Information Systems Frontiers
https://doi.org/10.1007/s10796-020-10066-9

A Formal Specification of Access Control in Android with URI Permissions

Samir Talegaon[1] · Ram Krishnan[1]

Accepted: 10 September 2020

Abstract

A formal specification of access control yields a deeper understanding of any operating system, and facilitates performing security analysis of the OS. In this paper, we provide a comprehensive formal specification of access control in Android (ACiA). Prior work is limited in scope, furthermore, recent developments in Android concerning dynamic runtime permissions require rethinking of its formalization. Our formal specification includes three parts, the user-initiated operations (UIOs) and app-initiated operations (AIOs) - which are distinguished based on the initiating entity, and the URI permissions which are utilized in sharing temporary access to data. We also studied the evolution of URI permissions from API 10 (Gingerbread) to API 22 (Lollipop), and a brief discussion on this is included in the paper. Formalizing ACiA allowed us to discover many peculiar behaviors pertaining to ACiA. In addition to that, we discovered two significant issues with permissions in Android which were reported to Google.

Keywords Android · System permissions · URI permissions · Access control · Formal model

1 Introduction

A formal specification of Access Control in Android (ACiA) facilitates a deeper understanding of the nature in which Android regulates app access to resources. Prior work on formalization of the perm mechanism exists, but is limited in its scope since most of it is based on the older install time perm system (Shin et al. 2010; Fragkaki et al. 2012; Betarte et al. 2015; Bagheri et al. 2015b). Hence, detailed analysis and testing needs to be conducted to build the model for ACiA (ACiA$_\alpha$), to enable a systematic review for security vulnerabilities.

Android contains a wide variety of software resources such as access to the Internet, contacts on the phone, pictures and videos etc., and hardware resources such as Bluetooth, NFC, WiFi, Camera etc. Android apps require the use of such resources, and they request access to them, from the Android OS. Android in turn, seeks user interaction to approve some of these requests and grant the necessary permissions to the apps (https://developer.android.com/training/permissions/requesting/, Enck et al. 2009b) (see Fig. 1). These permissions, provide some protection against unauthorized access of app data, however, research suggests that it is inadequate (Bugiel et al. 2012; Enck et al. 2011; Chin et al. 2011; Davi et al. 2010; Grace et al. 2012; Enck et al. 2009b). This inadequacy can cause issues with privacy and security of the user's data, and requires a formal approach towards that facilitates its analysis.

As mentioned before, Android apps need to request permissions from the user, and this results in the user needing to interact with such prompts. To avoid over-burdening the user with too many permission prompts, Android groups permissions together (based on functionality), and manages these permission groups collectively. These permission groups, however, are immutable, non-overlapping, and only dangerous permissions that fall within a group are granted or revoked from apps (leaving out normal and signature permissions); this makes permission groups in Android under-utilized. We explore several automated means of generating permission groups from the literature, to enhance this feature of Android, and, as a consequence the user experience as well. Besides this, we exploratively include normal permissions in the generated permission groups as well.

Formalization of the ACiA is a non-trivial task, and has received limited prior attention, apart from the fact

✉ Samir Talegaon
samir.talegaon@utsa.edu

1 The University of Texas at San Antonio, One UTSA Circle, San Antonio, TX 78249, USA

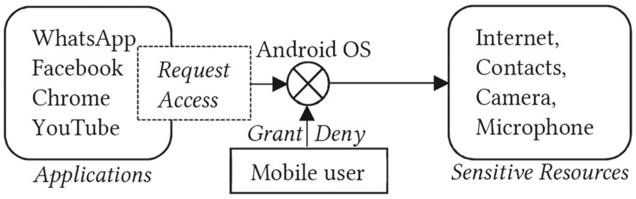


Fig. 1 Permission-based access control in Android

that much of this work has limitations with respect to the current ACiA, due to the major changes in ACiA with the introduction of runtime permissions and non-holistic nature of the work. We believe that the formal specification of access control in Android obtained from the documentation as well as the source code has not been done comprehensively. Our model of the ACiA, and its analysis enables a holistic and systematic review of the ACiA security policies, and facilitates the discovery of issues in the ACiA.

Our Contribution: We provide a formal model for access control in Android with two major components. The first component concerns system permissions in Android, and the second one is related to URI permissions in Android.

- *A comprehensive formal model of access control in Android.*
  - We built a formal model for ACiA (ACiA$_\alpha$), which includes user initiated operations (UIOs) and app-initiated operations (AIOs).
  - We also present two key issues that we encountered while building the ACiA$_\alpha$, pertaining to how custom permissions are handled in Android (https://issuetracker.google.com/issues/128888710, https://issuetracker.google.com/issues/129029397). These issues have been reported to Google and one of these issues has been fixed by Google.
- *A comprehensive study of the evolution of URI permissions in Android.*
  - We tested URI permissions in Android, and provide a detailed explanation of how URI permissions behave. This is required, since there is a lack of understanding of evolution of URI permissions in Android, both, in the literature and in Googles official documentation.
- *An exploratory analysis of mining algorithms to show feasibility of constructing useful permission groups in Android.*
  - We analyze and implement several mining algorithms from the literature, as an alternative approach to build effective permission groups in Android.

OUTLINE: In Section 2, we place our research amongst the current body of works and in Section 3, we describe the ACiA$_\alpha$ including the modifications done post our analysis of the evolution of URI permissions. In Section 4, we describe the testing methodologies used in building the ACiA model. Section 5 presents the anomalies and quirks in the ACiA that were discovered as a result of thorough testing. Section 6 describes our analysis of Android's URI permissions, including the observations made as a result of this meticulous analysis. In Section 7, we discuss permission-groups in Android, and show the feasibility of generating new permission groups via automated mining algorithms (obtained from the literature), and analyze them for suitability with respect to Android. Section 8 concludes with the overview for this paper, and presents the scope for future work.

2 Related Work

Formalizing ACiA has received some attention from prior works. Shin et al. (2010) present a model of ACiA and is one of the few works that is comparable to ours, in modeling the ACiA, including UIOs and AIOs. However, they do not distinguish between multiple competing custom permission definitions, because Android permissions were designed differently at the time. Also, it does not model dangerous runtime permissions nor does it include the URI permissions used to facilitate inter-app data sharing. Fragkaki et al. (2012) developed a formal model to analyze Android permissions and built a system, called as SORBET, to hold a few desired security properties which were not found in the Android's permission model. They also model the ACiA, but their work is centered largely around the URI permission system, with no mention of the UIOs nor the issue with Android's handling of multiple competing custom permission definitions. Apart from this, their work on URI permissions is based on the older versions of the Android OS, which has since undergone many changes, and the URI permissions need to be carefully studied again.

Betarte et al. (2015, 2016, 2017) present a state-based model of the ACiA, which offers pronounced analytical capabilities with respect to security. They define a model state as 8-tuples that record the current state for an Android device, which includes the installed apps, permissions, runtime components, temporary and permanently delegated URI permissions. However, their

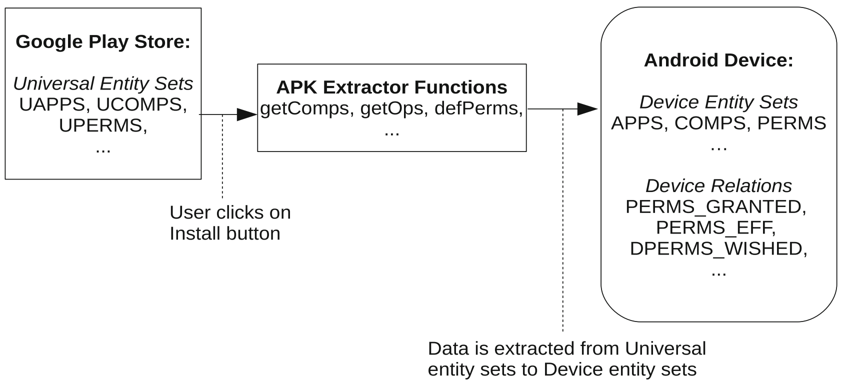

Google Play Store:
Universal Entity Sets
UAPPS, UCOMPS, UPERMS, ...
APK Extractor Functions
getComps, getOps, defPerms, ...
User clicks on Install button
Android Device:
Device Entity Sets
APPS, COMPS, PERMS
...
Device Relations
PERMS_GRANTED, PERMS_EFF, DPERMS_WISHED, ...
Data is extracted from Universal entity sets to Device entity sets

Fig. 2 Building blocks of the ACiA

work does not discuss the UIOs and also does not state the issue of multiple competing custom perm definitions. Bagheri et al. (2015a, 2018) built a formal ACiA model, however, no distinction between developer-defined custom permissions and effectively defined custom permissions are included (see Section 5 Obs - 1). Tuncay et al. (2018) identify that developers should always define custom permissions with the same parameters such as perm-group and protection-level; but, the UIOs proposed by them do not differentiate between multiple competing custom permission definitions. Also, the URI permissions are not included in their model, so this model is insufficient to obtain a holistic understanding of the ACiA.

