ROLE BASED ACCESS CONTROL MODELS FOR ANDROID

by

SAMIR TALEGAON, M.S.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN ELECTRICAL ENGINEERING

COMMITTEE MEMBERS:
Ram Krishnan, Ph.D., Chair
Eugene John, Ph.D.
Wei-Ming Lin, Ph.D.
Ravi Sandhu, Ph.D.

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Engineering
Department of Electrical and Computer Engineering
December 2020

DEDICATION

To my parents Sunil and Anita

ACKNOWLEDGEMENTS

I express my most sincere gratitude towards Dr. Ram Krishnan. I have had the pleasure of performing research under his guidance for the past five years, and in doing so, I have learned the importance of perseverance, and diligent hard work. From the philosophy I incorporated from him, I learned to never back down and to be proud of every aspect of my work; and, I thank him for his constant support and motivation during my research at UTSA. From him, I learned what it means to be professional and I shall always commemorate the guidance I received from him.

I would like to express my gratitude to Prof. Ravi Sandhu for agreeing to be in my committee and for providing me with constructive feedback. I also thank him for shaping the domain of access control and cyber-security into what it is today; this dissertation is partially based on his work in role-based access control. Additionally, he has steered the Institute of Cyber Security (ICS) at UTSA in a distinguished manner, of which I am proud to be a member.

I would like to thank Prof. Wei-Ming Lin for his support and guidance in my Masters program. He has been a great inspiration for me, and his help and advice has been instrumental in my success, and is something I would never forget. Thank you, Sir!

I would like to thank Prof. Eugene John for providing me with inspiration and guidance not only in the academic field, but also in matters of health. I will always take this advice to heart and be a better example for everyone.

The College of Engineering staffs, Ms. LiPien Bein and Ms. Khanh Nyugen have helped me throughout these years with submissions and other administrative matters, and for this I thank them. I would also like to thank Institute of Cyber Security staff member Ms.Suzanne Tanaka for her help with administrative matters.

I would like to thank my parents, Dad Sunil Talegaon and Mom Anita Talegaon, without whom I would not be here this day. Their continued guidance and love throughout my life has been the bedrock upon which I could arrive at where I am today.

I would like to thank Nihar Bendre for being my friend, motivating me and helping me in my

personal life to be a better person. Thank you for always being there for me, believing in me, and helping me in every way possible.

This research was made possible by the National Science Foundation (NSF) grants HRD-1736209, CNS-1553696, and Department of Defense ARO Grant W911NF-15-1-0518.

This Masters Thesis/Recital Document or Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Masters Thesis/Recital Document or Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Masters Thesis/Recital Document or Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Masters Thesis/Recital Document or Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this Masters Thesis/Recital Document or Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Masters Thesis/Recital Document or Doctoral Dissertation attest to the accuracy of this statement.

December 2020

ROLE BASED ACCESS CONTROL MODELS FOR ANDROID

Samir Talegaon, Ph.D.
The University of Texas at San Antonio, 2020

Supervising Professor: Ram Krishnan, Ph.D.

Android had been one of the most widely used mobile operating systems in recent times, owing to its intuitive UI, wide range of applications in the app store - Google Play, and easy to use APIs that enabled developers to design immersive and entertaining apps. As with any popular and successful OS, Android had suffered from design issues as pointed by several works in the past; however, in each case, Google has adopted an on-the-spot fix policy to resolve such issues, and had kept the permission-based access control mechanism, largely intact.

Administration of permissions in Android had been an issue since the original Android was released in 2007. This issue is the direct assignment of permissions to applications in Android. Despite recent changes to the system, from the install time permissions, to the runtime permissions, this issue had not been dealt with, from a holistic perspective. However, before we dealt with this issue, we had to obtain a comprehensive understanding of permissions in Android. We analyzed URI permissions in Android, used by applications to facilitate inter-application data sharing; and, system permissions that granted access to hardware and software resources, to the applications. Our analysis of Android's URI permissions, which consisted of API 10 through API 22, and system permissions, for API 29, yielded quite a few peculiarities, that we have documented in this work. Following this, a formal mathematical model, denoted by ACiA_{α}(Access Control in Android model), which consisted of system and URI permissions in Android was built, from the source code and available documentation. This model was tested using carefully designed test-apps, to verify its accuracy. Our meticulous analysis yielded several issues with the permission-based mechanism, some of which corroborated with the previous works. In a bid to improve this model, we explored the role-based access control model (RBAC) for Android.

RBAC worked by letting the administrators assign object rights to roles, and then assigning

these roles to the subjects; Android, however, assigned permissions directly to applications, which is why Google chose to severely limit the number of permissions that were under user control. This indicated that, RBAC in Android would be beneficial, not only to ease the administration of permissions by users, but also, to overcome some of the well documented fatigue encountered by users as a result of their interaction with the permission prompts. To implement RBAC and its administration in Android, several models were constructed, and the best one, from our perspective was chosen. This scenario consisted of assigning roles directly to the applications, thereby adhering to the principle of least privilege. To build the user assignment (UA) and permission assignment (PA) relations, required for implementing the model into Android, we used several role-mining algorithms. Post comparing the generated roles from these algorithms, the MinNoise RMP algorithm was chosen because the roles generated by this algorithm were the most suitable for Android. Further analysis of the roles generated from this algorithm, revealed, that the above-mentioned fatigue could be mitigated, apart from enabling an easier mechanism for application developers to request access to resources using roles; this substantiated the claims of RBAC's benefits in Android.

