runtime` would instantiate the target component and invoke its corresponding interface for handling.

III. 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 [22]) to inject malicious payloads. In addition, an attacker could also publish malicious SDKs, passively waiting for developers to include [23].

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 [3], [4], [12], component hijacking [20], content leak and pollution [24]. In these attacks, the 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 [7], [15], [25], [26].

IV. 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 V, and we describe the context-sensitive permission enforcement system in Section VI.

V. 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.

A. 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.

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

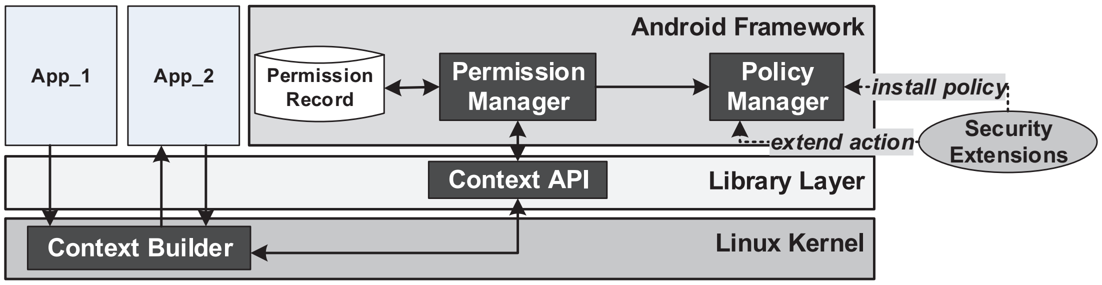


Fig. 1: Architecture of Context-Sensitive Permission Enforcement Framework.

PCC as Intra Context. We adopt a technique called *probabilistic calling context (PCC)* [27] 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 [27], 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 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.

B. 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.

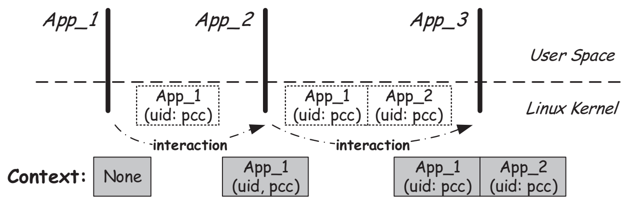


Fig. 2: Binder IPC Context Building.

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.

C. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

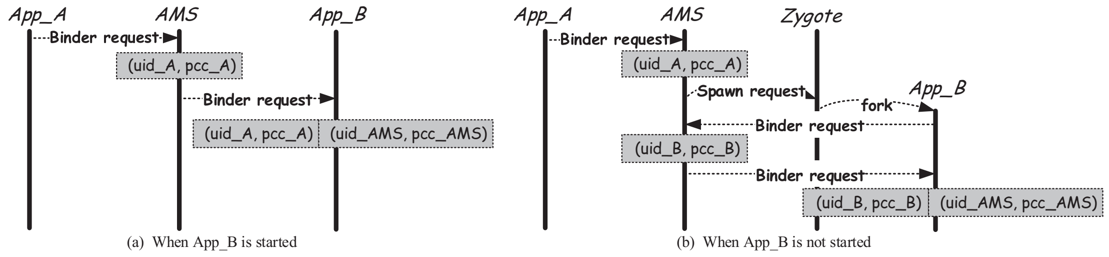


Fig. 3: Binder IPC Propagating Diagram in Component Interaction from App_A to App_B.

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 [28], Handler [29], AsyncTask [30] 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.

Compared with Scippa. Scippa [31] is a system to provide IPC call-chains across application process. Similar to our context propagating technique, Scippa also extends Binder driver and Android Runtime to propagate IPC context. However, it is limited in application to context-sensitive permission enforcement from the following aspects:

- First, Scippa does not build and propagate *intra-application context* during IPC interaction while it is quite important for preventing intra-applications attacks and inter-application attacks;
- Second, our system considers a more complicated component interaction model which covers two scenarios in sending Intents (see Fig. 3) and effectively hiding the presence of AMS in the IPC context;
- Third, our system proposes *thread-level propagating* and *event-level propagating* to systematically propagate IPC context inside an application, while Scippa only supports context propagating during thread interaction .