Enck et al. (2009a) provide an overview of Android's app level security policies and developed a tool (Kirin) to certify whether apps should be installed on a device after comparing the security policy of the apps extracted from their manifest, with the security requirements of the device owner. Shabtai et al. (2010) performed a comprehensive security assessment of Android's framework and made security recommendations based on this assessment. Felt et al. (2011) used Stowaway to determine the over privilege in Android apps which, according to them, was due to developers over requesting the system permissions (based on future, possible need), while at other times, due to developer confusion owing to the insufficient documentation available pertaining to the system APIs. Allowing app developers to specify security policies for their apps which dictate how other apps access their apps information is a novel concept which puts the onus of security on the developers (Ongtang et al. 2012). Some works (Wei et al. 2012; Taylor and Martinovic 2016) also discuss the evolution of the system permissions in Android. Papers such as these add on to work on system permissions, however, formalizing the ACiA is not their main focus.

To conclude, none of the works that model the ACiA, satisfy our requirements for capturing a holistic yet detailed model for the same. To begin with, only a few of the works that model ACiA use a holistic approach like ours, while the rest of them either only model the UIOs or the AIOs, but not both. Furthermore, even the works that employ a holistic approach in building a model for ACiA, are insufficiently detailed to provide a thorough understanding of the ACiA including the URI permissions, to the level of granularity we desire. The model ACiA$_\alpha$ we built, helped us discover two flaws in Android permissions which were reported to Google (https://issuetracker.google.com/issues/128888710, https://issuetracker.google.com/issues/129029397). We were also notified by Google that they fixed one of those flaws (https://issuetracker.google.com/issues/129029397).

3 Formal Specification of Access Control in Android

In this section, we present our formal model for ACiA, ACiA$_\alpha$, starting with the building blocks of the model (see Fig. 2). Some relevant background information is presented, before proceeding.

3.1 Background Information

The four main components of Android apps are activities, services, broadcast receivers and content providers. Components that are utilized within an app, need to be declared in the manifest of that app. Content providers are one such component, that facilitate inter-app data sharing via highly granular URI permissions. Within a content provider definition in the manifest file, we consider two important attributes, namely, android:exported and android:granturipermission. An example syntax for a content provider in the manifest file is shown in Fig. 3. The exported attribute decides whether the content provider is allowed to be accessed by other apps. The granturipermission attribute decides whether the URI permissions can be granted to other apps at run time.

```
<provider android:name="pkgname.Provider"
  android:authorities="pkgname.Provider"
  android:readPermission="pkg.Provider.READ_PERMISSION"
  android:writePermission="pkg.provider.WRITE_PERMISSION"
  android:exported="false"
  android:grantUriPermissions="true"
  android:enabled="true">
```

Fig. 3 Android manifest file

Table 2 APK extractor functions

$\texttt{getComps}: \text{UAPPS} \rightarrow 2^{\text{UCOMPS}}$
$\texttt{getOps}: \text{UCOMPS} \rightarrow 2^{\text{OP}}$
$\texttt{getAuthorities}: \text{UAPPS} \nrightarrow 2^{\text{UAUTHORITIES}}$
$\texttt{getCompPerm}: \text{UCOMPS} \times \text{OP} \nrightarrow \text{PERMS}$
$\texttt{appSign}: \text{UAPPS} \rightarrow \text{USIG}$
$\texttt{defPerms}: \text{UAPPS} \nrightarrow 2^{\text{UPERMS}}$
$\texttt{defPgroup}: \text{UAPPS} \nrightarrow 2^{\text{UPGROUP}}$
$\texttt{defProtlvlPerm}: \text{UAPPS} \times \text{UPERMS} \nrightarrow \text{UPROTLVL}$
$\texttt{defPgroupPerm}: \text{UAPPS} \times \text{UPERMS} \nrightarrow \text{UPGROUP}$
$\texttt{wishList}: \text{UAPPS} \nrightarrow 2^{\text{UPERMS}}$

3.2 Element Sets, Functions and Relations of the ACiA$_\alpha$

Our model for ACiA (ACiA$_\alpha$) was built after studying the documentation (https://developer.android.com/training/permissions/requesting/, https://source.android.com/devices/tech/config), reading Android's source code and verifying our findings via inter-app tests. The major classifications for entity sets, relations and functions are explained below in brief (for a comprehensive description of universal sets, device dets, relations and the functions see Talegaon and Krishnan (2019), Section 3.1).

Application data such as app names, permissions, app component names are stored on the Google Play, and on an Android device (see Table 1). The data stored by Google is mimicked by the universal sets, whereas, the data stored on an Android device, is mimicked by device sets. These sets are populated by Google along with the app developers and are assumed to be immutable for the purposes of this paper. To install apps, Android uses many different APIs which we summarize as APK Extractor Functions (see Fig. 2 and Table 2). These functions retrieve information from the apps that are being installed on the device; evidently, the relations maintained on the device are not useful for these functions. Upon successful installation, all the necessary device entity sets and relations (see Table 3) are updated as shown in Table 5 (InstallApp operation). The device relations (3, Column - 1) portray the information stored by an Android device to facilitate access control decisions. Convenience functions (Table 3, Column - 2) query existing relations maintained on the device; evidently, these functions fetch information based on apps that are already installed on the device. Similarly, to facilitate app un-installation, the helper functions (see Table 4) extract data from the device sets and relations. Many other operations take place during the normal course of operation of an app, and, this is portrayed by the UIOs and AIOs that mimic built-in methods such as RequestPermission, GrantPermission and GrantUriPermission. The operations within the UIOs and AIOs are assumed to be in-order, and, the relations do not automatically get updated, when the constituent sets or relations undergo a change.

Table 1 ACiA entity sets

Universal entity sets	Device entity sets
UAPPS	APPS
UCOMPS	COMPS
UAUTHORITIES	Authorities
UPERMS	PERMS
USIG	–
UPGROUP	PGROUP
UPROTLVL	PROTLVL
–	DATAPERMS
–	URI
–	OP

3.3 User Initiated Operations

ACiA$_\alpha$ - UIOs are initiated by the user or require their approval before they can be executed. These are discussed in this section; the detailed updates are available in Table 5.

The **AddApp** operation resembles the user clicking on "install" button on the Google Play Store, and upon successful execution, the requested app is installed on the device. The **DeleteApp** operation resembles a user un-installing an app from the Settings app. The **GrantDangerPerm** and **GrantDangerPgroup** operations resemble the user granting a dangerous perm/perm-group to an app via the Settings app; and, the execution of this operation result in an app receiving a dangerous perm/perm-group, respectively. The **RevokeDangerPerm** and **RevokeDangerPgroup** operations resemble the user revoking a dangerous perm or perm-group from an app via the Settings app; and, their execution results in an app's dangerous perm/perm-group getting revoked.

3.4 App Initiated Operations

The AIOs are initiated by the apps when attempting to perform several tasks (see Table 6) such as requesting a perm from the user, granting a URI permission to another app, revoking a URI permission from all apps etc.

Table 3 ACiA relations and convenience functions

APP_COMPS ⊆ APPS × COMPS	ownerApp : COMPS → APPS appComps : APPS → 2^{COMPS}
COMP_PROTECT ⊆ COMPS × OP × PERMS	requiredPerm : COMPS × OP $\nrightarrow$ PERMS allowedOps : COMPS → 2^{OP}
AUTH_OWNER ⊆ APPS × AUTHORITIES	authoritiesOf : APPS → $2^{\text{AUTHORITIES}}$
PERMS_DEF ⊆ APPS × PERMS × PGROUP × PROTLVL	defApps : PERMS → 2^{APPS} defPerms : APPS → 2^{PERMS} defPgroup : APPS × PERMS $\nrightarrow$ PGROUP defProtlvl : APPS × PERMS → PROTLVL
PERMS_EFF ⊆ APPS × PERMS × PGROUP × PROTLVL	effApp : PERMS → APPS effPerms : APPS → 2^{PERMS} effPgroup : PERMS $\nrightarrow$ PGROUP effProtlvl : PERMS → PROTLVL
DPERMS_WISHED ⊆ APPS × PERMS	wishDperms : APPS → 2^{PERMS}
PERMS_GRANTED ⊆ APPS × PERMS	grantedPerms : APPS → 2^{PERMS}
GRANTED_DATAPERMS ⊆ APPS × URI × DATAPERMS	grantNature: APPS × URI × DATAPERMS → **SemiPermanent, Temporary, NotGranted**} uriPrefixCheck : APPS × URI × DATAPERMS → $\mathbb{B}$

The **RequestPerm** operation resembles an app requesting a dangerous system perm from the Android OS. The **CheckAccess** operation resembles a component attempting to perform an operation on another component; components may belong to the same or distinct apps. Finally, the **AppShutdown** operation resembles an app shutting down, so all the temporary URI permissions granted to it are revoked unless they are persisted.