A core RBAC mechanism was implemented in Android along with sessions. Within RBAC for Android, Applications would be granted roles upon user acceptance and could activate them when required, during their runtime. Furthermore, we designed three administrative models for RBAC in Android, that relied on the principle of administering RBAC with another RBAC mechanism. One of these models was chosen, based on simplicity, and was implemented in Android version - S. This implementation and its features are documented in this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACRONYMS

AOSP Android Open Source Project.

OS Operating System.

VM Virtual Machine.

NFC Near Field Communication.

ACiA Access Control in Android.

RBAC Role Based Access Control.

ARBAC Administration of Role Based Access Control.

RiA RBAC in Android.

ARiA Administration of RBAC in Android.

UA User-Assignment.

PA Permission-Assignment.

URI Uniform Resource Identifier.

API Application Programming Interface.

RAdAC Risk-Adaptable Access Control.

RMP Role Mining Problem.

UPA User Permssion Assignment Matrix.

CHAPTER 1: INTRODUCTION & MOTIVATION

Android has grown from a measly used mobile operating system since it was introduced in 2007, to one of the most widely used operating systems today [61], and, as mobile devices become cheaper, its proliferation continues to increase. It is built on a modified Linux kernel and uses a touchscreen as the primary mode of interaction with the users. The majority of the software that constitutes the Android OS, is open source, distributed as the Android open-source project (AOSP) [38]. The Play Store and other applications such as Gmail, Google Maps and Google Calendar, that ship with most Android devices, is proprietary software developed by Google. Google identifies different versions of the Android OS via an identifier and a version number, such as Android Oreo which has the version number 8.

To fully utilize the device features, Android users install applications from the Play Store. These applications require access to the device resources, for providing their functionality. Android devices contain a wide variety of software resources such as photos, videos, contacts, and songs; and hardware resources such as Bluetooth, NFC, WiFi and Camera. Access to such resources pose a significant hazard to the user's privacy (access to photos, contacts, messages) and security (phones can be used to unlock cars, homes, garages etc.), which warrants the use of a robust access control mechanism. The applications installed by the users request access to the resources from the Android OS, in the form of permission prompts. A permission is a string that is used by the Android OS to protect its resources, similar to a lock and key mechanism. Consecutively, Android seeks user approval to grant some of these requests & provide the necessary access to the resources [44]; this is illustrated in Figure 1.1.

Preliminary analysis of Android permissions revealed that there were too many permissions (over 100) required by the applications. Also, the modern powerful Android devices allow users to install a high number of applications. It is not possible for users, most of whom are not trained in the field of access control, to manage such a vast number of permissions. Traditionally, Android dealt with this issue by utilizing a grant all/deny all policy, until Android version 5.0, after which,

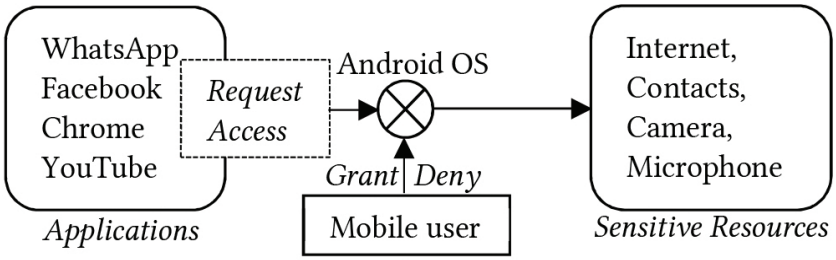


Figure 1.1: Permission-based access control in Android

they allowed the users to manage a partial set of permissions, known as dangerous permissions. However, all the permissions that applications need, cannot be administered by the users, partly due to the tedious nature of administration involving the direct assignment of permissions to applications. Since Google's approach for the resolution of this issue, results in users left with managing only a very small subset of permissions, building an access system for Android, that gives users control over almost all the permissions, is a non-trivial task.

Simplifying the administration of permissions, warrants an engineering of a completely new access control mechanism for Android. However, before achieving this, it is imperative to gain a comprehensive understanding of the current structure of permissions in Android, via a thorough mathematical formalization of Android permissions, which can then be analyzed. Upon reviewing the available literature on formalization of access control in Android (ACiA), we realized that it was insufficient to gain an in-depth understanding of the current permission model of Android, since most of the work is based on the older install time permissions. So, we formalized the ACiA, utilizing commonly used logical and set theory constructs into a model, called ACiA_{α}. Using this model, we could map the changes that would be required to be made, to improve the permission system in Android.

In the enterprise scenario, when the number of users and objects increased, a role-based access control (RBAC) [31] system was utilized, to ease the administrative burden faced by the administrators. RBAC achieves this, by associating objects with roles & then assigning these roles to the users. This simplifies access control, since, instead of managing access rights to individual objects, it is easier for the users to assign roles to objects. Furthermore, such roles can then be granted by the administrative teams, to other users to grant them the required access to the objects. The pro-

cess of forming meaningful roles for an RBAC system is known as role engineering. There are two major approaches to role engineering, the top-down approach, and the bottom-up approach. Since a top-down approach is infeasible in Android, due to the unavailability of the concept of job functions, we used the bottom-up approach to build roles, and demonstrated the feasibility of such an approach in Android.