VI. 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 context-sensitive policy framework. Besides, FineDroid also collects detailed context information to facilitate expressive context-based policy writing.

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

policy := <*action*> <*app*> <*permission*> <*context*>
action := *grant* | *deny* | ...

Fig. 4: Policy Language for FineDroid.

A. Permission Manager

Permission Manager first needs to intercept all permission requests. As introduced in [19], [32], two kinds of permission requests are intercepted: KEPs (Kernel Enforced Permissions) and AEPs (Android Enforced Permissions). The two kinds of permission requests are intercepted differently.

KEP Interception. Since KEP is enforced by the UID/GID isolation mechanism in Linux Kernel [32], we instrument the related modules in the Linux Kernel to intercept all KEP permission requests. The permission request is then redirected to *Permission Manager* in the Android framework for handling.

AEP Interception. *PermissionController* service is a unified point for permission checking in the Android framework. We instrument *PermissionController* service to redirect all permission requests to the *Permission Manager*. Since the AEP permission requests are preformed through Binder, *Permission Manager* could easily obtain the application contexts for the AEP permission requests with *Context API*.

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.

B. Policy Framework

FineDroid designs a declarative policy language to express the rules for handling permission requests in a context-sensitive manner. Figure 4 shows the structure of our policy language. Basically, a policy specifies the action <*action*> to perform when an app <*app*> requests a permission <*permission*> under the application context of <*context*>. To ease the expression of application context, our policy is structured in XML format, with the following tags. (Sample policies can be found in Figure 5 and Figure 7.)

- **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.

C. Expressive Context-based Policy Writing

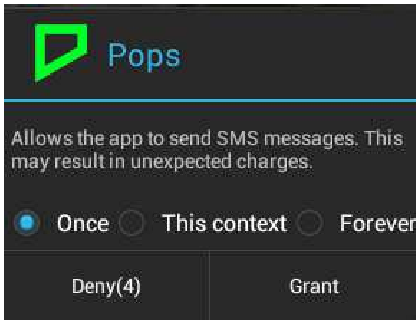
To ease the writing of context-based policies, FineDroid collects the detailed context information during context building. The recorded context information enables writers to specify context policies in a more expressive manner.

Intra-application Context Information. To support describing **uid-context** with method signatures, FineDroid collects the method signatures for all the Java methods in the current intra-application context. Note that the detailed context information is collected only when a new intra-application context (with a PCC value never see before in this app) is encountered. Since the detailed context information is kept in the file system, FineDroid using encryption to prevent attackers from modifying the context information.

Application Identity Information. In Android, a unique UID/GID pair is selected during application installation to

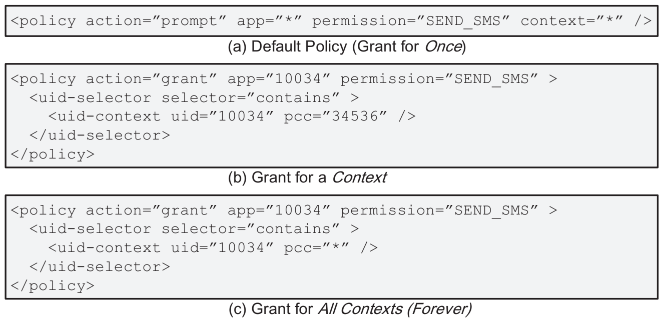
1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security



Allows the app to send SMS messages. This may result in unexpected charges.

Fig. 5: Example of In-Context Permission Granting Interface.



```
<policy action="prompt" app="*" permission="SEND_SMS" context="*" />
```

(a) Default Policy (Grant for *Once*)

```
<policy action="grant" app="10034" permission="SEND_SMS" >
  <uid-selector selector="contains" >
    <uid-context uid="10034" pcc="34536" />
  </uid-selector>
</policy>
```

(b) Grant for a *Context*

```
<policy action="grant" app="10034" permission="SEND_SMS" >
  <uid-selector selector="contains" >
    <uid-context uid="10034" pcc="*" />
  </uid-selector>
</policy>
```

(c) Grant for *All Contexts (Forever)*

Fig. 6: Sample Policies for In-Context Permission Granting.

represent as the identity of an app. However, UID/GID is not friendly for users to understand. Instead, FineDroid supports using the package name of an app to describe the app. To map a package name to UID/GID pair, FineDroid keeps all the package names of the application packages managed by *PackageManagerService*.

