A Formal Specification of Access Control in Android

Samir Talegaon(✉) and Ram Krishnan

University of Texas at San Antonio, San Antonio, USA
{samir.talegaon,ram.krishnan}@utsa.edu

Abstract. A formal specification of any access control system enables deeper understanding of that system and facilitates performing security analysis. In this paper, we provide a comprehensive formal specification of the Android mobile operating system's access control system, a widely used mobile OS. Prior work is limited in scope, in addition recent developments in Android concerning dynamic runtime permissions require rethinking of its formalization. Our formal specification includes two parts, the User-Initiated Operations (UIOs) and Application-Initiated Operations (AIOs), which are segregated based on the entity that initiates those operation. Formalizing ACiA allowed us to discover many peculiar behaviors in Android's access control system. In addition to that, we discovered two significant issues with permissions in Android which were reported to Google.

Keywords: Android · Permissions · Access control · Formal model

1 Introduction and Motivation

Android is a widely popular mobile OS; Android regulates access to its components and end-user resources with a permission based mechanism. A formal specification for access control in Android (AciA) facilitates a deeper understanding of the nature in which Android regulates app access to resources.Prior work targeting such formalization of the permission mechanism exists, but is limited in its scope since most of it is based on the older install time permission system [7,10,12,13]. Hence, detailed analysis and testing needs to be conducted to build this model, to enable a systematic review for security vulnerabilities.

Users install apps in Android which enable them to fully utilize the device features; and, permission based access control in Android (ACiA) works to regulate app access to sensitive resources. 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. The apps installed by the users require access to these resources for achieving full functionality and they request it from the OS. The OS in turn seeks user interaction to approve some of these requests and grant the necessary permissions to the apps [4,11]; this is illustrated in Fig. 1.

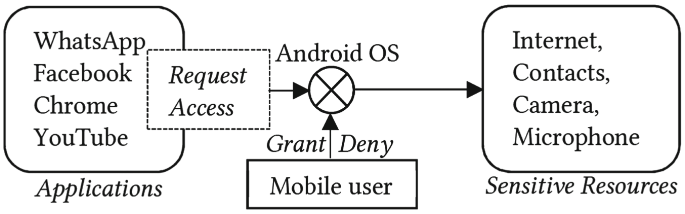

Fig. 1. Permission-based access control in Android

Formalization of ACiA is a non trivial task, and one that has received limited attention, apart from the fact 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 Android obtained from documentation as well as the source code has not been done comprehensively, that includes all the aspects of ACiA such as User Initiated Operations (UIO) and Application Initiated Operations (AIO). Our analysis also enables a holistic and systematic review of ACiA security policies and facilitates the discovery of loopholes in ACiA.

Our Contribution: We present a formal specification of ACiA that enables its analysis from the point of view of security. This model (ACiA$_\alpha$) sheds a light on the internal access control structure of the Android OS with respect to apps, permissions and uri permissions. Without such a model, finding and plugging individual security loopholes in ACiA becomes too complex and may not yield the results that can be obtained via a holistic approach. In order to be more precise and thorough, we have divided our ACiA$_\alpha$ formal specification into two parts, based on the initiating entity for that operation; UIOs and AIOs, initiated by users and apps respectively.

OUTLINE: In Sect. 2, we place our research amongst the current body of works and Sect. 3, describes ACiA formal specification. In Sect. 4, we present anomalies and quirks we discovered in ACiA that were revealed as a result of thorough testing and Sect. 5, describes conclusion and future work with respect to ACiA$_\alpha$. Finally, we end with Sect. 6 containing the references.

2 Related Work

ACiA has received some attention from prior works; a few such closely related works are described in this section. Shin et al. [13] build a model of ACiA and is one of the few works that come close to our work 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 that time. Also, owing to the early nature of this work, it does not model dangerous runtime permissions nor does it include the uri-permissions used to facilitate inter app data sharing. Fragkaki et al. [12] also model ACiA,

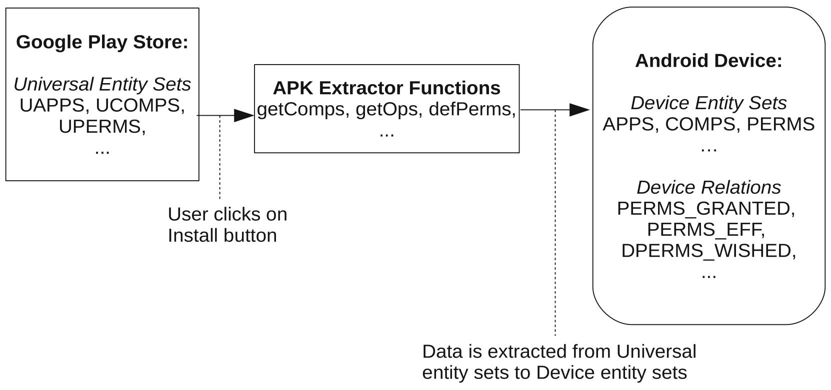

Fig. 2. Building blocks of the ACiA

but their work is centered largely around the uri-permission system. Android's UIOs are not discussed in the work including app installation, un-installation and the vital issue about multiple competing custom permission definitions. Hence, it is required to build a formal model of the ACiA with a holistic perspective, to obtain a deeper understanding of permissions in Android.

Betarte et al. [8–10] present a state-based model of ACiA, which is important owing to the analytical capabilities such a model can offer with respect to security. They define a model state as 8-tuples that record the current state for an Android device, which includes installed apps, permissions, runtime components, temporary and permanently delegated data permissions. They proceed by defining a valid state using 8 distinct conditions, including uniqueness of installed apps, validity of a delegated uri permission which is true is app that receives such a permission, is running, and, uniqueness of all resources on the device. Finally they show three example actions which are the launching of a component, reading of data by a runtime component from a content provider, and, delegation of a temporary uri permission. However, their work does not mention the UIOs along with the fact that apps from the same developer can define the same permissions into distinct permission-groups and protection-levels. Bagheri et al. [5,6] built a formal ACiA model, but, there is no distinction between defined custom permissions and effectively defined custom permissions. They refer to a compete model in the references, however, even in this model, the UIOs do not mention this distinction. In summary, even though the above works formalize both UIO and AIO operations of ACiA, they are limited in detail.

Tuncay et al. [14] identify that developers should always define custom permissions with the same particulars such as permission-group and protection-level; but, the UIOs proposed by them do not differentiate between multiple competing custom permission definitions. Apart from this, uri-permissions are not included in their model, so this model is insufficient to obtain a holistic understanding of ACiA.

To conclude, none of the works that model 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 detailed structure of ACiA, to the level of granularity we deem necessary. This encompasses all the aspects of Android permissions including detailed operations such as app installation/un-installation, permission grants/revocation, inter app component access and delegation of uri permissions which the most important ones. The detailed ACiA$_\alpha$ we built, helped us discover two flaws in Android's permission system which were reported to Google [1,3]. We were also notified by Google that they fixed one of those flaws [1], and the future versions of Android will not have that flaw.

3 Formal Specification of Access Control in Android

ACiA$_\alpha$ was built by reading the developer/source code documentation [2,4], reading the source code itself and verifying our findings via inter-app tests. The ACiA$_\alpha$ model is specified below.