The administration of RBAC with RBAC itself is a well-studied problem, and we explored some of the most well-known models of administrative RBAC, namely the ARBAC'97, ARBAC'99 and then ARBAC'02. The underlying principle of pre-requisite roles that is used by these models was deemed unfit for Android, since there does not exist any accepted notion of job functions for Android applications, which undermines the principle of risk limitation behind the pre-requisite roles. Also, since there is no accepted convention of the constituencies of roles (i.e.: the permission that are assigned to the roles (PA)), the administrative ease RBAC offers cannot be envisioned accurately. This is because, a proportionally higher number of roles with respect to the number of permissions, undermines the user's ability to easily manage the permission to role assignments (PA) and application to role assignments (UA). So, we built three new models for administration for RBAC, that are used to demonstrate the feasibility of the administration of RBAC, by the users in Android.

1.1 Problem Statement

Access control in Android has been rigid over the past few years. Even with the introduction of runtime permission system in API 23, the overall permission management mechanism has largely remained the same, that is, permissions are directly assigned to applications. Google often resorts to on-the-spot fixes for most of the issues encountered, related to access control. While such fixes provide some relief to the issues they address, it is our belief that a holistic and formalized approach needs to be taken, to resolve the issues related to access control, and build a robust permission mechanism for Android.

The role-based access control system has tremendous potential to mitigate administrative bur-

den, and thus promises substantial benefit for the Android OS. Prior works exploring RBAC in Android do exist, however, their main focus is application to permission management. Besides single person usage, shared Android devices are found in cases such as an employee attendance systems or a courier delivery system. Android devices are also used in security sensitive locations such as in the military, nuclear research, and the cyber-security field itself. Due to this wide array of use cases, multiple models that enable the users to administer permissions are required. Besides this, the administration of RBAC in Android is largely unexplored.

Keeping this in mind, we envision that it is feasible to develop and implement, an RBAC infused Android that can enable a holistic resolution of the issues with its permission management, and an administrative RBAC for Android to enable the users to be able to manage the permissions.

1.2 Summary of Contributions

The contributions of our work are presented below.

Android URI permission review (API 10 through 22)

Thorough analysis of Android's URI permission system including granting of uri permissions, revocation of uri permissions and delegation of uri permissions. This analysis is done on APIs 10 through 22 to grasp the developmental aspects of the data sharing model in Android.

Formalization of Access Control in Android (ACiA$_\alpha$)

A mathematical model of access control in Android using traditional logical and set theory constructs. This model reveals key aspects of access control including application installation, application uninstallation, permission grants, permission revocation, uri permission grants, uri permission revocation and delegation of uri permissions. A detailed description of the model is included in this work.

Modelling Role Based Access Control in Android

Role based access control is modeled in Android based on three major use cases for Android devices. We built three distinct models for RBAC in Android which are described in this work. To generate the permission assignment and user assignment relations in Android, we studied and

implemented several key role-mining algorithms available in the literature and chose the best algorithm which provided us with a small set of roles.

Modelling Administrative Models for RBAC in Android

Three new models for administration of RBAC in Android. A model employing a user prompt-based administration technique for UA and PA operations. A constraint-based model, wherein the user can set multiple constraint variables (called as `cvar`) that regulate the administrative RBAC operations. An automated model based on RAdAC, to manage UA and PA operations based on the risk they pose to the user's privacy and security.

Implementation of RBAC and its administrative model in Android

Selected RBAC and its administrative model is implemented in Android. The RiA_a model was chosen for RBAC in Android, while the ARiA_0 was chosen for its administration.

1.3 Related Publications

A formal model of access control in Android based on the work in Chapter 3 section 2, was accepted for publication in the SKM 2019 conference.

- Samir Talegaon and Ram Krishnan. "A Formal Specification of Access Control in Android", International Conference on Secure Knowledge Management in Artificial Intelligence Era. Springer, Singapore, 2019.

A formal model of access control in Android with extensions for the uri permissions in Android, based on the work in Chapter 3 section 1 was accepted for publication in the Information Systems Frontiers (ISF 2020) journal.

- Samir Talegaon and Ram Krishnan. "A Formal Specification of Access Control in Android with URI Permissions", Information Systems Frontiers, Accepted, September 2020.

Role based access control models for Android based on the work in Chapter 4 was accepted for publication in IEEE TPS 2020 conference.

- Samir Talegaon and Ram Krishnan. "Role-Based Access Control Models for Android", 2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA). Accepted, IEEE, 2020.

Administrative RBAC models in Android based on the work in Chapter 5 was accepted for publication in the JISIS 2020 journal.

- Samir Talegaon and Ram Krishnan (2020). "Administrative Models for Role Based Access Control in Android", Journal of Internet Services and Information Security (JISIS), 10(3), 31-46, August 2020.

1.4 Organization of Dissertation

This dissertation is organized in the following way. Chapter 2 involves background information and literature survey. In this chapter we provide an introduction to the access control in Android, and a literature review is also included, that places our work in the context of published related research.