VII. Security Extensions

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

A. For End-User: In-Context Permission Granting

In the current Android permission model, an app needs to declare all requested permissions in a manifest file and a user should decide whether to grant *all* these permissions at the *installation time* or to abort the installation. Recent user studies have indicated that this grant-all-or-not-install permission model could not help users make correct security decisions [33], [34]. To address this problem, Aurasium [35] repackages apps and provides time-of-use permission granting for Android but at the application level only.

A limitation of this solution is that a user may wish to grant the `SEND_SMS` permission only when she clicks the "Send" button but not in any other contexts of the application. Unfortunately, we are unaware of any existing technique capable of granting permissions based on contexts. FineDroid improves application-level permissions by providing *in-context permission granting*, which allows the user to allow or deny a permission for the current context. Figure 5 shows an in-context permission granting user interface, where the user can choose to allow/deny a permission (1) always in the current context, (2) always in all contexts, or (3) just once. Note that in-context permission granting still relies on users to make rational allow/deny decisions, but we argue that it could benefit users for the new flexibility in making decisions (see Section VIII-A).

We select 12 high-risk permissions, such as `INTERNET`, `SEND_SMS`, and `READ_CONTACTS`, to provide in-context granting when these permissions are requested. To support user granting, we add a new action type `prompt` to the policy language. When *Permission Manager* performs a "prompt" action, it would invoke our registered callback to handle this action. Our callback would prompt a dialog such as Figure 5 for users to make a granting decision.

To specify that the 12 high-risk permissions should be handled with "prompt" action, we register 12 default policies to the system through the interface of *Policy Manager*. Figure 6 (a) shows a sample policy. When a user chooses to grant a permission for the current context, we add a policy to the system to specify that this permission is granted to the app with current PCC value. Figure 6 (b) shows an example of such in-context permission granting policy. Note that the PCC integer value is used only by FineDroid to identify a context and is transparent to the end user, who makes context-sensitive decisions based only on the context in the application's user interface. When the application requests the same permission in another scenario/context (which has a new PCC value), our system could still prompt users for permission granting. If users are quite confident about that an application should use a permission forever, our system would add a policy as showed in Figure 6 (c).

B. 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 [3], [4], [20], 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 [36]. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

```
<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. 7: Policy to fix SEND_SMS permission leak in *SmsReceiver*.

is quite difficult to handle, due to two levels of security requirements in a single component.

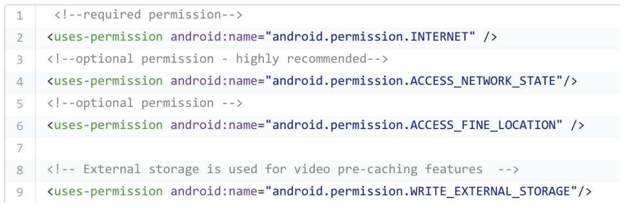
Our Solution. By tracking system-wide application context, FineDroid could be used to fix permission leak vulnerability. With *inter-application context*, we could find whether a component interaction is for *internal use* or for *public use*. With *intra-application context*, we could accurately specify 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 7 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 VIII-B, 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.

C. *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 [23]. 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 9 shows the format of this



```
 <!--required permission-->
<uses-permission android:name="android.permission.INTERNET" />
<!--optional permission - highly recommended-->
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<!--optional permission -->
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

<!-- External storage is used for video pre-caching features  -->
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