4 Experimental Setup

After we extract the model for ACiA using source code and developer documentation, testing was done via carefully designed inter app tests. These tests enabled the discovery of the flaws that are stated in the next section, apart from helping us understand the intricate details of operations such as app installation, uninstallation, perm grants and revocation, URI permission grants and revocation etc. We explain in brief, the test setup for studying evolution of Android's URI permissions, in this section.

Rationale for Testing. Mathematical models mitigate ambiguity in access control; documentation and source codes can be open to interpretation. Differences in interpretation leads to a plunge in accuracy of stated operations, which in turn leads to inaccurate predictions based on that interpretation. Since our entire model for ACiA depends on reading the source code and documentation, testing was performed (on an emulator) to ensure that our model is in line with the behavior of the Android OS and that of Android apps. Apart from this, we made several predictions

Table 4 Helper functions

userApproval : APPS × PERMS → $\mathbb{B}$
brReceivePerm : COMPS → PERMS
corrDataPerm : PERMS → $2^{\text{URI} \times \text{DATAPERMS}}$
checkExported : URI → $\mathbb{B}$
checkGrantUriPermission : URI → $\mathbb{B}$
belongingAuthority : URI → AUTHORITIES
requestApproval : APPS × APPS × URI → $2^{\text{DATAPERMS}_b}$
grantApproval : APPS × APPS × URI × $2^{\text{DATAPERMS}}$ → $\mathbb{B}$
prefixMatch : APPS × URI × DATAPERMS → $\mathbb{B}$
appAuthorized : APPS × URI × DATAPERMS → $\mathbb{B}$

Table 5 ACiA$_\alpha$ User Initiated Operations

Operation: **AddApp**$(ua : \text{UAPPS})$

Authorization Requirement: $\forall up \in \text{PERMS} \cap \texttt{defPerms}(ua).$

$$\texttt{appSign}\big(\texttt{effApp}(up)\big) = \texttt{appSign}(ua) \wedge \texttt{getAuthorities}(ua) \cap \bigcup_{a \in \text{APPS}} \texttt{getAuthorities}(a) = \emptyset$$

Updates:

$\text{APPS}' = \text{APPS} \cup \{ua\}$**;** $\text{COMPS}' = \text{COMPS} \cup \texttt{getComps}(ua)$

$\text{APP\_COMPS}' = \text{APP\_COMPS} \cup \{ua\} \times \texttt{getComps}(ua)$

$\text{AUTHORITIES}' = \text{AUTHORITIES} \cup \texttt{getAuthorities}(ua)$

$\text{AUTH\_OWNER}' = \text{AUTH\_OWNER} \cup \{ua\} \times \texttt{getAuthorities}(ua)$

$$\text{PERMS\_DEF}' = \text{PERMS\_DEF} \cup \bigcup_{up \in \texttt{defPerms}(ua)} \Big\{\Big(ua,\ up,\ \texttt{defPgroupPerm}(ua, up),\ \texttt{defProtlvlPerm}(ua, up)\Big)\Big\}$$

$$\text{PERMS\_EFF}' = \text{PERMS\_EFF} \cup \bigcup_{up \in \texttt{defPerms}(ua) \setminus \text{PERMS}} \Big\{\Big(ua,\ up,\ \texttt{defPgroupPerm}(ua, up),\ \texttt{defProtlvlPerm}(ua, up)\Big)\Big\}$$

$\text{PERMS}' = \text{PERMS} \cup \texttt{defPerms}(ua)$

$$\text{COMP\_PROTECT}' = \text{COMP\_PROTECT} \cup \bigcup_{\substack{c \in \texttt{appComps}(a); op \in \texttt{getOps}(c);\\ p \in \text{PERMS} \cap \texttt{getCompPerm}(op,\ c)}} \big\{(c,\ op,\ p)\big\}$$

$\text{PGROUP}' = \text{PGROUP} \cup \texttt{defPgroup}(ua)$

$$\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \cup \bigcup_{\substack{a' \in \text{APPS}\\ up \in \texttt{wishList}(a') \cap \text{PERMS s. t.}\\ \texttt{effProtlvl}(up) = \textbf{normal}}} \{(a',\ up)\} \cup \bigcup_{\substack{a' \in \text{APPS};\ up \in \texttt{wishList}(a') \cap \text{PERMS}\\ \text{s. t.}\big(\texttt{effProtlvl}(up) = \textbf{signature} \wedge\\ \texttt{appSign}(\texttt{effApp}(up)) = \texttt{appSign}(a')\big)}} \{(a',\ up)\}$$

$$\text{DPERMS\_WISHED}' = \text{DPERMS\_WISHED} \cup \bigcup_{\substack{a' \in \text{APPS};\ up \in \texttt{wishList}(a') \text{ s. t.}\\ \texttt{effProtlvl}(up) = \textbf{dangerous}}} \{(a',\ up)\}$$

Operation: **DeleteApp**$(a : \text{APPS})$

Authorization Requirement: **T**

Updates:

$$\text{COMP\_PROTECT}' = \text{COMP\_PROTECT} \setminus \bigcup_{\substack{c \in \texttt{appComps}(a)\\ op \in \texttt{allowedOps}(c)\\ p \in \text{PERMS} \cap \texttt{getCompPerm}(op,\ c)}} \big\{(c,\ op,\ p)\big\} \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\};\ c \in \texttt{appComps}(a')\\ op \in \texttt{allowedOps}(c)\\ p \in \texttt{effPerms}(a) \cap \texttt{requiredPerm}(c,\ op)}} \{(c,\ op,\ p)\}$$

$\text{AUTH\_OWNER}' = \text{AUTH\_OWNER} \setminus \{a\} \times \texttt{authoritiesOf}(a)$

$\text{AUTHORITIES}' = \text{AUTHORITIES} \setminus \texttt{authoritiesOf}(a)$

$\text{COMPS}' = \text{COMPS} \setminus \texttt{appComps}(a)$**;** $\text{APP\_COMPS}' = \text{APP\_COMPS} \setminus \{a\} \times \texttt{appComps}(a)$

$$\text{PERMS}' = \text{PERMS} \setminus \Bigg(\texttt{effPerms}(a) \setminus \bigcup_{a' \in \text{APPS} \setminus \{a\}} \texttt{defPerms}(a')\Bigg)$$

$$\text{PGROUP}' = \text{PGROUP} \setminus \Bigg(\texttt{defPgroup}(a) \setminus \bigcup_{a' \in \text{APPS} \setminus \{a\}} \texttt{defPgroup}(a')\Bigg)$$

$$\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \setminus \Bigg(\{a\} \times \texttt{grantedPerms}(a) \cup \bigcup_{a' \in \text{APPS} \setminus \{a\};\ p \in \texttt{effPerms}(a)} \{(a',\ p)\}\Bigg)$$

$$\text{DPERMS\_WISHED}' = \text{DPERMS\_WISHED} \setminus \Bigg(\{a\} \times \texttt{wishDperms}(a) \cup \bigcup_{a' \in \text{APPS} \setminus \{a\};\ p \in \texttt{effPerms}(a)} \{(a',\ p)\}\Bigg)$$

$$\text{PERMS\_EFF}' = \Bigg(\text{PERMS\_EFF} \setminus \bigcup_{p \in \texttt{effPerms}(a)} \Big\{\Big(a,\ p,\ \texttt{effPgroup}(p),\ \texttt{effProtlvl}(p)\Big)\Big\}\Bigg) \cup \bigcup_{p \in \texttt{effPerms}(a')} \Big\{\Big(a',\ p,\ \texttt{defPgroup}(a',\ p), \texttt{defProtlvl}(a',\ p)\Big)\Big\}, \text{ whr. } a' \in \texttt{defApps}(p) \setminus \{a\}$$

$$\text{PERMS\_DEF}' = \text{PERMS\_DEF} \setminus \bigcup_{p \in \texttt{defPerms}(a)} \Big\{\Big(a,\ p,\ \texttt{defPgroup}(a,\ p),\ \texttt{defProtlvl}(a,\ p)\Big)\Big\}$$

Table 5 (continued)