Chapter 3 talks about the access control in Android model (ACiA_{α}), and describes its formalization using traditional mathematical constructs. Following this, we present the model in two separate tables, so as to divide the operations initiated by the users, as opposed to those initiated by the applications themselves.

Chapter 4 explains the role-based access control for Android, along with three separate models for distinct use cases. It also explains the role-mining techniques used by us, to algorithmically generate meaningful roles and show feasibility of the technique with respect to Android permissions.

Chapter 5 describes the administration of RBAC in Android, along with three separate models for distinct purposes. It also describes how a constraint-based administration approach to RBAC in Android can be utilized to further reduce the user burden in its management.

Chapter 6 describes the implementation of RBAC and administrative RBAC in Android, with respect to our chosen model. A diagrammatic representation of the underlying architecture used to

implement these models in Android is also included and described in this section.

Finally, Chapter 7 concludes this dissertation with conclusions and provides a perspective for future work.

CHAPTER 2: BACKGROUND AND LITERATURE SURVEY

In this chapter, the Android OS is briefly described, along with the relevant background information pertaining to role-based access control (RBAC), and role-mining. Proceeding this, a literature review is described that places our work in the current body of research and serves as one of the primary motivations for our research.

2.1 Background Information

This section describes the relevant background information on the Android OS, the Role based access control models and the role-mining algorithms used to generate roles.

2.1.1 The Android Operating System

Applications in Android request permissions, initially by declaring the permission in the Android Manifest file (Fig. 2.1), which is like an index of the application, and then programmatically requesting the permissions at runtime. Depending on the type of permissions requested, and the prior permissions granted, the Android OS may show the user permission prompts, that they can accept. If the user approves such prompts, the permissions are granted to the applications permanently.

Permissions are categorized as normal, dangerous and signature depending on the risk they pose to the user's privacy and security [43]. Normal permissions are automatically granted, and signature permissions are granted by the Android OS based on the signing certificate of the application, both of which cannot be revoked by the user. Dangerous permissions are those that pose a significant risk to the user's privacy and security. Applications need to request user approval for such permissions. In addition to these pre-defined permissions, each application can define its own set of permissions which control access to its components. These are known as application defined permissions and they vary according to the applications that are installed on a device.

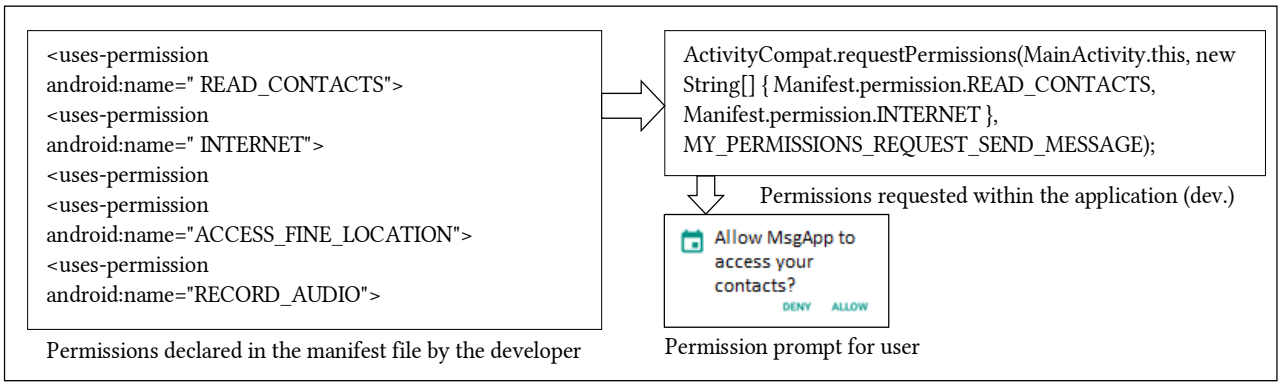


Figure 2.1: How permissions are declared, requested & approved in Android

2.1.2 Android Software Stack

Android OS is roughly divided into four main layers (see Figure 2.2). These are arranged from the Linux kernel, which is the base of the Android OS, up to the Application layer where all the installed applications reside.

Android devices are built on a modified Linux kernel, which is responsible for all basic tasks such as power management, device management and memory management along with interfacing the hardware. It contains all the necessary device drivers like camera, USB, Bluetooth, Wi-Fi, audio amongst others. Above the kernel layer, there are native libraries such as Media, OpenGL and SQL which are responsible for databases, playing audio & video, displaying graphics and more. This is the same level where the Android runtime operates, housing the Dalvik VM and other libraries. The Dalvik VM is similar to the JVM which is the Java Virtual Machine on the desktop computers. However, DVM is optimized for lower memory usage. Above the Android runtime, is the application framework which has the APIs for applications to use, such as UI, telephony, and location. This framework simplifies the use of core system components such as resource manager, activity manager and content providers. The next layer is the application layer, which contains all the system applications and third-party applications.