Fig. 8: Original policy to incorporate Flurry Ads in Android Apps.

```
...
<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. 9: Policy to prevent Flurry Ads from requesting location permission.

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 [37] would not cause problems here, because developers could modify the manifest file after code obfuscation.

Burden on Developers. To isolate third-party code packages, developers need to specify fine-grained permission configurations in the manifest file, as showed in Figure 9. Actually, it would not place significant burden on developers, since they already need to configure permissions when incorporating third-part code packages. For example, consider a developer who wishes to incorporate the Flurry SDK in her app. Flurry's documentation requires her to declare four permissions in the application manifest, shown in Figure 8 [38]. If she uses FindDroid, then instead of declaring those permissions, she would specify the policy in Figure 9. Finally, the developer need to set fine-grained permissions only when she wishes to restrict the permissions of untrusted SDKs.

VIII. 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 three 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.

A. *In-Context Permission Granting*

In-context permission granting provides flexible control over permission usage. What option to choose in a permission granting decision is a tradeoff between security and usability. The "forever" option represents one end of the spectrum with the maximum usability but possibly insufficient security (because the correct permission decisions might be different in different contexts), while the "for once" option represents the other end of the spectrum with the maximum security but the worst usability (because the user has to answer every

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

permission question). The "for this context" option that we propose represents a middle ground.

However, how useful is it? How often is it more appropriate than "for once" and "forever"? We conducted a user study to find it out. In this study, we hired 10 volunteers in our university to participate. We selected 70 top Google Play apps (e.g., Cut the Rope, Firefox, BBC News, IKEA Catalog) from 21 different categories, and each participant chose a subset of these apps and ran them on FineDroid. The task of the participating users was to thoroughly use the provided apps and to trigger as many features of the apps as possible.

Recommended Best-practice Setting. To help users specify policies using in-context permission granting without bringing in more risks, we recommend them the best-practice setting. We first recommended them about the three options that should choose when making granting decisions: (1) if you do not want the system to remember the decision, then choose "for once", (2) if you do not want to handle the request of this permission in the *same current scenario/context* any more, then choose "for this context", and (3) if you do not want to handle the request of this permission anytime in this app, then choose "forever". We then recommend them to deny a permission request "for once" when they are not sure whether to grant or not. If the denial of the permission request does not come with decrease of service quality or application stability, we recommend them to deny the same permission "for this context". We advice them to cautiously choose "forever" option. Since in-context permission granting provides new flexibility in controlling application behaviors, our recommended policy setting is expected to help users gain a better balance between security and usability.

Before the experiment, we gave them 5-minute training about in-context permission granting. We introduced them about the three options that could choose when making granting decisions: (1) if you do not want the system to remember the decision, then choose "for once", (2) if you do not want to handle the request of this permission in the *same current scenario/context* any more, then choose "for this context", and (3) if you do not want to handle the request of this permission anytime in this app, then choose "forever".

Overall Usability Impact. In all, we collected 158 permission granting traces for 70 apps. On average, each trace lasts about 12.3 minutes. Totally, users make 968 permission granting decisions. On average, users make about 6.13 (968/158) permission decisions per app, which means one decision for about 2 minutes. Thus, we can find that our new in-context permission granting mechanism does not place significant burden for end-users.

Effectiveness. To study the benefits drawn by in-context permission granting, we assume that each permission decision (allow or deny) that a user make is *rational*[1], i.e., no user would intentionally allow a permission when it should be denied, and vice versa. Specifically, we look into the following questions.

[1]Note that rational does not necessarily imply optimal, e.g., a user may choose "for once" while actually "for this context" would be more appropriate.