$\text{COMP\_PROTECT}' = \text{COMP\_PROTECT} \ \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ c \in \texttt{appComps}(a');\ op \in \texttt{allowedOps}(c) \\ p \in \text{PERMS} \cap \texttt{getCompPerm}(op,\ c)}} \{(c,\ op,\ p)\}$ $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \ \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ p \in \texttt{wishList}(a') \cap \text{PERMS s. t.} \\ \texttt{effProtlvl}(p) = \textbf{normal}}} \{(a',\ p)\} \ \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ p \in \texttt{wishList}(a') \cap \text{PERMS s. t.} \\ \big(\texttt{effProtlvl}(p) = \textbf{signature} \ \wedge \\ \texttt{appSign}\big(\texttt{effApp}(p)\big) = \texttt{appSign}(a')\big)}} \{(a',\ p)\}$ $\text{DPERMS\_WISHED}' = \text{DPERMS\_WISHED} \ \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ p \in \texttt{wishList}(a') \text{ s. t.} \\ \texttt{effProtlvl}(p) = \textbf{dangerous}}} \{(a',\ p)\}$ $\text{APPS}' = \text{APPS} \ \setminus \ \{a\}$
Operation: **GrantDangerPerm**$(a : \text{APPS},\ p : \text{PERMS})$ Authorization Requirement: $p \in \texttt{wishDperms}(a)$ Update: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \ \cup \ \{(a, p)\}$
Operation: **GrantDangerPgroup**$(a : \text{APPS},\ pg : \text{PGROUP})$ Authorization Requirement: $\exists p \in \texttt{wishDperms}(a).\ \texttt{effPgroup}(p) = pg$ Update: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \ \cup \bigcup_{p \in \texttt{wishDperms}(a) \text{ s. t. } \texttt{effPgroup}(p) = pg} \{(a, p)\}$
Operation: **RevokeDangerPerm**$(a : \text{APPS},\ p : \text{PERMS})$ Authorization Requirements: $p \in \texttt{grantedPerms}(a) \ \wedge \ \text{p} \in \texttt{wishDperms}(a)$ Update: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \ \setminus \ \{(a,\ p)\}$
Operation: **RevokeDangerPgroup**$(a : \text{APPS},\ pg : \text{PGROUP})$ Authorization Requirements: $\exists p \in \texttt{grantedPerms}(a).\ \texttt{effPgroup}(p) = pg \ \wedge \ \text{p} \in \texttt{wishDperms}(a)$ Update: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \setminus \bigcup_{\substack{p \in \texttt{grantedPerms}(a) \text{ such that} \\ \big(\texttt{effPgroup}(p) = pg \ \wedge p \in \texttt{wishDperms}(a)\big)}} \{(a,\ p)\}$

based on the model, and then verified them using these tests, and this methodical procedure that enabled us to discover flaws in the design of ACiA that were communicated to Google (https://issuetracker.google.com/issues/128888710, https://issuetracker.google.com/ https://issuetracker.google.com/). Google communicated that one of these flaws were fixed and has acknowledged the other flaw.

Experimental Setup for Building ACiA$_\alpha$. A simple three app based testing environment was designed, which was adapted for each individual test. The apps used for these tests were dummy apps with two activities and one service component. According to need, the apps were programmed to define a new perm using one of the available protection levels, or, into a hitherto undefined permission group.

Test Parameters. A total of four test parameters (TP) are considered (see Table 7) which include installation procedure for an app, uninstallation procedure for an all, installation sequence formultiple apps and uninstallation sequence for multiple apps. A few simple tests that were conducted to verify our findings using test apps are described below.

1. **Verifying authorization requirements for the AddApp operation.** The AddApp operation mimics the app installation procedure in Android, and several checks are required to pass before the installation can proceed.

   Checks found via the source code and documentation.

   (a) All custom permissions need to be unique.
   (b) If custom perm already exists, signatures of the apps should match.
   (c) All authorities also need to be unique.

Table 6 ACiA$_\alpha$ App Initiated Operations

Operation: **RequestPerm**$(a : \text{APPS},\ p : \text{PERMS})$
Authorization Requirement: $(a, p) \in \text{DPERMS\_WISHED} \quad \wedge$
$\Big(\big(\exists p' \in \text{PERMS} \setminus \{p\}.\ \texttt{effPgroup}(p') = \texttt{effPgroup}(p) \wedge (a, p') \in \text{PERMS\_GRANTED}\big) \quad \vee \quad \texttt{userApproval}(a, p)\Big)$
Updates: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \cup \{(a, p)\}$

Operation: **CheckAccess**$(c_{src} : \text{COMPS},\ c_{tgt} : \text{COMPS},\ op : \text{OP})$
Authorization Requirement:
$\texttt{ownerApp}(c_{src}) = \texttt{ownerApp}(c_{tgt}) \vee \Big(op \in \texttt{allowedOps}(c_{tgt}) \wedge \texttt{requiredPerm}(c_{tgt}, op) \in \texttt{grantedPerms}\big(\texttt{ownerApp}(c_{src})\big) \wedge \big(op = \textbf{sendbroadcast} \wedge \texttt{brReceivePerm}(c_{src})\big) \Rightarrow \texttt{brReceivePerm}(c_{src}) \subseteq \texttt{grantedPerms}\big(\texttt{ownerApp}(c_{tgt})\big)\Big)$
Update: -

Operation: **AppShutdown**$(a : \text{APPS})$
Authorization Requirement: **T**
Updates: $\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \setminus \bigcup_{\substack{(a,\ uri,\ dp) \in \text{GRANTED\_DATAPERMS such that} \\ \texttt{grantNature}(a,\ uri,\ dp) = \textbf{Temporary}}} \{(a,\ uri,\ dp)\}$

Verification methodology. For this test, we designed three test apps that each define the same perm, however, two are signed with the same certificate, whereas the third is signed with a different certificate. Upon attempting installation we encountered the following.

> Case for defining new perm (see Fig 4): Apps 1 and 2 could be installed even though they re-defined the same perm, however, App3's installation could not proceed since it was signed with a certificate from a different developer.

2. **Check whether perm definitions were changed in accordance to the apps that were present on a device.** The three apps mentioned above, were designed to define a single perm, into 3 distinct perm groups i.e.: pgroup1, pgroup2 and pgroup3. These apps were installed on an Android emulator multiple times, and each time their install order was changed. After each installation we checked the perm attributes via adb commands and discovered the issue 1 in Section 5 as discussed in the next section.

5 Observations

In this section, we discuss the observations made after the ACiA model was built. Our analysis of ACiA$_\alpha$ yields some interesting and peculiar observations; and, after a thorough review of the same, we derived the rationale behind these observations and make predictions based on them. Testing these predictions yield a number of potential flaws in ACiA, which were reported to Google. Every important observation was verified using test-apps, and the final model is designed to capture all the important aspects of ACiA. Below we note a few such important observations and the operations where they were encountered.

1. **Undefined behavior in case of competing custom perm definitions.** Android allows multiple definitions of the same perm (from apps signed with the same certificate) to co-exist on a device. The effective definition for such a perm is taken from the first app that defines it; any subsequent definitions of the same permission are ignored. This can cause issues when that app that defined the permission is un-installed,

Table 7 Test parameters used for ACiA$_\alpha$ model evaluation

TP1[a]	*Install Procedure* e.g.:$adb push and then use GUI for installation, or $adb uninstall
TP2	*Uninstall Procedure* e.g.:$adb uninstall, or Use GUI for uninstallation
TP3	*Install order* e.g.: install App1, App2, App3; or install App2, App1, App3; or install App3, App2, App1
TP4	*Uninstall order* e.g.: uninstall App3, App2, App1; or uninstall App1, App2, App3; or uninstall App2, App1, App3

[a]TP:Test Parameter

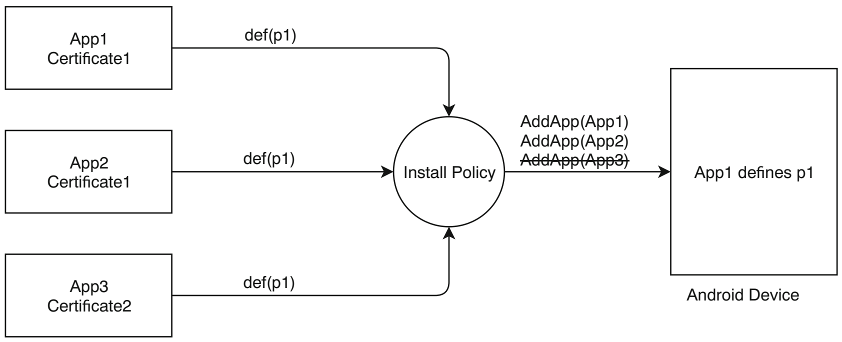


Fig. 4 App installation authorization requirement - New permission definition

since there is no order with which Android changes the definition of the perm, hence, the perm definition randomly jumps from the un-installed app, to any other app that defined that permission (see Fig. 5).