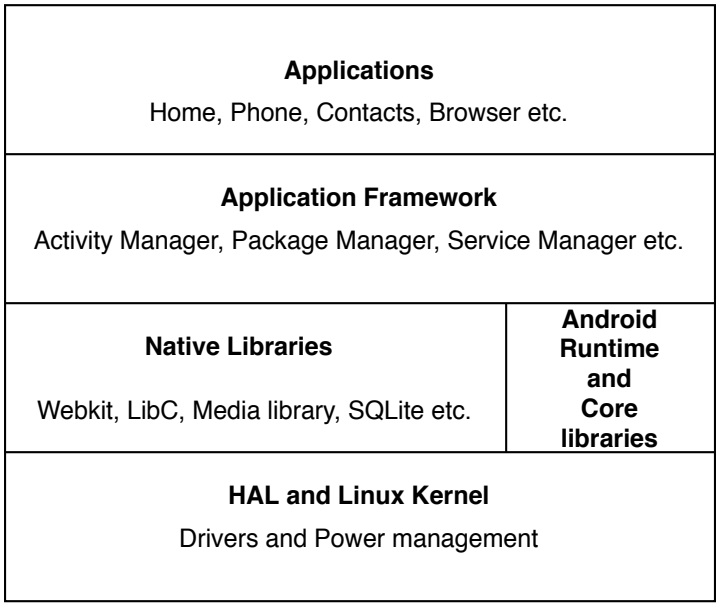


Figure 2.2: Android Software Stack

2.1.3 URI Permissions in Android

As mentioned before, applications in Android installed by the users need access to the device resources. Similarly, these applications need to access other application data as well. For example, an email application can provide temporary access to a photo editor application, so that the editor is able to work on an image. This temporary access is achieved through URI permissions in Android. URI stands for Uniform Resource Identifier, and applications that wish to share their data can have such a URI address by implementing a content provider. A content provider is an external window for stored data, which allows selective access to be granted to other applications for temporary use.

2.1.4 Role Based Access Control

RBAC [31] is a security administration mechanism used in enterprise scenarios, to assist administrators in management of user access to resources. RBAC works by assigning permissions to roles, and assigning select roles to the objects. There are 3 different models of RBAC; core RBAC, hierarchical RBAC and constrained RBAC. The core RBAC (see Figure 2.3) provides the necessary basic implementation of RBAC. The RBAC in Android built and described in this work, is based

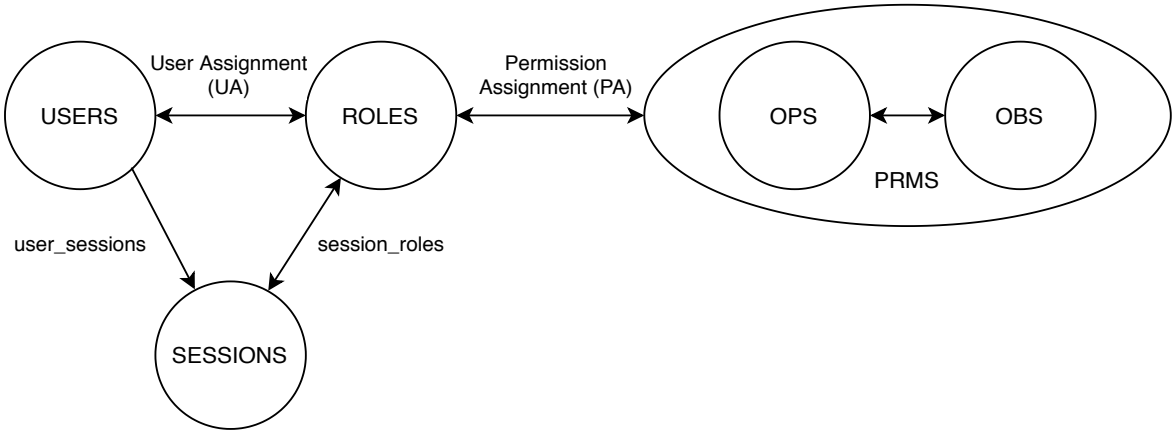


Figure 2.3: Core RBAC

on the core RBAC model.

2.1.5 Role-mining

Role-mining is the bottom up approach of role-engineering [21], in which, roles are generated algorithmically from the user-permission assignment matrix. Several role-mining algorithms were used in this work, to build roles for Android. The goal was to come up with a set of roles, that are comparatively small in size with respect to the number of permissions. These roles should have good coverage i.e.: coverage of permissions so that all the permissions are assigned to at least one role. The algorithms utilized in this work are: the Fast Miner and Complete Miner algorithms [72], Basic RMP [69], δ RMP [71] and Min-Noise RMP [40]. The roles generated by these algorithms were analyzed, and the Min-Noise RMP algorithm was chosen for role generation.

2.2 Literature Review

In this subsection, our work is compared and contrasted with the available literature.

2.2.1 Android Permissions

System permissions in Android protect access to hardware resources such as camera, Bluetooth, NFC, and Wi-Fi amongst others. Enck et. al. [26] provide an overview of Android's app level

security policies and developed a tool (Kirin) to certify whether applications should be installed on a device after comparing the security policy of the application extracted from its manifest, with the security requirements of the device owner. It also said that due to the enhancements done to the mandatory access control model in the Android OS, it is difficult for developers to define through security policies which can definitively protect their applications. Shabtai et al. [59] performed a comprehensive security assessment of Android's framework and made security recommendations based on this assessment. Fragkaki et al. [32] developed a formal model to analyze Android's permission system and built a system, called as SORBET, to hold a few desired security properties which were not found in the Android's permission model. Their model is primarily based on Android's URI permissions and the system permissions in Android are not detailed enough. Gustavo et. al. [17] furthered the development of a comprehensive formal model of Android's system permissions with elements which, according to them, were partially analyzed in prior works. These works consider either Android's URI permissions or system permissions, but not both. The permission model in Android, described in this work, is geared towards providing a more holistic perspective of access control in Android.