In the normal course of action, the Android user downloads many apps from the Google Play Store. App data such as app names, permissions, app component names are stored at Google and on an Android device. The data stored by Google is mimicked by Universal Sets, whereas, the data stored on an Android device, is mimicked by Device Sets. To install the apps the OS uses many different APIs which we summarize as APK Extractor Functions, and, as shown in the Fig. 2, these functions assist in the installation procedure by extracting the required data from the Universal sets. Upon successful installation, all the necessary device entity sets and relations are updated as shown in Table 4 (InstallApp operation). Similarly, to facilitate app uninstallation, the helper functions enable us to extract data from the device sets and relations for their removal. Many other operations take place during the normal course of working of an app, and, this is portrayed by the UIOs and AIOs that mimic built-in methods such as RequestPermission, GrantPermission, GrantUriPermission etc.

3.1 Building Blocks of ACiA$_\alpha$

ACiA$_\alpha$ operations utilize certain element sets, functions and relations that are listed in Tables 1, 2 and 3. Table 1 shows primary data sets from the Google Play Store (Universal Entity Sets - column 1) and a generic Android device (Device Entity Sets - column 2).

Universal Sets. The Universal Sets are designed to mimic the data structures of the Google Play Store and begin with the letter "U"; they are populated by Google along with app developers and are assumed to be immutable for the purposes of this paper.

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

Table 2. APK extractor functions

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

Table 3. ACiA relations and convenience functions

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

Table 4. Helper functions

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

- UAPPS: the universal set of applications available in the app store (any app store e.g.: Google Play store, Amazon app store).
- UCOMPS: the universal set of components for all the applications from the app store.
- UAUTHORITIES: the universal set of authorities for all the content providers that are defined by all the applications from the app store. An authority is an identifier for data that is defined by a content provider.
- UPERMS: the universal set of permissions consisting of pre-defined system-permissions and application-defined custom permissions from the app store.
- USIG: the universal set of application signatures from the app store.
- UPGROUP: the universal set of permission-groups for pre-defined system permissions as well as application-defined custom permissions for all applications from the app store.
- UPROTLVL: the set of all pre-defined permission protection-levels on an Android device. The protection-level of a permissions corresponds to the significance of the information guarded by it and consists of a base protection-level and additional protection-flags. For the purposes of this paper we only consider the base protection-levels i.e.: **normal**, **dangerous** and **signature**.

Device Sets. The Device Sets are designed to mimic the data structures of a generic Android device and are populated by the device itself in accordance with pre-defined policies from Google.

- APPS: the set of all pre-installed system applications and user-installed custom applications on an Android device; this set includes the stock Android system as well, defined as a single element. So, APPS $\subseteq$ UAPPS for any given Android device (realistically).
- COMPS: the set of all the components belonging to the pre-installed system applications and user-installed custom applications on an Android device. So, COMPS $\subseteq$ UCOMPS on a given Android device.
- AUTHORITIES: the set of all authorities belonging to all applications (pre-installed system applications and user-installed applications) that are installed on an Android device.
- PERMS: the set of all application-defined custom permissions and pre-defined system permissions on an Android device. Note that, PERMS $\subseteq$ UPERMS on a given Android device.
- PGROUP: the set of all application-defined custom permission-groups and pre-defined system permission-groups on an Android device. Note that UPGROUP $\subseteq$ PGROUP on a given Android device.
- PROTLVL: the set of all protection levels present on the device which are the same for any Android device. Note that, PROTLVL $=$ UPROTLVL on a given Android device.
- DATAPERMS: the set of all data-permissions that applications with content providers can grant to other applications, to provide permanent or temporary access to their data. There are two types of data-permissions in Android; base data-permissions and modifier data-permissions. We denote

the base data permissions as $\text{DATAPERMS}_\text{b}$ and the modifier data permissions as $\text{DATAPERMS}_\text{m}$. So, $\text{DATAPERMS}_\text{b} = \{\textbf{dpread}, \textbf{dpwrite}\}$ and $\text{DATAPERMS}_\text{m} = \{\textbf{mpersist}, \textbf{mprefix}, \textbf{none}\}$ and therefore, $\text{DATAPERMS} = \text{DATAPERMS}_\text{b} \times \text{DATAPERMS}_\text{m} = \{(\textbf{dpread}, \textbf{none}), (\textbf{dpwrite}, \textbf{none}), (\textbf{dpread}, \textbf{mpersist}), (\textbf{dpread}, \textbf{mprefix}), (\textbf{dpwrite}, \textbf{mpersist}), (\textbf{dpwrite}, \textbf{mprefix})\}$ for a given Android device.
- URI: the set of all data addresses that applications with content providers can define, which includes certain pre-defined addresses from system applications.
- OP: the set of all operations that may be performed on any Android component. Note that the component types for Android are - Activity, Service, Broadcast Receiver and Content Provider; and, this set is pre-populated by Google. This means that any operations that may be performed on any components installed on a given Android device have to be chosen from this set. Examples of operations that can be performed on components include: *startActivity* on an Activity, *startService* on a Service, *sendBroadcast* on a Broadcast Receiver, and, *create, read, update* and *delete* (CRUD) operations on a Content Provider (this is not an exhaustive list of operations).

APK Extractor Functions. Functions that retrieve information from an application that is about to be installed on the device; evidently, the relations maintained in the device are not useful for these functions. We call these functions APK Extractor Functions. These are shown in Table 2.

- `getComps`, a function that extracts the set of components belonging to an application from the universal set of components.
- `getOps`, a function that extracts the set of allowed operations for a given component, based on the type of that component.
- `getAuthorities`, a partial function that extracts the set of authorities that are defined by an application. An application can define multiple unique authorities and no two authorities from any two applications can be the same.
- `getCompPerm`, a partial function that maps application components and operations they support, to the permissions that other applications are required to posses, to perform these operations. To obtain the set of valid operations on any given component, we use the function `getOps` on that component. Apart from this, a component may not be protected by any permission, and in such a case, the component can be freely accessed by any installed applications (the decision of allowing inter-application component access for any application is made by the developer of that application). If a component is protected by a permission that is not defined on the given Android device, other applications may not perform any operations on such a component (auto deny).
- `appSign`, a function that extracts the signature of an application from the universal set of signatures. This function is used to match application signatures in the pre-requisite condition for granting of signature permissions (i.e.: permissions with the protection level - signature).
- `defPerms`, a function that extracts the custom-permissions that are defined by an application, from the UPERMS. When any application gets installed,

it can define new permissions that are distinct from the pre-installed system permissions and are used to regulate access to its components by other installed applications.

- `defPgroup`, a function that extracts the custom permission groups that are defined by an application, from the UPGROUP. When any application gets installed, it can define new permission-groups that are distinct from the pre-installed permission-groups and are used to mitigate the number of permission prompts shown to the user (a permission prompt is an application asking for certain permission).
- `defProtlvlPerm`, a function that extracts a protection level for a permission as defined by an application. Protection-level is defined for all permissions by some applications, and different applications may define distinct protection-levels for the same permission[1]. Note that, $\forall ua \in$ UAPPS, $\forall up \in$ UPERMS, $\forall pl_1 \neq pl_2 \in$ UPROTLVL. $\texttt{defProtlvlPerm}(ua, up) = pl_1 \Rightarrow \texttt{defProtlvlPerm}(ua, up) \neq pl_2$.
- `defPgroupPerm`, a partial function that extracts the permission-group for some permissions if defined by an application. Permission-group may be defined for some permissions by some applications, and different applications may define distinct permission-groups for the same permission (see footnote 1). Note that, $\forall a \in$ UAPPS, $\forall p \in$ UPERMS, $\forall pg_1 \neq pg_2 \in$ UPGROUP. $\texttt{defPgroupPerm}(ua, up) = pg_1 \Rightarrow \texttt{defPgroupPerm}(ua, up) \neq pg_2$.
- `wishList`, a function that extracts a set of permissions wished by an application, from the UPERMS; this contains all those permissions that the application may ever need in its lifetime.