Q1. How often is "for this context" chosen by users? By examining the options chosen by users when making granting decisions, we found the users made 590 "for this context" decisions, which is the mostly chosen option. Based on users' context-sensitive decisions, our system automatically handles 37,666 permission requests without further bothering the users. It means 37666/(37666+590)=98.5% unnecessary permission prompts are effectively eliminated within in-context permission granting.

The above results, while encouraging, should be taken with a grain of salt, because users may not always choose the most appropriate option when making decisions. It means that users may choose "for this context" option when "forever" would be more appropriate. Since the "forever" option is not new introduced by us, we should not attribute all the eliminated 37,666 permission prompts to in-context permission granting. To investigate the benefits owing to in-context permission granting, we first need to judge when "for this context" option is the most appropriate. However, manually judging would be laborious and subject to our biases. Finally, we find a way to deduce the cases when "for this context" option is the most appropriate from users' actual decisions (see Q2 and Q3). Thus, we could measure the benefits that owe to in-context permission granting from these cases (see Q4).

Q2. How often is "for once" more appropriate than "for this context"? In our study, users have made a total of 72 choices of "for once" option in 50 contexts. However, no user has ever chosen both "allow for once" and "deny for once" in the same permission request context of an app. This implies that if these users had chosen "for this context" instead of "for once", they would have the same security effect but would have avoided many permission prompts. This study shows that "for once" option rarely, if ever, could be a more appropriate choice than "for this context".

Q3. How often is "for this context" the most appropriate choice? When a user both allowed and denied the same permission (with "for this context" option) but in different application contexts in an app, it is reasonable to infer that the "forever" option is inappropriate for this permission in this app. Instead, "for this context" option is probably the most appropriate choice in this case. For each user in each app, our user study covered a total of 296 permissions and 740 distinct permission request contexts. Among these permissions, 61 permissions have been both granted and denied by the same user in different contexts of an app, for a total of 217 distinct permission request contexts. In other words, in these 217 (or 217/740=29.3%) permission request contexts, "for this context" is the most appropriate choice.

Q4. Without "for this context" option, how many more prompts would users need to handle? In a time-of-use permission granting system where "for this context" option is unavailable, when a user is asked a permission question in one of the above 217 permission contexts, the user could choose either "for once" or "forever". Note that the "forever" option would be inappropriate in the 217 contexts discussed above. As a result, rational users have to choose "for once" option, thus would have to handle more prompts. In the user study, for the 217 contexts above, the system

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

automatically handles 12,678 permission requests based on the users' context-sensitive decisions. It implies that users would have to handle 12678/968≈13 times more permission prompts if the "for this context" option is unavailable.

Longer-term Usability Study. Since FineDroid remembers the user's context-specific permission decisions, we expect that the user needs to repeat fewer permission decisions. To measure this improved usability in a longer term, we chose five volunteers from the previous study to participate in a five-day study. We asked each participant to choose her five most frequently used apps from the previous study so that they would be willing to spend more time on these apps. During the study, just as in the previous study, our system recorded all the permission granting decisions made by the user and by FineDroid, respectively. In total we collected 25 permission granting traces in 17 apps. The traces show that the participants made a total of only 27 permission granting decisions, all of which occurred in the first hour of using the app. By contrast, FineDroid automatically made 93,817 permission granting decisions. This result corroborates what we observed during our previous shorter-term user study: that in-content permission granting significantly reduces the number of permission request prompts.

B. 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 [39] and `WRITE_SMS` leak [40]. 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.

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 10 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-designed 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 10) 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 7 shows an example of how to prevent `SEND_SMS` leak (path *c* in Figure 10) in *SmsReceiver*. Similarly, we could fix the vulnerability of path *a* and *b*.

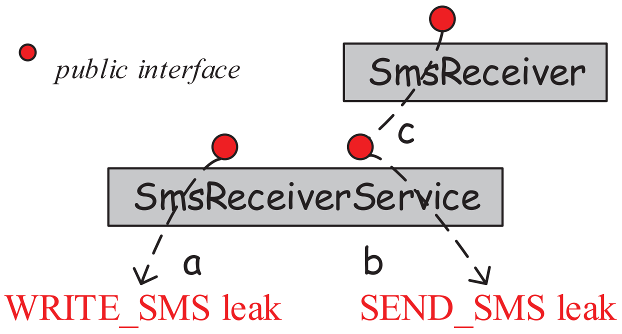