Felt et al. [29] used Stowaway to determine the over privilege in Android applications which, at times, was due to developers wanting to additional system permissions, including some which their applications may need in the future, but not in current versions; while at other times, there was developer confusion due to insufficient documentation available pertaining to the APIs. Allowing app developers to specify security policies for their applications which dictate how other applications access their information is a novel concept which puts the onus of security on the developers [50]. Works such as [66, 75] discuss the system permission evolution in Android. These papers make useful contributions to modeling Android security; however, their work primarily adds on to the understanding of system permissions in Android. Our intention is to extract information on the URI permissions as well as system permissions, to provide a holistic understanding of access control in Android.

2.2.2 Formalizing Access Control in Android

Many works have targeted the Android's permission mechanism to understand how it internally works while others have proposed new methods for implementing access control in Android. PScout [11] analyses Android permission system in a bid to provide insight in to Android's API permissions, statically. Barrera et. al. [14] perform empirical analysis of 1,100 Android applications and identified the frequently used permissions and suggest improvements to Android's permissions model to increase the granularity for these permissions, while reducing it for the non-frequent permissions. Stowaway [29] detects permission over-privilege by mapping API calls to permissions, and identifying those permissions which are not used in conjunction with any API call. These works prove an insight in to the Android's permission system, however they do not propose any new access control methods and only suggest improvements on the Android's permission-model.

VetDroid [77] uses 1,249 free applications for performing dynamic analysis which has advantages over static analysis and it uses results from this analysis to identify potential security flaws in Android applications. Fang et. al. [28] investigate issues in Android's security system and identify flaws relating to course granularity of permissions, un-friendly permission management mechanism, insufficient documentation on API permissions and a few attacks experienced by Android applications such as permission escalation and TOCTOU (Time of Check & Time of Use). These works provide novel insights into the runtime issues experiences by Android applications, but are not directed towards modifying the permission-based access control mechanism.

2.2.3 Role Based Access Control in Android

Few works have implemented RBAC in Android, however, the true form of core RBAC which includes the use of sessions along with role activation and de-activation remains largely untested. Abdella et.al. [10] implemented RBAC in Android, however, their focus is on context aware rules which govern the permission grants & revocation. Rohrer [53] implemented DR-BACA model, which is an adaptation of the RBAC model, in Android, which considers the dynamic nature of

contexts and controls application requests for permissions using factors such as location of the device, time or date on the device, and events which take place on the device. However, these modifications to the Android OS do not make the use of sessions in any way, nor do they mention role activation & de-activation. It is for this reason, this work proposes a hitherto un-implemented RBAC model for improving security and privacy in the Android OS.

2.2.4 Administration of RBAC in Android

Few works have implemented RBAC in Android, however, works which implement administrative models for RBAC in Android are yet to be found. As such, this work is the first such attempt to create a user friendly administration model for Android. A few works that implement RBAC in Android, without a user controlled administrative model, are described below.

Abdella et.al. [10] implemented RBAC in Android, by assigning roles to permissions and granting these roles to applications. They use contextual information to limit the permissions that can be activated within a given role that has been granted to an app, to reduce user administrative burden. However, roles are arbitrarily created in their model. This work is based on the belief, that, to fully realize the benefits of RBAC in Android, roles need to be carefully crafted because, arbitrarily created roles hamper the effective advantage gained by using RBAC in Android. However, in this prior work, the administration of RBAC in Android is done automatically via the Policy Decision Manager without letting the user select which contexts they wish to consider. It is our belief that access control for Android must take into consideration user input prior to making such decisions. We have incorporated user input in all our models that lets the user decide upon the operation's successful execution.

Rohrer [53] implemented DRBACA model, which is an adaptation of the RBAC model, in Android, which considers the dynamic nature of contexts and controls application requests for permissions using factors such as location of the device, time or date on the device, and events which take place on the device. However, the 6 tupled rule mechanism is too complex to be understood by normal Android users, and is geared more towards the enterprise environment. We

have created our models with the Android users in mind, most of whom are not knowledgeable in the field of access control.

Administration of RBAC has undergone extensive work in the past. Sandhu et.al. [58] (ARBAC96) introduced the distinctions between an ordinary role and permission, with administrative roles and administrative permissions. It was also the first model that introduced the administration of RBAC with the help of RBAC itself. ARBAC97 [55, 56] introduced a decentralized administration for UA, PA and RR assignment relations. Our models obey this distinction, which can also be seen in [49, 57]. It also defined the *can_assign* and the *can_revoke* relations which dictate restrictions on administrative operations, and depend on the concept of pre-requisite roles to limit which roles can be granted to a user. Since Android does not maintain permissions with a hierarchy in mind, the concept of pre-requisite roles cannot be materialized in Android.