Device Relations and Convenience Functions. The Device Relations are derived from the Device Sets and portray the information stored by an Android device to facilitate access control decisions. Any relation is always pre-defined for built-in applications and system-permissions, but needs to be updated for user-installed applications and application-defined custom-permissions. Convenience functions query existing relations maintained on the device; evidently, these functions fetch information based on applications that are already installed on the device. These are listed in Table 3.

- APP_COMPS, a one-to-many relation mapping application to it's components. Note that, $\forall a_1 \neq a_2 \in$ APPS, $\forall c \in$ COMPS. $(a_1, c) \in$ APP_COMPS $\Rightarrow (a_2, c) \notin$ APP_COMPS

[1] (Two scenarios) Scenario A: Multiple applications from the same developer define the same permission into distinct permission-levels and/or permission-groups; this is a valid condition, but, only the first application's definition of the permission counts whereas the rest are ignored.
Scenario B: Multiple applications from different developers define the same permission into distinct permission-levels and/or permission-groups; this condition is invalid, since only one developer is allowed to define a new permission at any given time. However, once that application gets uninstalled, other applications from different developers are able to define the same permission!

  - `ownerApp`, a function mapping application component to their owner application. Note that, a component can only belong to a single application. So, $\forall c \in \text{COMPS}.\ (\texttt{ownerApp}(c),\ c) \in \text{APP\_COMPS}$.
  - `appComps`, a function mapping an application to a set of its components. This function is used while an application is being uninstalled, to get the components of the application to be removed from the device. Formally, $\texttt{appComps}(a) = \{c \in \text{COMPS} \mid (a,\ c) \in \text{APP\_COMPS}\}$.

- COMP_PROTECT, a relation that maintains the permissions that are required for operations to be performed on application components. Note that, as it pertains to broadcasts, the sender as well as the receiver may require permissions, however, this relation only maintains the permissions protecting receiving components. To obtain permissions that are required by senders of broadcasts (to be granted to receivers), a helper function `brReceivePerm` defined in the following subsection can be used. So, $\forall c \in \text{COMPS},\ \forall op \in \text{OP},\ \forall p_1 \neq p_2 \in \text{PERMS}.\ (c,\ op,\ p_1) \in \text{COMP\_PROTECT} \Rightarrow \big((c,\ op,\ p_2) \notin \text{COMP\_PROTECT} \wedge p_1 = \texttt{getCompPerm}(c,\ op)\big)$
  - `requiredPerm`, a function that gives the permission that an application component is required to have, to initiate an operation with another component. Note that two components from the same application do not normally need these permissions. So, $\forall c \in \text{COMPS},\ \forall op \in \text{OP}.\ \big(c,\ op,\ \texttt{requiredPerm}(c,\ op)\big) \in \text{COMP\_PROTECT}$
  - `allowedOps`, a function that gives the set of operations that can be performed on a component. Since not all components support all the operations, $\texttt{allowedOps}(c) = \{op \in \text{OP} \mid (c,\ op,\ p) \in \text{COMP\_PROTECT} \wedge p \in \text{PERMS}\}$

- AUTH_OWNER, a one-to-many relation that maps the authorities to their owning applications on a given device. If an application tries to re-define an already defined authority on an Android device, it will not get installed on that device. Note that, $\forall a_1 \neq a_2 \in \text{APPS},\ \forall auth \in \text{AUTHORITIES}.\ (a_1,\ auth) \in \text{AUTH\_OWNER} \Rightarrow \big((a_2,\ auth) \notin \text{AUTH\_OWNER} \wedge auth \in \texttt{getAuthorities}(a)\big)$
  - `authoritiesOf`, a function that give the authorities of a certain application that is installed on an Android device. So, $\texttt{authoritiesOf}(a) = \{auth \in \text{AUTHORITIES} \mid (a,\ auth) \in \text{AUTH\_OWNER}\}$

- PERMS_DEF, a relation mapping user-installed applications, the custom-permissions defined by these applications, the permission-group and the protection-level of such permissions as defined by the respective applications. Note that, $\forall a \in \text{APPS},\ \forall p \in \text{PERMS},\ \forall pg_1 \neq pg_2 \in \text{PGROUP},\ \forall pl_1 \neq pl_2 \in \text{PROTLVL}.\ (a,\ p,\ pg_1,\ pl_1) \in \text{PERMS\_DEF} \Rightarrow \big((a,\ p,\ pg_2,\ pl_1) \notin \text{PERMS\_DEF} \wedge (a,\ p,\ pg_1,\ pl_2) \notin \text{PERMS\_DEF}\big)$
  - `defApps`, a function that returns a set of applications that define a permission. When an application is uninstalled, this function is used to retrieve the set of application that define a certain permissions, thus facilitating the decision of permission removal. So, $\texttt{defApps}(p) = \{a \in \text{APPS} \mid (a,\ p,\ pg,\ pl) \in \text{PERMS\_DEF}\}$

  - `defPerms`, a function that gives the set of permissions that are defined by an installed application. This function is used while an application is uninstalled from a device, to obtain the set of permissions defined by that application so that they may be removed from the device. So, $\texttt{defPerms}(a) = \{p \in \text{PERMS} \mid (a,\ p,\ pg,\ pl) \in \text{PERMS\_DEF}\}$
  - `defPgroup`, a partial function that gives the permission-group for a permission as defined by an installed application. Note that, not all permissions that are defined by applications are categorized into permission-groups. So, $\forall a \in \text{APPS},\ \forall p \in \text{PERMS},\ \forall pg \in \text{PGROUP}.\ \texttt{defPgroup}(\text{a},\ \text{p}) = \text{pg} \Rightarrow \big((\text{a, p, pg, pl}) \in \text{PERMS\_DEF}\ \wedge\ \text{pl} \in \text{PROTLVL}\big)$
  - `defProtlvl`, a function that gives the protection-level for a permission as defined by an installed application. When an application from a certain developer is uninstalled from a device and another application from the same developer is still installed on the device, this function is used to transfer the permission definition to that of the remaining application. So, $\forall a \in \text{APPS},\ \forall p \in \text{PERMS},\ \forall pl \in \text{PROTLVL}.\ \texttt{defProtlvl}(\text{a, p}) = \text{pl} \Rightarrow \big((\text{a, p, pg, pl}) \in \text{PERMS\_DEF}\ \wedge\ \text{pg} \in \text{PGROUP}\big)$

– PERMS_EFF, a relation mapping all the pre-installed system applications and user-installed custom applications on a device, the permissions defined by them, the permission-groups and protection-levels of such permissions. In case of multiple apps attempting to re-defined a permission on a device, Android follows a first come first serve policy, thus only accepting the definition of the first app that is installed. This relation reflects the effective definition of permissions, as defined by system, system applications or user-installed applications. Note that, $\forall a \in \text{APPS},\ \forall p_1 \neq p_2 \in \text{PERMS},\ \forall pg_1 \neq pg_2 \in \text{PGROUP},\ \forall pl_1 \neq pl_2 \in \text{PROTLVL}.\ (a,\ p_1,\ pg_1,\ pl_1) \in \text{PERMS\_EFF} \Rightarrow (a,\ p_2,\ pg_1,\ pl_1) \notin \text{PERMS\_EFF}\ \wedge\ (a,\ p_1,\ pg_2,\ pl_1) \notin \text{PERMS\_EFF}\ \wedge\ (a,\ p_1,\ pg_1,\ pl_2) \notin \text{PERMS\_EFF}$