2. **Normal permissions are never re-granted after app un-installation.** According to Android, **normal** and **signature** permissions are defined to be install-time permissions by Android, so, when multiple apps define the same perm, app un-installation results in any new normal permissions to be not granted to apps. Signature permissions are automatically re-granted by Android.
3. **Apps can re-grant temporary URI permissions to themselves permanently**. Android enables apps to share their data via content providers, temporarily (using intents with URI permissions), or semi-permanently (using the grantUriPermissions) method. When an app receives a temporary URI perm, it can even grant this perm to any other app temporarily or semi-permanently. This is clearly a flaw as no app can control this style of chain URI perm grants; this flaw is not exactly new and was discovered a few years ago (Fragkaki et al. 2012).
4. **Custom perm names are not enforced using the reverse domain style.** Although Google recommends developers use the reverse domain style naming convention for defining custom-permissions, this isn't enforced by Google (see Fig. 6). This can lead to unwanted behavior when a new app fails to install, because it attempted to redefine a perm that already exists on the device (new app is from a different developer).
5. **Complex custom perm behavior upon app un-installation**. During app un-installation, extensive testing was done to ensure that we captured an accurate behavior for Android. Care was taken while removing perm definitions, since only if there are no other apps defining the same perm, is that perm removed from the system. For this test case we

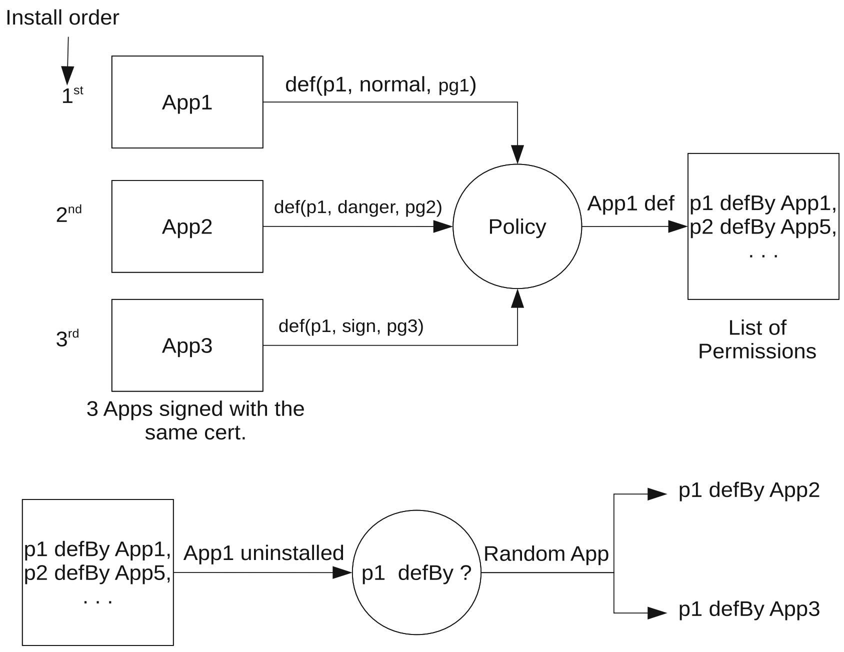


Fig. 5 Anomaly in Android custom permissions

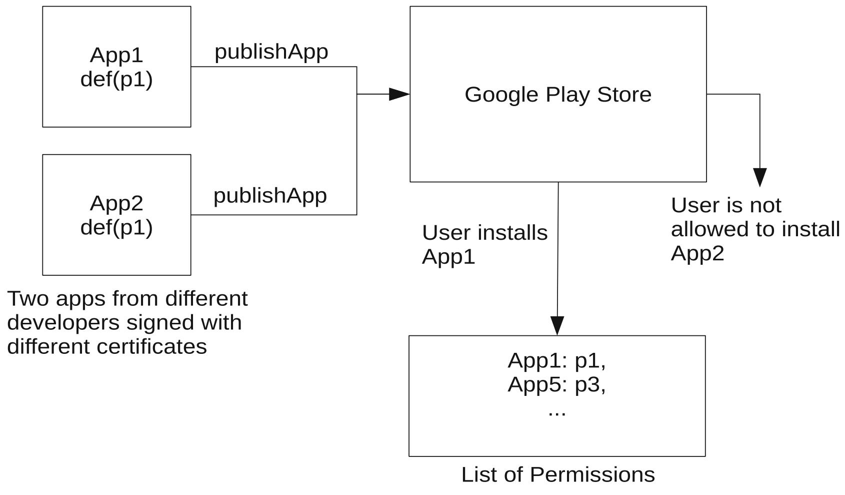


Fig. 6 Drawback of not enforcing custom permission Names

constructed 3 test apps and performed worst case testing with respect to perm definitions and found Issue #2 described above. This is a grave issue since the documentation states that all normal perms are always granted when their apps are installed on the device (Table 8).

6 Android's URI Permissions

In this section, we describe the experimental setup used for analyzing URI permissions in Android, and, the observations that were made as a result. While the URI permissions changed dramatically over the course of APIs 10 onwards, the system permissions did not undergo extensive smaller changes except for the change from install time permissions to runtime permissions. There is a lack of understanding of how URI permissions work in Android. Also, since URI permissions have changed across APIs, and, because at any given point there is a mix of APIs deployed on Android devices (see Table 9), it is critical we understand how URI permissions work, across various APIs.

In order to thoroughly understand the intricate behavior of Android's URI permission system we built 6 dummy apps, and performed a total of 96 tests based on the following parameters for every API in consideration. The APIs under consideration for this analysis were - 10, 16-19 and 21-22.

6.1 Experimental Setup

We designed and used a total of six apps as described in Table 10. These apps were deployed on an emulator from Android studio.

Parameters for testing. The parameters for testing are shown below.

I **Attributes in the manifest file.** The first parameter for testing is the boolean value for the exported and granturipermission attribute.

II **App scenarios.** The second parameter comprises of 4 distinct testing scenarios for the 6 apps noted above (see Table 8). The scenarios are designed to incorporate all the most common app usage in the real world, involving apps with a content provider, manifest-apps

Table 8 App scenarios for URI permission testing

Scenario	Apps	Explanation
1	Owner-app, A5, A6	Owner-app declared perm to guard its content provider and requests this perm in the manifest. The URI perm to this content provider maybe granted to other apps.
2	Owner-app, Manifest-App, A3, A4, A5, A6	Owner-app declares and requests perm to its content provider in the manifest. Manifest-app requests access to this perm in its manifest and is granted to it at install time. Other apps may receive access to this content provider at runtime.
3	Owner-app, Manifest-App, A3, A4, A5, A6	Owner-app declared perm to its content provider but does not request it in the manifest. Manifest-app requests perm to this content provider in its manifest. Other apps may be granted the URI perm to this content provider at runtime.
4	Owner-app, Manifest-App, A3, A4, A5, A6	Owner-app does not declare perm to guard its content provider. No other apps can request perm to this content provider since it does not exist.

Table 9 Android device API distribution

Android version	Percentage of devices
Jelly bean API 18	98.4%
KitKat API 19	98.1%
Lollipop API 22	92.3%
Marshmellow API 23	84.9%
Nougat API 25	66.2%
Orea API 27	53.5%
Pie API 28	39.5%
Android 10	8.2%

with perm to a content provider and other apps with URI permissions to the content provider.

III **Key questions pertaining to Android URI permissions.** The third and final parameter of our study consists of six questions to establish the scope of URI perm, their re-delegation and revocation in Android apps. These questions (see Table 10) pertain to the granting, revocation, and delegation of URI permissions in Android.

6.2 Observations from the Study of Evolution of Android's URI Permissions

Extensive testing was done on Android's URI permissions, while seeking the answers to some intriguing questions mentioned above. These tests yield some interesting observations that are described below.

Case I *(Exported = False, GrantUriPermission = False)*

Answers 1 - 6: As the exported and GrantUriPermission attributes are false, the Manifest-apps and Delegated-apps do not get any access the App A1's URI permissions (no other app except Owner-app gets access to the URI and owner cannot delegate any permission)

Case II *(Exported = False, GrantUriPermission = True)*

Answers 1, 2: The Manifest-app does not have access to the URI permissions, however, Owner-apps have access to these permissions and can delegate them to delegated apps. The delegated-apps can keep the URI permissions until Device is Rebooted, App-stack finishes or the RevokeUriPermission method is invoked for URI permissions granted via Intents; and until Device is Rebooted or RevokeUriPermission method is invoked for URI permissions granted via the grantUriPermission.

Answers 3 - 6: The Delegated-apps receive perm from Owner-app and can delegate this perm using Intents and GrantUriPermission. The Delegated-app can receive delegated permission from Owner-app → DA-1 → DA-2 which they keeps until - Device Reboots, the App-Stack finishes, or the RevokeUriPermission method is invoked for URI permissions granted with Intents; and until, Device Reboots, the RevokeUriPermission method is invoked for URI permissions granted via the grantUriPermission method. Manifest-apps do not have perm to URI and cannot revoke URI permissions, but the Owner apps can revoke them. Only the Owner-app can revoke URI permissions via the RevokeUriPermission method.