Fig. 10: Permission leak paths in Mms application.

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 7) 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 [20]). 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 [20], 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 7, 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.

C. 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

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

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 9 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.

D. 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 (CaffeineMark3, AnTuTu, and Linpack) to measure the overall overhead introduced by FineDroid. 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 I 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 Δ2.02ms	2.18ms Δ0.02ms
IMEI(AEP)	0.62ms	0.69ms Δ0.06ms	1.09ms Δ0.40ms

TABLE I: 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 7. Table II 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	1.99 ms	0.90ms

TABLE II: Results on policy matching.

We also measure the performance of the aforementioned optimization which caches previous policy matching results to reduce the overall matching overhead. From Table III, we can find this optimization significantly reduces the total cost of policy matching.

Permission Type	**FineDroid w/o opt.**	**FineDroid w/ opt.**	**Reduced Overhead**
Socket(KEP)	4.94 ms	3.06 ms	1.88ms
IMEI(AEP)	3.88 ms	1.99ms	1.89ms

TABLE III: Results on cached policy matching optimization.

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

E. PCC Conflict Probability

Since PCC is probabilistic [27], two distinct intra-application contexts may have the same PCC value, which would degrade the effectiveness of our context tracking technique. Thus, this section measures the probability of PCC conflict. In the user study performed in Section VIII-A, our system recorded all the application execution contexts with detailed calling context information. By analyzing the logs collected in the user study, we found there are 4.68 distinct permission request contexts on average in the collected 158 permission granting traces, and there are no conflict PCC values found in these contexts. Since our system only calculates PCC values in permission request contexts, the total number of PCC in an application should be relatively small. As evaluated in [27], a 32-bit PCC only has few conflicts for millions of unique contexts, thus we believe the adoption of PCC would hardly introduce context conflicts in our system.

IX. Discussion

A. About FineDroid.

To propagate application context, FineDroid relies on Android Runtime instance in each application to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

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 [41], [42] 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.

B. About In-context Permission Granting.

It is a long-running research problem about how to communicate end-users with the application behavior to help them make security decisions. However, making decisions is hard, helping users to make decisions is even harder. Permission granting mechanism itself could not tell users the correct decision, but the way it interacts with end-users directly impacts the security and usability. In installation-time permission granting, users are confused to make security decisions without knowing the permission usage. Thus, researchers proposed time-of-use permission granting for Android, such Apex [43], Dr. Android and Mr. Hide [5]. In time-of-use permission granting, users could delay the granting decisions to runtime, at the exact time when application requests the permission. Compared with installation-time granting, time-of-use granting could provide better understanding of the risks of granting decisions. However, time-of-use granting would increase the number of granting decisions and interrupt the normal usage of the application. In-context permission granting proposed by this paper seems a better choice than time-of-use granting, because it not only allow users make decisions at runtime, but also avoid asking users the same permission granting questions. The comparison among installation-time granting, time-of-use granting and in-context granting is depicted in Table IV. By providing users with a new way to interact with the permission granting mechanism, in-context permission granting significantly advances this line of research, achieving a better balance between security and usability.

Granting mechanism	Security	Usability
Installation-time granting	×	√
Time-of-use granting	√	×
In-context granting	√	√

TABLE IV: Comparison of three permission granting mechanisms.

Although in-context permission granting is appealing, it still needs users to judge whether an app should gain a permission or not. We do not claim that it could help users to make probably more correct allow/deny decisions, but argue the flexibility in making granting decisions could benefit users. Actually, we expect this kind of permission granting decisions can be provided by a trusted party, such as Google Play or company IT administrators. The power of in-context permission granting lies in its flexibility in regulating permission requests in a context-sensitive manner.