  - `effApp`, a function that gives the pre-installed system application, the Android OS, or the user-installed application that defined a permission. This function is used during the signature matching process required to be completed before any application is installed. So, $\forall p \in \text{PERMS}.\ (\texttt{effApp}(p),\ p) \in \text{PERMS\_EFF}$
  - `effPerms`, a function that gives the set of permissions as effectively defined by the system, a system application or a user-installed custom application. This function is used while an application is uninstalled from a device, to obtain the set of permissions defined by that application so that they may be removed from the device. So, $\texttt{effPerms}(\text{a}) = \{\text{p} \in \text{PERMS} \mid (\text{a, p, pg, pl}) \in \text{PERMS\_EFF}\ \wedge\ \text{pg} \in \text{PGROUP}\ \wedge\ \text{pl} \in \text{PROTLVL}\}$
  - `effPgroup`, a function that maps a permission to its permission-group. This function is used when making access control decisions to auto grant certain requested dangerous permissions. Note that, $\forall p \in \text{PERMS}.\ \texttt{effPgroup}(\text{p}) = \text{pg} \Rightarrow \big((\text{a, p, pg, pl}) \in \text{PERMS\_EFF}\ \wedge\ \text{pg} \in \text{PGROUP}\ \wedge\ \text{pl} \in \text{PROTLVL}\big)$

  - `effProtlvl`, a function mapping a permission to its protection level on an Android device. This function is used to obtain permission protection-levels used during permission granting process. Note that, $\forall p \in$ PERMS. $\texttt{effProtlvl}(\mathrm{p}) = \mathrm{pl} \Rightarrow \big(\mathrm{a, p, pg, pl}) \in \mathrm{PERMS\_EFF} \ \wedge \ \mathrm{pg} \in \mathrm{PGROUP} \ \wedge \ \mathrm{pl} \in \mathrm{PROTLVL}\big)$

- DPERMS_WISHED, a many-to-many relation mapping applications to the dangerous permissions requested by them in the manifest. Since normal and signature permission grants happen at install time, only dangerous permissions are a part of this relation. Note that, $\forall a \in$ APPS, $\forall p \in$ PERMS. $(a,\ p) \in \mathrm{DPERMS\_WISHED} \Rightarrow p \in \texttt{wishList}(a) \wedge \texttt{effProtlvl}(p) =$ dangerous

  - `wishDperms`, the mapping of an application to a set of dangerous permissions requested by it in the manifest. Formally, $\texttt{wishDperms}(a) = \{p \in \mathrm{PERMS} \ \mid (a,\ p) \in \mathrm{DPERMS\_WISHED}\}$.

- PERMS_GRANTED, a many-to-many relation mapping applications to the permissions granted to them. Note that, $\forall a \in \mathrm{APPS}, \forall p \in \mathrm{PERMS}.\ (a, p) \in \mathrm{PERMS\_GRANTED} \Rightarrow p \in \texttt{wishList}(a)$

  - `grantedPerms`, the mapping of an application to the a set of permissions granted to it. Formally, $\texttt{grantedPerms}(a) = \{\ p \in \mathrm{PERMS} \mid (a,\ p) \in \mathrm{PERMS\_GRANTED}\}$.

- GRANTED_DATAPERMS, a relation mapping applications to the data permissions granted to them. Data permissions are granted to applications by the applications that own that data permission.

  - `grantNature`, a function that gives the nature of a data permission grant to an application. Such a nature can be Permanent, Temporary and Not Granted (when the data permission was not granted to that application); a permanent permission grant survives device restarts whereas a temporary permission grant is revoked once the application is shut down. So, $\forall a \in$ APPS, $\forall uri \in$ URI, $\forall dp_b \in \mathrm{DATAPERMS_b}$, $\forall dp_m \in \mathrm{DATAPERMS_m}$, $\forall dp \in$ DATAPERMS.
    $\texttt{grantNature}(a,\ uri,\ dp) = \textbf{SemiPermanent} \Rightarrow \big(dp_m = \textbf{mpersist} \ \wedge \ (a,\ uri,\ dp) \in \mathrm{GRANTED\_DATAPERMS}\big) \ \vee$
    $\texttt{grantNature}(a,\ uri,\ dp) = \textbf{Temporary} \Rightarrow \big(dp_m = \emptyset \ \wedge \ (a,\ uri,\ dp) \in \mathrm{GRANTED\_DATAPERMS}\big) \ \vee$
    $\texttt{grantNature}(a,\ uri,\ dp) = \textbf{NotGranted} \Rightarrow (a,\ uri,\ dp) \notin \mathrm{GRANTED\_DATAPERMS}$
  - `uriPrefixCheck`, a function that checks the data-permission for an application against a prefix match given by the data-permission modifier **mprefix**. Since data-permissions can be granted on a broad scale, this modifier makes it possible for the application to receive access to all the sub-URIs that begin with the specific URI that has been granted. For example, if any data-permission is granted consisting of the **mpersist** modifier for a URI to an application such as content://abc.xyz/foo,

then, that application receives access to all the URIs that are contained in the granted URI such as content://abc.xyz/foo/bar or content://abc.xyz/foo/bar/1 and so on. So, $\forall a \in$ APPS, $\forall uri \in$ URI, $\forall dp_b \in$ DATAPERMS$_\mathrm{b}$, $\forall dp_m \in$ DATAPERMS$_\mathrm{m}$, $\forall dp = (dp_b,\ dp_m) \in$ DATAPERMS. `uriPrefixCheck`(a, uri, dp) $= \mathbb{T} \Rightarrow \big(\mathrm{dp_m} = \mathbf{mprefix} \wedge$ `prefixMatch`(a, uri, dp)$\big)$

Helper Functions. The Helper functions facilitate access control decisions by extracting data from the Android device and abstracting away complicated details for the Android device without compromising details about the Android permission model. These are listed in Table 4.

- `userApproval`, a function that gives the user's choice on whether to grant a permission for an application.
- `brReceivePerm`, a function that gives a permission that is required to be possesed by an application component in order to receive broadcasts from this component. Note that broadcast receivers from the same application do not need this permission.
- `corrDataPerm`, a function that obtains the correlated data address and data permission for a system level permission.
- `belongingAuthority`, a function that obtains the authority to which the given URI belongs. At any given time a URI can belong to only a single authority.
- `requestApproval`, an application-choice function that provides the data-permissions for the URIs that are requesting by one application and granted by the other application; only if conditions mentioned below are met, otherwise it returns a null set. Note that, $\forall a_2 \neq a_1 \in$ APPS, $\forall uri \in$ URI, $\forall dp \in$ DATAPERMS. $\texttt{requestApproval}(a_2,\ a_1,\ uri) \neq \emptyset \Rightarrow \bigwedge_{dp\ \in\ \texttt{requestApproval}(a_1,\ a_1,\ uri)} \texttt{appAuthorized}(a_2,\ uri,\ dp)$
- `grantApproval`, an application-choice boolean function that provides the data-permissions for the URIs that are chosen to be delegated by one application to another ; only if conditions mentioned below are met, otherwise it returns a null set. Note that, $\forall a_1 \neq a_2 \in$ APPS, $\forall uri \in$ URI, $\forall dp \in$ DATA PERMS. $\texttt{grantApproval}(a_1,\ a_2,\ uri) = \mathbf{T} \Rightarrow \bigwedge_{dp\ \in\ \texttt{grantApproval}(\mathrm{a_1},\ \mathrm{a_2},\ \mathrm{uri})} \texttt{appAuthorized}(a_2,\ uri,\ dp)$
- `prefixMatch`, a boolean function that matches an application, a uri and a data-permission to one of the **mprefix** data-permissions using the relation GRANTED_DATAPERMS.
- `appAuthorized`, a boolean function to check if an application has a certain data-permission with respect to the provided URI. So, $\forall a \in$ APPS, $\forall uri \in$ URI, $\forall dp \in$ DATAPERMS. `appAuthorized`(a, uri, dp) $\Rightarrow$ $(a,\ uri,\ dp) \in$ GRANTED_DATAPERMS $\vee$ $\texttt{ownerOf}\big(\texttt{belongingAuthority}(uri)\big) = a \vee \exists p \in \texttt{grantedPerms}(a).\ (uri,\ dp) \in \texttt{corrDataPerm}(p) \vee$ `uriPrefixCheck`(a, uri, dp)