Case III *(Exported = True, GrantUriPermission = False)*

Answer 1: No app can grant URI permission as grantUriPermission is false.

Answers 2 - 6: Delegated app does not exist on this case.

Case IV *(Exported = True, GrantUriPermission = True)*

Answers 1 - 3: Manifest-app can grant the URI permissions via Implicit Intent using syntax intent.set Action ("Action string"). The Delegated-apps keeps the URI perm until their app-stack finish, RevokeUriPermission

Table 10 Apps used for testing URI permissions

Apps	App description
Owner-app (A1)	App having a content provider to share its data.
Manifest-App (A2)	App requesting the system perm to A1s content provider in its manifest.
Delegated-apps (A3, A4, A5, A6)	Dummy apps used for testing, and having three activities each with methods for URI perm re-delegation to one another. These apps do not request the URI permissions at install time and, depending on the scenario, may be granted permissions to the URI via Intents or GrantUriPermission method.

method is invoked, or, until the device reboots. The Delegated-apps can re-delegate the URI permissions via Intents and the GrantUriPermission method.

Answer 4 - 6: The Delegated-apps keep the URI perm until app stack finishes or until the Manifest-app or Owner-app invoke the revokeUriPermission method or until Device Reboots. Only Manifest-apps and Owner-apps can revoke URI permissions. URI permissions can be revoked using the revokeUriPermission method which revokes URI perm from ALL apps.

6.3 Answers to Key Questions

The questions that were put forth in Section 6.1 (Table **??**) are answered below.

1. **Which app is allowed to delegate URI permissions to other apps?** Before API 16, only those apps which requested URI permissions in the manifest were allowed to delegate them to other apps. After API 16 the Owner-app received access to grant its own URI permissions.
2. **Once URI permissions are delegated to an app, how long will the permission remain with the app?** Once a URI permission is delegated to an app via an Intent, it remains with that app until all its activities have ended which means that none of its activities are running in the task stack. URI permissions granted via the GrantUriPermissions method, remain with the app until the RevokeUriPermissions method is onvoked. It should be noted that all such permissions are automatically revoked when a device is rebooted, unless the URI perm is granted with intent containing the *persist* tag.
3. **Can an app to which temporary URI permissions were granted, re-delegate these permissions to other apps, and does any conditions exist on the scope of such re-delegation?** The URI permissions which the Delegated-apps have, can be further re-delegated by those apps using Intents and GrantUriPermissions method, without any restrictions.
4. **What are the key changes in URI permissions with respect to their delegation, re-delegation and revocation?** It is be surprising that in API 10 the Owner app did not get any access to its own content provider to delegate its perm to other apps. This meant that the Owner app must request permissions to its own content provider to enable it to delegate it to other apps! In API 16 this was changed to allow the Owner-apps to gain access to their own URI, which allowed them to delegate URI permissions to other apps. No major changes were detected after API 16 to API 22 (Table 11).

6.4 Current App Initiated Operations for URI Permissions

Post performing the evaluation study for Android's URI permissions, we present the operations for the latest Android API 29 in Table 12. The **RequestDataPerm** operation denotes the URI permission requests by apps. The **GrantDataPerm** operation resembles the URI perm delegation by apps; and, it only succeeds if the app trying to grant the permissions has the necessary access. The **RevokeDataPerm** operation resembles the revocation of URI perm from an installed app. The **RevokeGlobalDataPerm** operation is similar to the **RevokeDataPerm** except that it revokes the URI permissions from all apps on the device. The **CheckDataAccess** operation checks if a particular app has access to a URI. URI permissions are delegated to apps by other app possessing those permissions.

7 Permission Groups

In this section, a few algorithms for mining permission groups in Android are discussed. Permission groups are a step towards increasing user friendliness of access control in Android, by reducing the number of prompts, to which users are required to respond. However, permission groups currently deployed in Android are immutable, non-overlapping and are thus rigid. On the other hand, it is possible to have permission groups that employ an overlapping structure which have benefits such as, being able to moderate the trade-off with respect to the cardinality of permissions within a group, and the cardinality of the set of permission groups themselves, according to the requirement. The mining algorithms we use to generate permission groups, employ a bottom up approach (Vaidya et al. 2006), in which, algorithms are used to generate groups of permissions from a user-permission assignment (UPA) matrix. Various algorithms for mining groups from the UPA matrix have been published, and we have implemented and analyzed a few of such algorithms. It should be noted that the results presented in this section merely point towards the feasibility of a permission group based architecture for Android, and these permission groups are not perfect for every need. Accordingly, permission groups can be generated tailored to the demand, and can be incorporated in Android.

Table 11 Questions for testing URI permissions

1	How do apps grant URI permissions to other apps which do not have them?
2	How long do the delegated-apps keep their permissions, once granted?
3	How does URI permission re-delegation work with regards to delegated-apps?
4	How long do the URI permissions being re-delegated by delegated-apps to others last?
5	Which app can revoke delegated permissions?
6	How can URI permissions be revoked?

7.1 Permission Group Mining Algorithms

The mining algorithms that were implemented from their pseudo-code to generate the permission groups are described below.

7.1.1 Fast-Miner and Complete-Miner Algorithm

The Fast-Miner (FM) and Complete-Miner (CM) (Vaidya et al. 2006) are algorithms, that generate a large set of candidate permission groups. The FM algorithm restricts the intersections between users to a maximum of two, and its output consists of more than 14,000 candidate permission groups. This algorithm executes fully within a reasonable amount of time, and, we use its output to extract permission groups for Android (see Table 13a). The CM algorithm, however, has no such restriction, and so, it takes a significant amount of time to execute generating over 200,000 permission groups.

7.1.2 Basic-RMP Algorithm

The Basic-RMP algorithm (Vaidya et al. 2007) is adapted from the largest uncovered tile mining algorithm (LUTM) (Geerts et al. 2004) defined for databases. It greedily discovers largest uncovered tiles from the UPA, to construct the permission groups. The output of this algorithm contains many permission groups consisting of singular permissions, and, this is due to the nature of the algorithm. While the algorithm ensures that each permission is assigned to at-least one permission group, the tiles considered by the algorithm are contiguous and do not provide optimal area

Table 12 ACiA$_\alpha$ App Initiated Operations for URI permissions

Operation: **RequestDataPerm**$(a_{src} : \text{APPS},\ a_{tgt} : \text{APPS},\ uri : \text{URI})$

Authorization Requirement: $\texttt{requestApproval}(a_{src}, a_{tgt}, uri) \neq \emptyset$

Updates: $$\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \cup \bigcup_{dp \in \texttt{requestApproval}(a_{src}, a_{tgt}, uri)} \{(a_{src}, uri, dp)\}$$

Operation: **GrantDataPerm**$\left(a_{src} : \text{APPS},\ a_{tgt} : \text{APPS},\ uri : \text{URI},\ dp : 2^{\text{DATAPERMS}}\right)$

Authorization Requirement: $\texttt{grantApproval}(a_{src}, a_{tgt}, uri, dp)$

Updates: $\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \cup (a_{tgt}, uri) \times dp$

Operation: **RevokeDataPerm**$(a_{src} : \text{APPS},\ a_2 : \text{APPS},\ uri : \text{URI},\ dp : \text{DATAPERMS})$

Authorization Requirement 1: $\neg\psi$

Update 1: $\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \setminus \{(a_{src}, uri, dp)\}$

Authorization Requirement 2: ψ

Update 2: $\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \setminus \{(a_{tgt}, uri, dp)\}$

where $$\psi :\equiv \quad a_{src} = \texttt{ownerOf}\left(\texttt{belongingAuthority}(uri)\right) \lor \exists p \in \texttt{grantedPerms}(a_{src}).\ (uri, dp) \in \texttt{corrDataPerm}(p)$$

Operation: **RevokeGlobalDataPerm**$(a : \text{APPS},\ uri : \text{URI})$

Authorization Requirement: ψ

Update: $$\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \setminus \bigcup_{a \in \text{APPS}} \{(a, uri, dp)\}$$

Operation: **CheckDataAccess**$(a : \text{APPS},\ uri : \text{URI},\ dp : \text{DATAPERMS})$