In ARBAC97, when administrators assigned roles to users, such assignments enabled further role assignments by satisfying the pre-requisite role qualifications. Thus, a single role assignment presented an increased risk for administrators. Distinctions between mobile and immobile users in ARBAC99 [57] enabled administrators to grant the users, memberships in roles, without increasing the security risk based on such a membership. In ARBAC97, pre-requisite roles presented an increased burden of administration, by requiring administrators to grant all the chained pre-requisite roles before they could grant a particular role. ARBAC02 categorized an organization into organization units which comprise of user pool and permission pool. These pools enable administrators to directly assign a user to a user pool, and a role to the user without needing to pre-assign all pre-requisite roles. Our administrative models do not employ a pre-requisite role requirement, because in case of Android, the administrators are assumed to be the device owner, app developers and Google itself. Also, pre-requisite roles require a hierarchical design of the system being administered, however, no such hierarchy exists in the permissions or roles for Android.

The UARBAC [45] paper established six design requirements based on three security principles i.e.:flexibility and scalability, psychological acceptability, and the economy of mechanism. From these six principles, our models obey the requirements for equivalence and reversibility, because

the operations presented in our models do not create additional side effects i.e.: when a role is granted to an app, this does not allow for any other role to be automatically granted to this app. Reversibility is the requirement by which an operation can be reversed, with the help of an opposite operation. All the "assign" operations presented in this work, are reversed by the appropriate "revoke" operations.

SARBAC [22] introduces the concept of administrative scope which indicates modifiable role hierarchy. Our models do not contain administrative role hierarchies; however, operations are processed either depending on a set of modifiable constraints set by the administrator i.e.:device owner, or automatically via the RAdAC system [47].

The administrative models presented in this work, aim to mitigate user burden on the administration of RBAC in Android. Particularly the constraint based model reduces the user burden by letting the users pre-set constraint values, which are then matched to the values during the authorization checks for the operations. The RAdAC based model makes automated access control decisions by letting the user dictate the risk that is acceptable for any given operation. The system then calculates the situational risk, based on which it makes an access control decision. Since our models are built for Android, no administrative role hierarchy exists in our models.

CHAPTER 3: FORMALIZING ACCESS CONTROL IN ANDROID

This chapter describes the motivation for formalization of URI permissions, the carefully designed meticulous tests that were conducted to build a formal model for permissions in Android and the formal model for access control in Android.

3.1 Formalizing URI Permissions in Android (API 10 - API 22)

Owing to the lack of sufficient documentation, Android's URI permission mechanism needed to be understood. For this, we tested Android API 10 through 22 via carefully designed inter app tests, to gather information on how inter app data sharing worked. This is crucial for our formalization, since it is often the area where data leaks occur, due to developer error resulting in applications sharing their data without any protection in place.

3.1.1 Motivating Scenario for Studying URI Permissions in Android

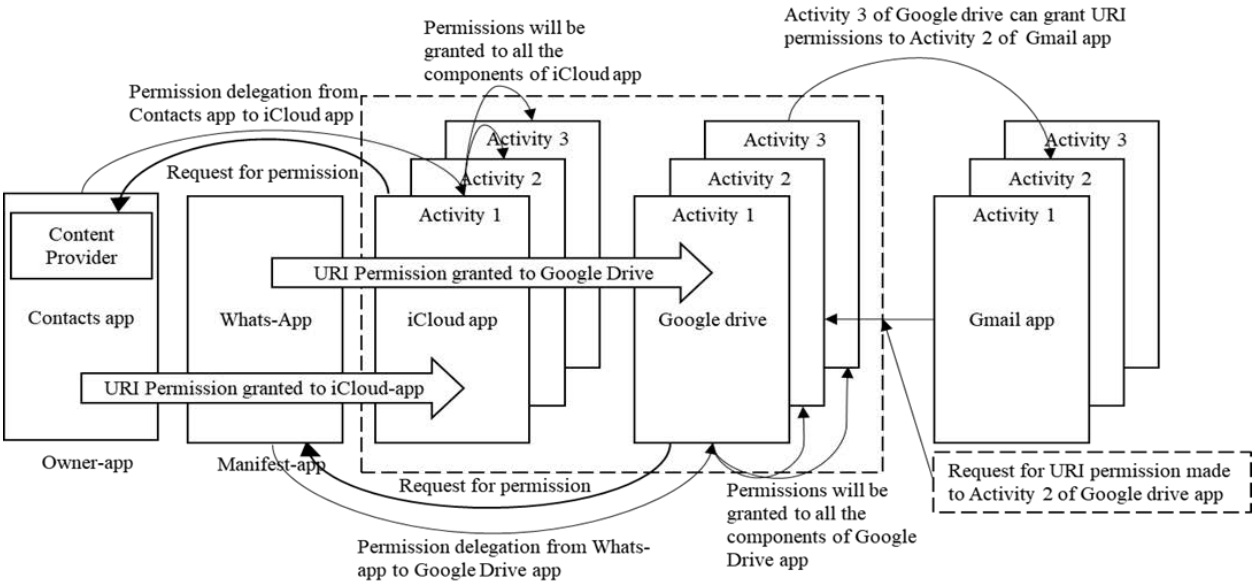


Figure 3.1: Test Results (API 22)

Consider an example scenario consisting of 4 applications installed on a user's Android device (see Figure 3.1). The first application is the Contacts application which has a content provider.