Understanding the ACiA$_\alpha$ operations

- Updates on Administrative operations are assumed to be in order, this means that they need to be executed in the order in which they are listed.
- The universal and on device sets are the building blocks of the relations, however, in this model, the sets and the relations need to be updated individually. This means that when a relation is constructed from two sets (for example), updating the sets will not impact the relation in any way.

3.2 User Initiated Operations

ACiA$_\alpha$ - UIOs are initiated by the user or require their approval before they can be executed. Note that certain special apps that are signed with Google's or the platform signature are exempt from this requirement, since they have access to a broader range of "system only" permissions that may enable them to perform these operations without user intervention. Also to be noted that only the most important updates are discussed in this section, the detailed updates are available on Table 5. For each operation, the updates are assumed to be executed in-order.

- **AddApp:** This 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. It is required, for app installation to proceed, that any custom permission definitions either be unique or in case of multiple such definitions, that they are all defined by apps signed with the same certificate.
- **DeleteApp:** This operation resembles a user un-installing an app from the Settings application. For this operation to proceed there are no conditions that need to be satisfied.
- **GrantDangerPerm/GrantDangerPgroup:** These operations resemble the user granting a dangerous permission/permission-group to an app via the Settings app; and, the execution of this operation result in an app receiving a dangerous permission/permission-group respectively. It is required for the app to have requested atleast 1 such dangerous permission from the same permission-group in the manifest.
- **RevokeDangerPerm/RevokeDangerPgroup:** These operations resemble the user revoking a dangerous permission/permission-group from an app via the Settings app; and, their execution results in an app's dangerous permission/permission-group getting revoked. It is required that the application be granted to said permission/permission-group prior to execution of these operations.

3.3 Application Initiated Operations

The AIOs are initiated by the apps when attempting to perform several tasks such as requesting a permission from the user, granting a uri permission to another app, revoking a uri permission from all apps etc. With the exception of the **RequestPerm** operation, these operations do not require user interaction and can be completed by the Android OS.

– **RequestPerm:** This operation resembles an app requesting a dangerous system permission from the Android OS. Such a permission request is successful only if the user grants it to the app, or, the app requesting it already has

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:
$\text{APPS}' = \text{APPS} \cup \{ua\}$; $\text{COMPS}' = \text{COMPS} \cup \texttt{getComps}(ua)$; $\text{APP\_COMPS}' = \text{APP\_COMPS} \cup \{ua\} \times \texttt{getComps}(ua)$
$\text{AUTHORITIES}' = \text{AUTHORITIES} \cup \texttt{getAuthorities}(ua)$;
$\text{AUTH\_OWNER}' = \text{AUTH\_OWNER} \cup \{ua\} \times \texttt{getAuthorities}(ua)$
$\text{PERMS\_DEF}' = \text{PERMS\_DEF} \cup \bigcup_{up \in \texttt{defPerms}(ua)} \Big\{\big(ua,\ up,\ \texttt{defPgroupPerm}(ua,\ up), \texttt{defProtlvlPerm}(ua,\ up)\big)\Big\}$
$\text{PERMS\_EFF}' = \text{PERMS\_EFF} \cup \bigcup_{up \in \texttt{defPerms}(ua) \setminus \text{PERMS}} \Big\{\big(ua,\ up,\ \texttt{defPgroupPerm}(ua,\ up), \texttt{defProtlvlPerm}(ua,\ up)\big)\Big\}$
$\text{PERMS}' = \text{PERMS} \cup \texttt{defPerms}(ua)$
$\text{COMP\_PROTECT}' = \text{COMP\_PROTECT} \cup \bigcup_{c \in \texttt{appComps}(a);\ op \in \texttt{getOps}(c);\ p \in \text{PERMS} \cap \texttt{getCompPerm}(op,\ c)} \{(c,\ op,\ p)\}$
$\text{PGROUP}' = \text{PGROUP} \cup \texttt{defPgroup}(ua)$
$\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \cup \bigcup_{\substack{a' \in \text{APPS} \\ up \in \texttt{wishList}(a') \cap \text{PERMS such that} \\ \texttt{effProtlvl}(up) = \textbf{normal}}} \{(a',\ up)\} \cup \bigcup_{\substack{a' \in \text{APPS};\ up \in \texttt{wishList}(a') \cap \text{PERMS such that} \\ (\texttt{effProtlvl}(up) = \textbf{signature} \wedge \\ \texttt{appSign}(\texttt{effApp}(up)) = \texttt{appSign}(a'))}} \{(a',\ up)\}$
$\text{DPERMS\_WISHED}' = \text{DPERMS\_WISHED} \cup \bigcup_{\substack{a' \in \text{APPS};\ up \in \texttt{wishList}(a') \text{ such that} \\ \texttt{effProtlvl}(up) = \textbf{dangerous}}} \{(a',\ up)\}$
Operation: **DeleteApp**$(a : \text{APPS})$
Authorization Requirement: **T**
Updates:
$\text{COMP\_PROTECT}' = \text{COMP\_PROTECT} \setminus \bigcup_{\substack{c \in \texttt{appComps}(a) \\ op \in \texttt{allowedOps}(c) \\ p \in \text{PERMS} \cap \texttt{getCompPerm}(op,\ c)}} \{(c,\ op,\ p)\} \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\};\ c \in \texttt{appComps}(a') \\ op \in \texttt{allowedOps}(c) \\ p \in \texttt{effPerms}(a) \cap \texttt{requiredPerm}(c,\ op)}} \{(c,\ op,\ p)\}$
$\text{AUTH\_OWNER}' = \text{AUTH\_OWNER} \setminus \{a\} \times \texttt{authoritiesOf}(a)$; $\text{AUTHORITIES}' = \text{AUTHORITIES} \setminus \texttt{authoritiesOf}(a)$
$\text{COMPS}' = \text{COMPS} \setminus \texttt{appComps}(a)$; $\text{APP\_COMPS}' = \text{APP\_COMPS} \setminus \{a\} \times \texttt{appComps}(a)$
$\text{PERMS}' = \text{PERMS} \setminus \Big(\texttt{effPerms}(a) \setminus \bigcup_{a' \in \text{APPS} \setminus \{a\}} \texttt{defPerms}(a')\Big)$
$\text{PGROUP}' = \text{PGROUP} \setminus \Big(\texttt{defPgroup}(a) \setminus \bigcup_{a' \in \text{APPS} \setminus \{a\}} \texttt{defPgroup}(a')\Big)$
$\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \setminus \Big(\{a\} \times \texttt{grantedPerms}(a) \cup \bigcup_{a' \in \text{APPS} \setminus \{a\};\ p \in \texttt{effPerms}(a)} \{(a', p)\}\Big)$
$\text{DPERMS\_WISHED}' = \text{DPERMS\_WISHED} \setminus \Big(\{a\} \times \texttt{wishDperms}(a) \cup \bigcup_{a' \in \text{APPS} \setminus \{a\};\ p \in \texttt{effPerms}(a)} \{(a',\ p)\}\Big)$
$\text{PERMS\_EFF}' = \Big(\text{PERMS\_EFF} \setminus \bigcup_{p \in \texttt{effPerms}(a)} \Big\{\big(a,\ p,\ \texttt{effPgroup}(p), \texttt{effProtlvl}(p)\big)\Big\}\Big) \cup \bigcup_{p \in \texttt{effPerms}(a')} \Big\{\big(a',\ p,\ \texttt{defPgroup}(a',\ p), \texttt{defProtlvl}(a',\ p)\big)\Big\}$, where $a' \in \texttt{defApps}(p) \setminus \{a\}$