Authorization Requirement: $\texttt{appAuthorized}(a, uri, dp)$

Update: -

Table 13 Mined permission groups

Perm groups	Assigned permissions
(a) FM/CM	
PG1	INTERNET, ACCESS_NETWORK_STATE
PG2	com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, WAKE_LOCK
PG3	INTERNET, READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.-RECEIVE, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, WAKE_LOCK
PG4	INTERNET
(b) Basic RMP	
PG1	WRITE_CONTACTS, GET_ACCOUNTS, READ_CONTACTS
PG2	WRITE_CONTACTS, GET_ACCOUNTS, READ_CONTACTS, READ_CALL_LOG
PG3	WRITE_CONTACTS
PG4	GET_ACCOUNTS
(c) Delta RMP	
PG1	WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, WAKE_LOCK
PG2	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, INTER-NET, ACCESS_NETWORK_STATE, VIBRATE, ACCESS_WIFI_STATE
PG3	INTERNET, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE
PG4	GET_ACCOUNTS, READ_PHONE_STATE, CAMERA, ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, VIBRATE,WAKE_LOCK
(d) MinNoise RMP	
PG1	WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, WAKE_LOCK
PG2	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, INTER-NET, ACCESS_NETWORK_STATE, VIBRATE, ACCESS_WIFI_STATE
PG3	INTERNET, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE
PG4	GET_ACCOUNTS, READ_PHONE_STATE, CAMERA, ACCESS_FINE_LOCATION, ACCESS_COARSE_LOCATION, WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, VIBRATE, WAKE_LOCK

with respect to the entire UPA. Due to the permission groups consisting of a single permission, which are generated by this algorithm, the output of this algorithm do not satisfy our requirements.

7.1.3 δ Approx.-RMP Algorithm

The δ-approximation algorithm (Vaidya et al. 2010) generates permission groups by using the candidate permission groups, generated by the Fast Miner algorithm as input, and, greedily picking the best candidate permission group until the original UPA is fully covered within an approximation called δ. Thus, it uses both, the largest tile mining algorithm as well as subset enumeration algorithm to efficiently generate the optimal set of permission groups.

7.1.4 Min-Noise RMP Algorithm

The Min-noise RMP algorithm (Guo 2010) generates permission groups by fixing the number of groups and then minimizing the approximation δ, used in the Delta-RMP algorithm above. This algorithm also takes the output from the FM algorithm and proceeds to greedily extract permission groups until all the permissions are categorized. The output of this algorithm can be found in Fig. 7d and Table 13d.

7.2 Analysis of the Generated Permission Groups

The algorithms mentioned above, were run on our data set consisting of top 500 free apps from the Google Play store (obtained from APK Pure[1]). While the total number of permissions in Android are more than 500, only 161 of them are ever requested by any of the apps in our data set, and out of these 161, nearly 40 permissions are rarely requested by any app. So, for the purposes of this paper, it is assumed that the maximum number of permissions in Android is 161. It can be seen from Fig. 7a, that 125 permissions are requested by 0 to 50 apps in our data set, which implies that these 125 permissions are the most commonly requested permissions in Android (when considering the apps from our data-set). It also indicates that about 175 apps need between 5 to 10 permissions, which is indicative that a large portion of the apps from our data set do not require more than 10 permissions.

Coverage of permissions, is used to judge the quality of the permission groups generated by the mining algorithms; it is obtained by dividing the total number of unique permissions assigned to any permission group, to the total number of permissions ever requested by any app (which is known to be 161). A permission in a group is considered unique, when other permission groups being collectively considered do not possess that permission. It can be seen from Fig. 7b, that amongst all the algorithms tested, the Delta RMP and the MinNoise RMP algorithm generate permission groups with a good coverage.

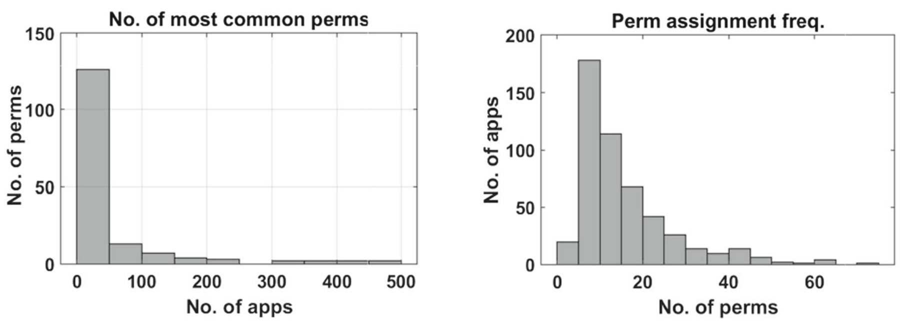
[1] https://apkpure.com/

The Fig. 7c generated from the results for the Delta RMP algorithm, shows the percentage of the number of permission groups generated, permissions covered and the under-privilege of permissions with respect to an increase in delta. Delta is the difference between the UPA matrix and the generated permission groups (a few examples of this are shown in Table 13) and user permission group assignment (not shown, as it is outside the scope of this paper) (Vaidya et al. 2010). Under-privilege of permissions occurs when a lower than requested number of permissions are assigned to apps. It can be observed from this graph that when a delta of 6% is considered, the under-privilege is at 4%, the permissions covered are at 70% however the number of permission groups that need to be considered are 80 (it should be noted that in the graph, the number of groups considered are not a percentage). According to the total number of permissions requested by apps in our data set, which is 161, needing to consider 80 permission groups is a disadvantage.

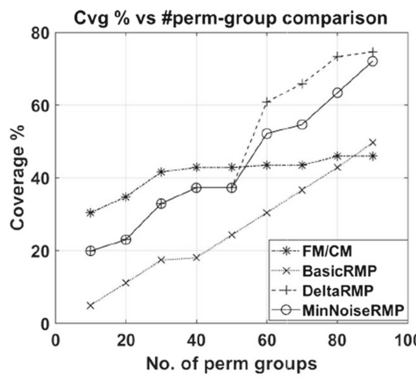
Consider Fig. 7d, which is generated from the results of the MinNoise RMP algorithm, shows the under-privilege and over-privilege percentage of permissions (over-privilege is the assignment of more than requested permissions to apps), when an increasing number of successively mined permission groups are considered. Firstly, this graph shows that even with 20 permission groups mined by this algorithm, the under-privilege percentage is merely 10%; secondly, it shows the sharp rise in the over-privilege percentage above 120 mined permission groups which is noteworthy. Contrasting this with the results of the Delta RMP algorithm (see Fig. 7c), when considering the first 20 mined, the under-privilege is at 21%. Thus, for Android, the MinNoise RMP algorithm is better at generating permission groups than the Delta RMP algorithm.

Finally, the Fig. 7e is obtained by comparing the number of permission assignments to the number of permission group assignments (with the permission groups generated by the MinNoise RMP algorithm). From Fig. 7e, it can be observed that when the coverage is at 20%, the number of permission group assignments drop below the number of permission assignments. This 20% coverage reflects the consideration of about 10 permission groups (from Fig.7b), and a corresponding under-privilege of 20% (7d). This implies that with 10 generated permission groups, the under-privilege of permissions is only about 1 in every 5 permissions requested by the apps, and is considered by us as a positive outcome of the MinNoise RMP mining algorithm. As stated earlier, the remaining permissions required by apps can be obtained by, firstly assigning them to custom-developer-defined permission groups, and then by requesting those groups from the user. A few example

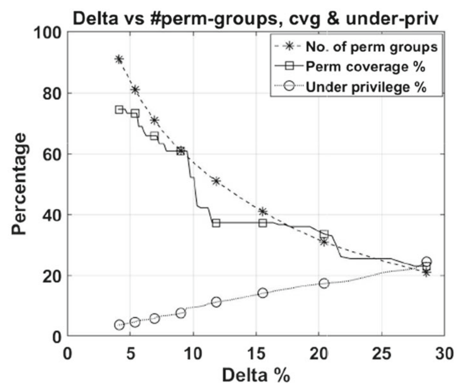
Fig. 7 Results from permission group mining algorithms for Android



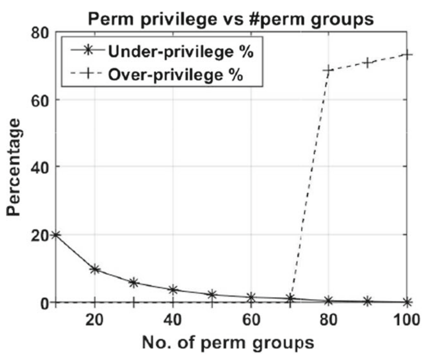
(a) No. of permission assignments reqd. in stock Android



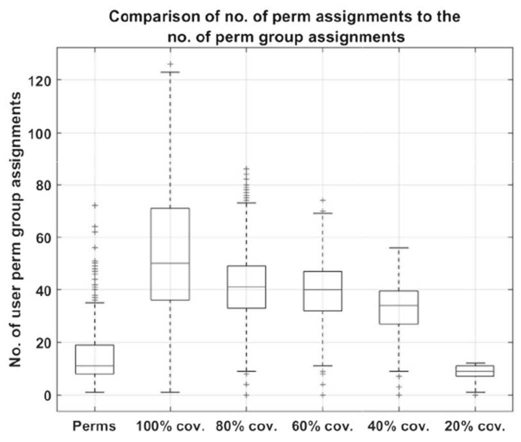
(b) Cvrg. of mining alg.