X. Related Work

Permission System Extensions. Dr. Android and Mr. Hide [5] provides finer semantics for coarse-grained permissions by rewriting privileged API invocations. SEAndroid [6] 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 [7] 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.

Permission Granting Extensions. Fratantonio et al. [44] explored the practicality of the adoption of finer-grained system for the Internet permission by designing an automated system to extract all the domains names that an app need to access, and rewritten the apps to restrict the Internet access ability. Apparently, this solution is limited and not generally applicable to other permissions. Aurasium [35] provides time-of-use permission granting for legacy Android apps by automatically repackaging applications to attach user-level sandboxing code. With well-designed DEX sandbox and native code sandbox, AppCage [42] also supports time-of-use permission granting, but it does not require application rewriting and can effectively defeat native code attack. Roesner et al. [45] introduced access control gadgets (ACGs) which embed permission-granting semantics in normal user actions. Apex [43] introduces partial permission granting at installation time and runtime constraints over permission requests. Although these extensions improve the original granting mechanism in some degree, they do not give users the ability to associate permission granting decisions to the corresponding contexts, while it is an appealing feature provided by the in-context permission granting.

Application Interaction Hardening. Felt et al. [12] 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 [13] 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 isolated trusted and untrusted domains, without considering the communication

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

problems inside a single domain. XManDroid [15] 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 [11] 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 [46] 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.

Application Internal Isolation. To isolate in-app Ads, a separate process is introduced by AFrame [18], AdDroid [17] and AdSplit [16] for running Ads libraries. By intersecting the permissions that can be used by different code packages in the same application, Compac [19] 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 [8]–[11] 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.

XI. 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 three security extensions upon FineDroid for end-users, administrators and application developers. The performance evaluation shows that the overhead introduced by FineDroid is minor.

References

[1] “Android remains the leader in the smartphone operating system market,” http://www.idc.com/getdoc.jsp?containerId=prUS24108913.

[2] “Sophos security threat report 2013,” http://www.sophos.com/en-us/security-news-trends/reports/security-threat-report/android-malware.aspx.

[3] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, “Systematic detection of capability leaks in stock android smartphones,” in *Proc. of NDSS ’12*.

[4] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, “The impact of vendor customizations on android security,” in *Proc. of CCS ’13*.

[5] J. Jeon, K. K. Micinski, J. A. Vaughan, A. Fogel, N. Reddy, J. S. Foster, and T. Millstein, “Dr. android and mr. hide: fine-grained permissions in android applications,” in *Proc. of SPSM ’12*.

[6] S. Smalley and R. Craig, “Security enhanced (se) android: Bringing flexible mac to android,” in *Proc. of NDSS ’13*.

[7] S. Bugiel, S. Heuser, and A.-R. Sadeghi, “Flexible and fine-grained mandatory access control on android for diverse security and privacy policies,” in *Proc. of USENIX Security ’13*.

[8] M. Conti, V. T. N. Nguyen, and B. Crispo, “Crepe: Context-related policy enforcement for android,” in *Proc. of ISC ’10*.

[9] K. Singh, “Practical context-aware permission control for hybrid mobile applications,” in *Proc. of RAID ’13*.

[10] F. Rohrer, Y. Zhang, L. Chitkushev, and T. Zlateva, “Dr baca: Dynamic role based access control for android,” in *Prof. of ACSAC ’13*.

[11] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel, “Semantically rich application-centric security in android,” in *Proc. of ACSAC ’09*.

[12] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin, “Permission re-delegation: attacks and defenses,” in *Proc. of USENIX Security ’11*.

[13] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach, “Quire: lightweight provenance for smart phone operating systems,” in *Proc. of Security ’11*.

[14] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, “Practical and lightweight domain isolation on android,” in *Proc. of SPSM ’11*.

[15] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry, “Towards taming privilege-escalation attacks on Android,” in *Proc. of NDSS ’12*.

[16] S. Shekhar, M. Dietz, and D. S. Wallach, “Adsplit: Separating smartphone advertising from applications,” in *Proc. of USENIX Security’12*.