(*continued*)

Table 5. (*continued*)

$\text{PERMS\_DEF}' = \text{PERMS\_DEF} \setminus \bigcup_{p \in \texttt{defPerms}(a)} \{(a,\ p,\ \texttt{defPgroup}(a,\ p),\ \texttt{defProtlvl}(a,\ p))\}$
$\text{COMP\_PROTECT}' = \text{COMP\_PROTECT} \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ c \in \texttt{appComps}(a');\ op \in \texttt{allowedOps}(c) \\ p \in \text{PERMS} \cap \texttt{getCompPerm}(op,\ c)}} \{(c,\ op,\ p)\}$
$\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \cup$ $\bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ p \in \texttt{wishList}(a') \cap \text{PERMS such that} \\ \texttt{effProtlvl}(p) = \textbf{normal}}} \{(a',\ p)\} \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ p \in \texttt{wishList}(a') \cap \text{PERMS such that} \\ (\texttt{effProtlvl}(p) = \textbf{signature} \ \wedge\ \texttt{appSign}(\texttt{effApp}(p)) = \texttt{appSign}(a'))}} \{(a',\ p)\}$
$\text{DPERMS\_WISHED}' = \text{DPERMS\_WISHED} \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\};\ p \in \texttt{wishList}(a') \text{ such that} \\ \texttt{effProtlvl}(p) = \textbf{dangerous}}} \{(a',\ p)\}$
$\text{APPS}' = \text{APPS} \setminus \{a\}$
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: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \cup \bigcup_{p \in \texttt{wishDperms}(a) \text{ such that } \texttt{effPgroup}(p) = pg} \{(a, p)\}$
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: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \setminus \bigcup_{\substack{p \in \texttt{grantedPerms}(a) \text{ such that} \\ (\texttt{effPgroup}(p) = pg \ \wedge\ p \in \texttt{wishDperms}(a))}} \{(a,\ p)\}$

another permission from the same permission group. If successful, the app is granted the requested dangerous permission.

- **RequestDataPerm:** This operation denotes the uri-permission requests by apps. Such a request may be granted by apps only if they have the required access. Once this request is successful, the app requesting it is granted the uri-permission.
- **GrantDataPerm:** This operation resembles the uri-permission delegation by apps; and, it only succeeds if the app trying to grant the permissions has access to do so.
- **RevokeDataPerm:** This operation resembles the revocation of uri-permission from an installed app. Applications can revoke uri-permissions from other apps only if they have been granted such a permission via the manifest, or, is the owner app for that uri.
- **RevokeGlobalDataPerm:** This operation is similar to the **RevokeDataPerm** except that it revokes the uri-permissions from all applications on the device. Applications that receive access to the content provider may only invoke this function successfully.
- **CheckDataAccess:** This operation checks if a particular app has access to a uri. Uri permissions are delegated to apps by other app possessing those permissions.

Table 6. ACiA$_\alpha$ Application Initiated Operations

Operation: **RequestPerm**$(a : \text{APPS},\ p : \text{PERMS})$ Authorization Requirement: $(a,p) \in \text{DPERMS\_WISHED} \ \wedge$ $\Big(\big(\exists p' \in \text{PERMS} \setminus \{p\}.\ \texttt{effPgroup}(p') = \texttt{effPgroup}(p) \wedge (a,p') \in \text{PERMS\_GRANTED}\big) \ \vee \ \texttt{userApproval}(a,p)\Big)$ Updates: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \ \cup \ \{(a,p)\}$
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**$\Big(a_{src} : \text{APPS},\ a_{tgt} : \text{APPS},\ uri : \text{URI},\ dp : 2^{\text{DATAPERMS}}\Big)$ 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 $\psi \ : \ \equiv \quad a_{src} = \texttt{ownerOf}\big(\texttt{belongingAuthority}(uri)\big) \ \vee \ \exists p \in \texttt{grantedPerms}(a_{src}).\ (uri,\ dp) \in \texttt{corrDataPerm}(p)$ 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: -
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: $\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \ \setminus \bigcup_{\substack{(a,\ uri,\ dp) \ \in \ \text{GRANTED\_DATAPERMS such that} \\ \texttt{grantNature}(a,\ uri,\ dp) \ = \ \textbf{Temporary}}} \{(a,\ uri,\ dp)\}$

- **CheckAccess:** This operation resembles a component attempting to do an operation on another component; components may belong to the same or distinct apps. This operation can succeed if the app attempting to perform it has been granted the required permissions.
- **AppShutdown:** This operation resembles an app shutting down, so all the temporary uri-permissions granted to it are revoked unless they are persisted.

4 Experimental Setup and Observations

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 this section, apart from helping us understand the intricate details of operations such as application installation, uninstallation, permission grants and revocation etc. A brief overview on the testing methodology is explained in the section below.

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 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 based on the model, and then verified them using these tests, and it is this very methodical procedure that enabled us to discover flaws in the design of ACiA that were communicated to Google via its issue-tracker.

4.1 Experimental Setup

A simple three app base 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 permission 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 for multiple apps and uninstallation sequence for multiple apps.

For brevity, we demonstrate a few simple tests that we conducted to verify our findings using test apps as follows.

1. **Verifying authorization requirements for 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) If the app being installed defines a new permission, it is required that such a permission be unique and is not already defined on the device.
(b) In the case it is already defined, the app must come from the same developer as the one that defined the permission; this means their signatures must match.
(c) If the app being installed defines a new authority for a content provider, this authority must be unique.

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

Verification Methodology. For this test, we designed three test apps that each define the same permission, 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 permission (see Fig. 3): Apps 1 and 2 could be installed even though they re-defined the same permission, however, App3's installation could not proceed since it was signed with a certificate from a different developer.
Case for defining new authority (see Fig. 4): Apps 1 and 3 could be installed since they defined a unique authority, however, App 2's installation could not proceed since it attempted to re-define an authority that already existed on the device.