(c) Delta RMP perm privilege



(d) MinNoise RMP perm privilege



(e) Reduction in no. of assignments

permission groups generated by the mining algorithms are discussed below.

The FM, CM and the Basic RMP algorithms generate permission groups (see Table 13) that contain a lot of common permissions (see Table 13a and b). This increases the number of permission groups required to encompass all the permissions, which dramatically blunts the advantage gained by grouping permissions together. The Delta RMP and the MinNoise RMP algorithms on the other hand generate permission groups with good coverage i.e.: a low number of common permissions inter-group (see Table 13c and d). It can be observed from the tables, that the number of unique permissions for permission groups from each of the five algorithms, FM, CM, Basic RMP, Delta RMP and MinNoise RMP algorithms are 7, 7, 4, 13 and 13 respectively (permission groups for FM and CM are identical for the first few hundred groups). Thus, it is feasible to obtain permission groups for use in Android, by using the MinNoise algorithm (5 permission groups, 13 permissions).

8 Conclusion

In this paper, we have provided a comprehensive formal specification of access control in Android. By formalizing the ACiA, we were able to identify many issues in Android's permission framework. The thorough formalization we have performed in this paper, facilitates a formal security analysis. Some examples of such analysis are,

- What ways can an app receive a system or URI permission it does not possess? Once granted, is there a way in which the permission is revoked?
- What are the requirements for an app to get installed on an Android device?
- How can an app receive URI permission to a content provider?
    - Can this content provider be accessed without the URI permission?
- Can an application access another app's content provider without its URI permission?
- How can the permission's group and protection level be changed?

We have also implemented and analyzed several mining algorithms from the literature, for mining permission groups, and have presented an alternative to Android's permission groups. A few permission groups generated by such algorithms have been shown in this paper as well. We have also shown that the mined permission groups can be used in lieu of Android's own permission groups, towards obtaining a more user-friendly access control mechanism for Android.

Acknowledgements This work is partially supported by DoD ARO Grant W911NF-15-1-0518, NSF CREST Grant HRD-1736209 and NSF CAREER Grant CNS-1553696.

References

(2019) Android perm protection lvl "normal are never re-granted!" https://issuetracker.google.com/issues/129029397, [Online; accessed 21-March-2019].

(2019a) Android Permissions — Android Open Source Project. https://source.android.com/devices/tech/config, [Online; accessed 17-June-2019].

(2019) Issue about Android's permission to permission-group mapping. https://issuetracker.google.com/issues/128888710, [Online; accessed 21-March-2019].

(2019b) Request App Perms — Android Devs. https://developer.android.com/training/permissions/requesting/, [Online; accessed 12-March-2019].

Bagheri, H., Kang, E., Malek, S., Jackson, D. (2015a). In *Intl. Symp. on Formal Methods* (pp. 73–89): Springer.

Bagheri, H., Sadeghi, A., Garcia, J., Malek, S. (2015b). COVERT: Compositional analysis of android Inter-App permission leakage. *IEEE Transactions on Software Engineering*, *41*(9), 866–886.

Bagheri, H., Kang, E., Malek, S., Jackson, D. (2018). A formal approach for detection of security flaws in the android permission system. *Formal Aspects of Computing*, *30*(5), 525–544.

Betarte, G., Campo, J.D., Luna, C., Romano, A. (2015). *Verifying Android's Permission Model*, (pp. 485–504). Cham: Springer.

Betarte, G., Campo, J., Luna, C., Romano, A. (2016). Formal analysis of android's Permission-Based security model 1. *Scientific Annals of Computer Science*, *26*(1), 27–68.

Betarte, G., Campo, J., Cristiá, M., Gorostiaga, F., Luna, C., Sanz, C. (2017). Towards formal model-based analysis and testing of android's security mechanisms. In *2017 XLIII Latin American Computer Conference (CLEI)* (pp. 1–10): IEEE.

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R., Shastry, B. (2012). Towards taming privilege-escalation attacks on android. In *NDSS, Citeseer*, (Vol. 17 p. 19).

Chin, E., Felt, A.P., Greenwood, K., Wagner, D. (2011). Analyzing inter-application communication in android. In *Proc. of the 9th International Conference on Mobile Systems, Applications, and Services* (pp. 239–252).

Davi, L., Dmitrienko, A., Sadeghi, A.R., Winandy, M. (2010). Privilege escalation attacks on android. In *International conference on Information security* (pp. 346–360): Springer.

Enck, W., Ongtang, M., McDaniel, P. (2009a). On lightweight mobile phone application certification. In *Proc. of the 16th ACM Conference on Computer and Communications Security* (pp. 235–245).

Enck, W., Ongtang, M., McDaniel, P. (2009b). Understanding android security. IEEE security & privacy, pp. 50–57.

Enck, W., Octeau, D., McDaniel, P.D., Chaudhuri, S. (2011). A study of android application security. In *USENIX Security Symposium*, (Vol. 2 p. 2).

Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D. (2011). Android permissions demystified. In *Proc. of the 18th ACM conference on Computer and communications security* (pp. 627–638).

Fragkaki, E., Bauer, L., Jia, L., Swasey, D. (2012). *Modeling and Enhancing Android's Permission System*, (pp. 1–18). Berlin: Springer.

Geerts, F., Goethals, B., Mielikäinen, T. (2004). Tiling databases. In *International conference on discovery science* (pp. 278–289): Springer.

Grace, M.C., Zhou, Y., Wang, Z., Jiang, X. (2012). Systematic detection of capability leaks in stock android smartphones. In *NDSS*, (Vol. 14 p. 19).

Guo, Q. (2010). A formal approach to the role mining problem. PhD thesis, Rutgers University-Graduate School-Newark.

Ongtang, M., McLaughlin, S., Enck, W., McDaniel, P. (2012). Semantically rich application-centric security in android. *Security and Communication Networks*, *5*(6), 658–673.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., Glezer, C. (2010). Google android: a comprehensive security assessment. *IEEE Security & Privacy*, *8*(2), 35–44.

Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T. (2010). A formal model to analyze the permission authorization and enforcement in the Android framework. In *Proc. - socialcom 2010: 2nd IEEE international conference on social computing, PASSAT 2010: 2nd IEEE International Conference on Privacy, Security, Risk and Trust* (pp. 944–951).

Talegaon, S., & Krishnan, R. (2019). A formal specification of access control in android. In *International Conference on Secure Knowledge Management in Artificial Intelligence Era* (pp. 101–125): Springer.

Taylor, V.F., & Martinovic, I. (2016). Quantifying permission-creep in the google play store. arXiv:160601708.

Tuncay, G.S., Demetriou, S., Ganju, K., Gunter, C.A. (2018). Resolving the predicament of android custom permissions. In *Proc. 2018 Network and Distributed System Security Symposium*. Reston: Internet Society.

Vaidya, J., Atluri, V., Warner, J. (2006). Roleminer: mining roles using subset enumeration. In *Proceedings of the 13th ACM conference on Computer and communications security* (pp. 144–153).

Vaidya, J., Atluri, V., Guo, Q. (2007). The role mining problem: finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM symposium on Access control models and technologies* (pp. 175–184).

Vaidya, J., Atluri, V., Guo, Q. (2010). The role mining problem: a formal perspective. *ACM Transactions on Information and System Security (TISSEC)*, *13*(3), 1–31.

Wei, X., Gomez, L., Neamtiu, I., Faloutsos, M. (2012). Permission evolution in the android ecosystem. In *Proc. of the 28th Annual Computer Security Applications Conference* (pp. 31–40).

Samir Talegaon received the M.S. degree in Electrical Engineering from The University of Texas at San Antonio (UTSA) in 2014. Currently he is working on a doctoral degree at the Electrical and Computer Engineering Department at UTSA. His research interests include access control in Android and Android platform analysis and modification.

Ram Krishnan is an Associate Professor of Electrical and Computer Engineering at the University of Texas at San Antonio, where he holds Microsoft President's Endowed Professorship. His research focuses on (a) applying machine learning to strengthen cybersecurity of complex systems and (b) developing novel techniques to address security/privacy concerns in machine learning. He actively works on topics such as using deep learning techniques for runtime malware detection in cloud systems and automating identity and access control administration, security and privacy enhanced machine learning and defending against adversarial attacks in deep neural networks. He is a recipient of NSF CAREER award (2016) and the University of Texas System Regents' Outstanding Teaching Award (2015). He received his PhD from George Mason University in 2010.