[17] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner, “Addroid: Privilege separation for applications and advertisers in android,” in *Proc. of AsiaCCS ’12*.

[18] X. Zhang, A. Ahlawat, , and W. Du, “Aframe: Isolating advertisements from mobile applications in android,” in *Proc. of ACSAC ’13*.

[19] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du, “Compac: Enforce component-level access control in android,” in *Proc. of CODASPY ’14*.

[20] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “Chex: statically vetting android apps for component hijacking vulnerabilities,” in *Proc. of CCS ’12*.

[21] W. Enck, M. Ongtang, and P. McDaniel, “Understanding android security,” *IEEE Security & Privacy*, 2009.

[22] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna, “Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications,” in *Proc. of NDSS’14*.

[23] “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.

[24] Y. Zhou and X. Jiang, “Detecting passive content leaks and pollution in android applications,” in *Proc. of NDSS ’13*.

[25] 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 *Proc. of OSDI’10*.

[26] 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 *Proc. of CCS ’11*.

[27] M. D. Bond and K. S. McKinley, “Probabilistic calling context,” in *Proc. of OOPSLA ’07*.

[28] “Android message class,” http://developer.android.com/reference/android/os/Message.html.

[29] “Android handler class,” http://developer.android.com/reference/android/os/Handler.html.

[30] “Android asynctask class,” http://developer.android.com/reference/android/os/AsyncTask.html.

[31] M. Backes, S. Bugiel, and S. Gerling, “Scippa: System-centric ipc provenance on android.”

[32] Y. Zhang, M. Yang, B. Xu, Z. Yang, G. Gu, P. Ning, X. S. Wang, and B. Zang, “Vetting undesirable behaviors in android apps with permission use analysis,” in *Proc. of CCS ’13*.

[33] P. G. Kelley, S. Consolvo, L. F. Cranor, J. Jung, N. Sadeh, and D. Wetherall, “A conundrum of permissions: Installng applications on an Android smartphone,” in *Proc. of USEC ’12*.

[34] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, “Android permissions: user attention, comprehension, and behavior,” in *Proc. of SOUPS ’12*.

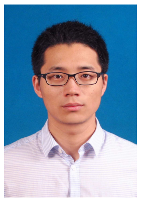
[35] R. Xu, H. Saıdi, and R. Anderson, “Aurasium: Practical policy enforcement for android applications,” in *Proc. of USENIX Security’12*.

[36] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, “Analyzing inter-application communication in android,” in *Proc. of MobiSys ’11*.

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TIFS.2016.2581304, IEEE Transactions on Information Forensics and Security

[37] "Proguard," http://developer.android.com/tools/help/proguard.html.
[38] "Documentation for incorporating flurry android sdk," https://developer.yahoo.com/flurry/docs/publisher/code/android/.
[39] "Send_sms capability leak in android open source project," http://www.csc.ncsu.edu/faculty/jiang/send_sms_leak.html.
[40] "Smishing vulnerability in multiple android platforms," http://www.csc.ncsu.edu/faculty/jiang/smishing.html.
[41] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *Proc. of WiSec '14*.
[42] Y. Zhou, K. Patel, L. Wu, Z. Wang, and X. Jiang, "Hybrid user-level sandboxing of third-party android apps," in *Proc. of AsiaCCS '15*.
[43] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending android permission model and enforcement with user-defined runtime constraints," in *Proc. of AsiaCCS '10*.
[44] Y. Fratantonio, A. Bianchi, W. Robertson, M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "On the Security and Engineering Implications of Finer-Grained Access Controls for Android Developers and Users," in *Proc. of DIMVA '15*.
[45] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. Wang, and C. Cowan, "User-driven access control: Rethinking permission granting in modern operating systems," in *Proc. of SP '12*.
[46] M. Zhang and H. Yin, "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications," in *Proc. of NDSS '14*.



Yuan Zhang in an Assistant Professor in Software School, Fudan University. He received his Ph.D. degree from Fudan University in 2014 and B.Eng. degree from Nanjing University in 2009. His research interests include system security and compiler techniques.

Min Yang is a Professor in Software School, Fudan University. He received the B.Sc. and Ph.D degrees in computer science from Fudan University in 2001 and 2006, respectively. His research interests are in system software and security.

Guofei Gu is an Associate Professor in the Department of Computer Science & Engineering at Texas A&M University (TAMU). He received his Ph.D. degree in Computer Science from the College of Computing, Georgia Institute of Technology. His research interests are in network and system security, social web security, cloud and software-defined networking (SDN/OpenFlow) security.

Hao Chen is an Associate Professor in the Department of Computer Science at the University of California, Davis. He received the BS and MS degrees from Southeast University and the PhD degree from the Computer Science Division, University of California, Berkeley. His primary interests are computer security and mobile computing.

1556-6013 (c) 2016 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.