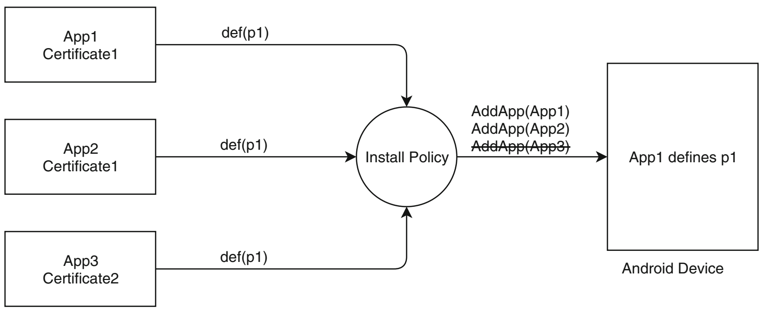

Fig. 3. App installation authorization requirement - new permission definition

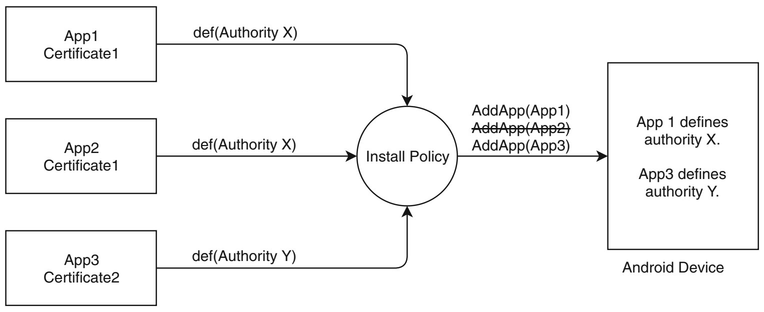

Fig. 4. App installation authorization requirement - unique authority

2. **Check whether permission definitions were changed in accordance to the apps that were present on a device.** The 3 above mentioned apps were designed to define a single permission, but into 3 distinct permission groups i.e.: pgroup1, pgroup2 and pgroup3. Upon installation of app1, the permission p1 was defined on the device into the permission-group "pgroup1"; following this, apps 2 and 3 were installed with no change in p1's permission-group (expected result). However, after app1 was uninstalled using the GUI uninstallation method, the permission definition of p1 changed randomly to "pgroup2" or "pgroup3"; this behavior was replicated using a combination of distinct sequences for TP3 and TP4 and each test yielded the same result. This meant that Android was randomly assigning permission definitions to apps, when the initial app defining such a permission was uninstalled (this means that a random app's definition of the permission would be enforced upon app1's uninstallation; once enforced, such a definition stays until that app gets uninstalled and so on). It is to be noted that this issue occurs during the GUI uninstallation method, and was not observed when apps were removed using the command line (something which only developers use anyway). This makes the issue more relevant, since users normally uninstall apps using the GUI and not the command line tools.

4.2 Observations from ACiA Acquired via Testing the ACiA$_\alpha$ Model

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 [1,3], and, Google has said that [1] has been fixed and will be available in the future version of Android. We also present our rationales for these anomalies, wherever necessary. The model building phase for ACiA$_\alpha$ is quite complex due to the lengthy nature of Android's source code. Every important observation was verified using test-apps, and the final model is designed to capture all the important aspects

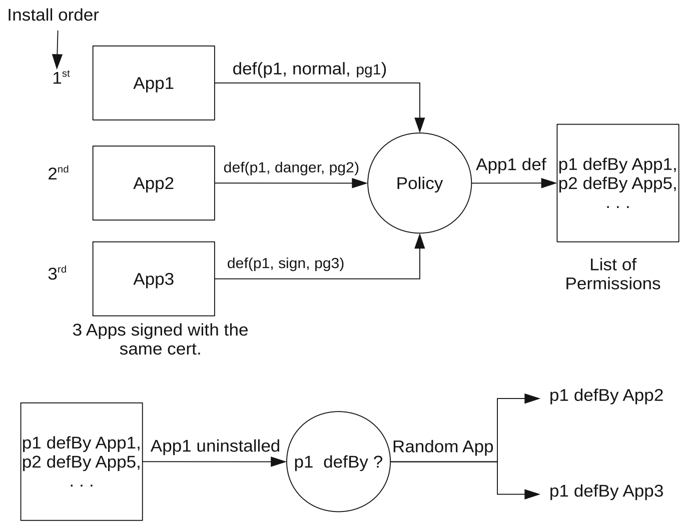

Fig. 5. Anomaly in Android custom permissions

of ACiA. Below we note a few such important observations and the related operations where they were encountered.

1. **Undefined behavior in case of competing custom permission definitions.** Android allows multiple definitions of the same permission (from apps signed with the same certificate) to co-exist on a device. The effective definition for such a permission is taken from the first app that defines it; any subsequent definitions of the same permission are ignored by Android. This can cause issues when that app that defined the permission is un-installed, since there is no order with which Android changes the definition of the permission, hence, the permission definition randomly jumps from the un-installed app, to any other app that defined the permission.

   Explanation Using ACiA$_\alpha$ Model: This issue was encountered while testing ACiA, based on ACiA$_\alpha$ model and can be demonstrated via the Authorization Requirement of the **AddApp** operation (see Table 5), the PERMS_EFF updates in the **AddApp** and **DeleteApp** operations. While an app (App1) is being installed (see Fig. 5), Android checks to see if the custom permission defined by the app (p1) does not already exist on the device; when this check passes, the app gets installed (assume it passes). Then the relation PERMS_EFF gets updated to indicate App1 effectively defined the permission p1. Upon installing two additional apps (App2 and App3) that also define the same permission and are signed with the same certificate as App1, Android will ignore their definition of the permission p1; this is in line with how Android should work. Upon un-installation of App1, however, we can see that, in the operation **DeleteApp**, the relation PERMS_EFF gets modified after choosing a random permission from the set of permissions defined by any other app - in this case either App2 or App3.

Rationale: This random jump between permission definitions upon app uninstallation is an unwanted behavior; and, may occur despite the fact that developers are expected to stick to the same definitions for any custom-permissions they define, since this is not enforced by Google.

Proposed Resolution: We believe that Android should remember the order of app installations and modify permission definitions in-order rather than take a random approach to the same; alternatively, keeping in line with highest protection level first, the permission definition that puts the permission into the higher protection level should be utilized by Android. This will enable developers to definitively know, which definition of a custom permission is active.

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 permission, app un-installation results in any new normal permissions to be not granted to apps. This is not the case with signature permissions, as they are automatically re-granted by Android.

   Explanation Using ACiA$_\alpha$ Model: Consider two apps App1 and App2 that define a permission p1, where App1 defines this permission to be in the **normal** protection-level whereas App2 defines the same permission in the **dangerous** protection-level; since App1 got installed first, according to PERMS_EFF from the operation **AddApp** (see Table 5), its definition is effective i.e.: protection-level of p1 is **normal**. If, at this step, App1 is uninstalled, App2's definition of the permission becomes active, this functions properly according to the model. However, App2 is not granted this permission nor do any other apps that may have requested this permission in their manifests prior to App1's un-install. This is not true for **signature** permissions that are granted upon signature match, nor does it apply to **dangerous** permissions that are requested at run-time by apps. To top it all, in the event that a developer defines a custom-permission without specifying any protection-level to it, the default protection-level applies that is **normal**, this further exacerbates the issue mentioned above and is particularly difficult for new developers. We have reported this issue to Google [1].

   Rationale: We believe that this is an unwanted behavior, and the reason is that if a **signature** permissions are being granted in the above mentioned scenario, **normal** permissions should be granted as well since both these permissions are listed as install-time-granted permissions.

   Proposed Resolution: We believe that Android should re-grant such converted **normal** permissions in the same way it re-grants the **signature** permissions, so that, the behavior of permissions can be correctly predicted by developers.

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. Apart from this, apps can protect the entire content providers with a single (single permission for read and write) or double (one permission for read and one for write) permissions. When an app receives a temporary uri permission, it can even grant those permissions to any other apps temporarily or semi-permanently. This is clearly a flaw as no app can control this style of chain uri permission grants; this flaw is not exactly new and was discovered a few years ago [12].

   Explanation Using $ACiA_\alpha$ Model: We can see from Table 6, the **GrantDataPerm** operation does not keep a record of the type of uri-permission grant (temporary or permanent). The authorization requirement for this operation is a simple boolean helper function from Table 4 - `grantApproval`. This is in line with how Android works, and, once the app shuts down, as can be seen from the operation **AppShutdown**, merely the temporarily granted permissions are revoked. The relation GRANTED_DATAPERMS (from Table 3) is responsible for keeping track of the types of uri-permission grants.
4. **Custom permission names are not enforced using the reverse domain style.** Although Google recommends developers to use the reverse domain style naming convention for defining custom-permissions, no formal regulation is done by Google. This can lead to unwanted behavior for the end-user when a new app fails to install, as it attempts to re-define a permission that already exists on the device (if this new app is from a different developer), confusing the user.

   Rationale: Google's attempt at providing developers free reign over custom-permissions may backfire and cause an unaware user to be unable to install required apps. This issue should be rectified by Google by regulating custom-permission names (Fig. 6).
5. **Complex custom permission 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 permission definitions, since only if there are no other apps defining the same permission, is that permission removed from the system. For this test case we constructed 3 test apps and performed worst case testing with respect to permission definitions and found Issue #2 described above. This is a grave issue since the documentation states that all normal permissions are always granted when their apps are installed on the device.

   Explanation Using $ACiA_\alpha$ Model: From Table 5, we can see that the **DeleteApp** operation that models the app un-installation procedure is quite complex. Android allows apps to define custom permissions, so, after the un-installation of such apps, Android removes any custom permissions effectively defined by that app. However, in the event that another app with the same certificate defines the permissions to be removed, Android simply switches

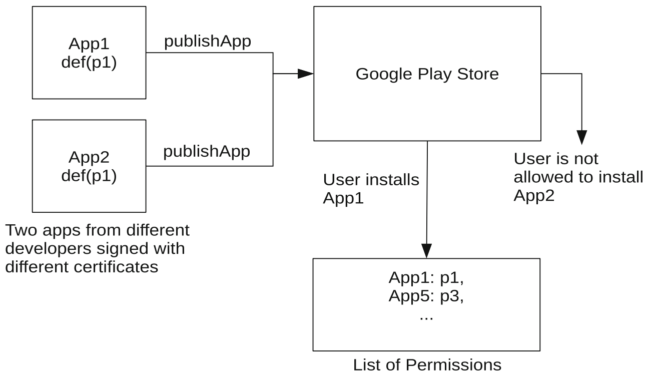

Fig. 6. Drawback of not enforcing custom permission names

the permission definition and keeps that permission from getting mistakenly deleted. Any and all updates to the permissions protecting app components, granted permissions or dangerous permissions requested by apps need to be postponed until after all such custom permission definitions effectively defined by the app being removed, have been dealt with; otherwise these permission definitions would be inconsistent with their expected behavior.

Rationale: This is because Android performs a wide array of book-keeping operations upon the un-installation of any apps, this is done to maintain consistency across the defined custom permissions and effective custom permissions that exist on the device, the permissions that are granted to apps and the dangerous permissions requested by apps to name a few.

5 Conclusion

We have built a model for ACiA and present it in brief, in this paper. Quite a few peculiar behaviors come to light as we delve deeper into Android, some of which we were able to discover and present in this work as well. Future work includes formal analysis to answer many interesting questions such as.

- Given an app and a system permission it does not posses, is there a way in which that app receives that permission? Which ways?
- Given an app and a system permission it does posses, is there a way in which that permission is revoked from this app? Which ways?
- Given an app and it's parameters, can this app be installed on an Android device? Which ways?
- Given an app and a content provider path, is there a way this app can receive a uri permission to that path? If yes, how many ways exist for the app to receive such a permission?

  - Can the app access that data stored by that content provider without obtaining any uri-permission to that path? If yes, which are those?
- Given two apps, X and Y, where Y protects its content provider with a uri permission, is there a way for app X to access Y's content provider without obtaining the said uri permission? If yes, which are those?
- Given a permission defined into a permission-group, is there a way its permission-group can be changed on the device? If yes, how many ways can this be achieved? Similarly, is there a way to change its protection level? If yes, how which are those?
- Given an app and a component, can the component be associated with more than one app?

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

1. Android permission protection level "normal" are never re-granted! (2019). https://issuetracker.google.com/issues/129029397. Accessed 21 Mar 2019
2. Android Permissions—Android Open Source Project (2019). https://source.android.com/devices/tech/config. Accessed 17 June 2019
3. Issue about Android's permission to permission-group mapping (2019). https://issuetracker.google.com/issues/128888710. Accessed 21 Mar 2019
4. Request App Permissions—Android Developers (2019). https://developer.android.com/training/permissions/requesting/. Accessed 12 Mar 2019
5. Bagheri, H., Kang, E., Malek, S., Jackson, D.: Detection of design flaws in the android permission protocol through bounded verification. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 73–89. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_6
6. Bagheri, H., Kang, E., Malek, S., Jackson, D.: A formal approach for detection of security flaws in the android permission system. Formal Aspects Computi. **30**(5), 525–544 (2018)
7. Bagheri, H., Sadeghi, A., Garcia, J., Malek, S.: COVERT: compositional analysis of android inter-app permission leakage. IEEE Trans. Softw. Eng. **41**(9), 866–886 (2015)
8. Betarte, G., Campo, J., Cristiá, M., Gorostiaga, F., Luna, C., Sanz, C.: Towards formal model-based analysis and testing of android's security mechanisms. In: 2017 XLIII Latin American Computer Conference (CLEI), pp. 1–10. IEEE (2017)
9. Betarte, G., Campo, J., Luna, C., Romano, A.: Formal analysis of android's permission-based security model 1. Sci. Ann. Comput. Sci. **26**(1), 27–68 (2016)
10. Betarte, G., Campo, J.D., Luna, C., Romano, A.: Verifying android's permission model. In: Leucker, M., Rueda, C., Valencia, F.D. (eds.) ICTAC 2015. LNCS, vol. 9399, pp. 485–504. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-25150-9_28
11. Enck, W., Ongtang, M., McDaniel, P.: Understanding android security. IEEE Secur. Priv. **7**(1), 50–57 (2009)

12. Fragkaki, E., Bauer, L., Jia, L., Swasey, D.: Modeling and enhancing android's permission system. In: Foresti, S., Yung, M., Martinelli, F. (eds.) ESORICS 2012. LNCS, vol. 7459, pp. 1–18. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33167-1_1
13. Shin, W., Kiyomoto, S., Fukushima, K., Tanaka, T.: A formal model to analyze the permission authorization and enforcement in the android framework. In: Proceedings - SocialCom 2010: 2nd IEEE International Conference on Social Computing, PASSAT 2010: 2nd IEEE International Conference on Privacy, Security, Risk and Trust, pp. 944–951 (2010)
14. Tuncay, G.S., Demetriou, S., Ganju, K., Gunter, C.A.: Resolving the predicament of android custom permissions. In: Proceedings of the 2018 Network and Distributed System Security Symposium. Internet Society, Reston (2018)