As we discussed in the previous section, the Contacts application can selectively share its data by protecting its content provider with a URI permission. The second app is Whatsapp for which the user wants access to the Contacts application's URI so they the user can message his friends and family. Requesting access to the URI permission in its manifest file is similar to the example shown in Figure 3.1 and Whats-application gets this permission since the user accepted to allow it. Therefore Whats-application gets the access to Contacts application URI. The third and fourth applications are iCloud and Google Drive applications and the user wishes to backup his Contacts information to these online storage mediums. For this reason, these applications need occasional access to Contacts-application's data and can be granted access to the URI when needed, via Intents and the GrantUriPermission method, by either of the two applications (Contacts-application or Whats-application). Keeping in mind this operating scenario, many questions arise as follows.

- Which application is allowed to delegate permissions to other applications? (Contacts-application, Whats-application, Google Drive application or iCloud application). Whether the Contacts application can provide the data to Google drive application directly to facilitate the backup which the user intends to perform routinely?

- Once permissions are delegated to iCloud or the Google Drive application, how long will the permissions remain with the application? (Until revoked manually, the application stack finishes or until the device is rebooted?). The user wishes to know, if the access to data, which was granted will remain indefinitely with the Google Drive and iCloud applications, or is revoked once the backup finishes?

- Can the iCloud or Google Drive application re-delegate these permissions to other applications, and whether any conditions exist on the scope of re-delegation? The user is concerned whether the Google Drive application is going to provide access to the Contacts-application data to any other applications willy-nilly, which he did not intend to occur.

- The last but not the least, what changes do the different API levels in Android bring to such URI permissions, their delegation, re-delegation and revocation? The user is concerned if he

upgrades his phone, whether all the applications will work as required, or their functionality will break with the new phone.

Answering these questions, requires extensive testing to be done in a methodical way, so the unabridged scenario of the evolution of Android's URI permission system is better understood and helps to evaluate the potential improvements.

3.1.2 Methodology for Testing the URI Permissions in Android

Six basic applications were designed and built, which correspond to the applications discussed in the previous section, and are detailed below. In order to thoroughly understand the intricate intertwined behavior of Android's URI permission system, a total of 96 test points which tests the behavior of URI permission system based on the following parameters for every API in consideration, were designed. The APIs - 10, 16-19, 21-22, were used for URI permission analysis.
These six applications are described below.

- **Owner-app or A1**: Owner-app is the application which protects its content provider with URI permissions and wants to share data with other applications on the device. This app corresponds to the Contacts app.

- **Manifest-App or A2**: App A2 requests these URI permissions in Manifest file and will be called the Manifest-app. This app corresponds to the WhatsApp application.

- **Delegated-applications or A3, A4, A5 and A6**: Dummy applications used for testing purposes having 3 activities each and methods for URI permission re-delegation to one another. These applications 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. These applications correspond to the Google Drive and iCloud applications discussed previously.

3.1.3 Parameters for testing.

The first parameter we consider is the attributes (exported and granturipermission) in the manifest file. The second parameter, is the app scenario for testing and finally the third parameter is the question being asked with regards to the URI permissions; these parameters are discussed in detail below.

- **Parameter 1 - Attributes for the provider tag.** The first parameter for our tests is the provider tags in the manifest and their truth values consisting of 4 distinct scenarios. This is shown in Table 3.1, as the first column of the table, namely the attributes are exported and GrantUriPermission.

- **Parameter 2 - Four distinct testing scenarios for the 6 applications noted above.**

  Scenarios (with applications in Consideration (AiC) noted on point):

  - AiC : Owner-app/A5/A6: The owner-app declares permission to guard its content provider as well as requests the permission in Manifest. Other applications have regular access to this URI permission and may be granted URI permissions depending on the other parameters.
  - AiC : Owner-app/Manifest-app/A3/A4/A5/A6: The Owner-app declares permission to guard its content provider & requests the permission in Manifest. Manifest-app A2 requests access to the URI permissions at install time and is granted by the user. No other applications have regular access to this URI permission and may be granted URI permissions via Intents and the GrantUriPermission method, depending on the other parameters.
  - AiC : Owner-app/Manifest-app/A3/A4/A5/A6: The Owner-app declares permission to guard its content provider but does not request it in Manifest. Manifest-app A2 requests these URI permissions. No other applications have regular access to this URI and may be granted URI permissions depending on the other parameters.

- AiC : Owner-app/A2/A3/A4/A5/A6: The Owner-app does not declare permission to guard its content provider in Manifest. No other applications have regular access to this URI permission however, point to be noted that the URI itself is not being protected by permissions. (Note that A2 is no longer a Manifest-app in this scenario as no permissions exist to protect A1's content provider, no manifest permission can exist)

These are noted as "scenario X" as shown in the Table 3.1 below where X is the scenario in consideration.

- **Parameter 3 - Six questions to establish the scope of URI permissions, their re-delegation and revocation in Android applications.**

1. How does Owner-app/Manifest-app grant URI permissions to other applications which do not have such access? (This is mainly the question pertaining to syntax of the grant operation depending on the mechanism intended for use e.g.: Implicit/Explicit Intent, GrantUriPermission method etc.)
2. How long do the Delegated-applications keep their permissions, once granted? (Does it last until reboot or until revoked or any other condition?)
3. How does URI permission re-delegation work with regards to Delegated-applications? (When the Delegated-applications receive the URI permission, how can they re-delegate it further to other applications?)
4. How long do the permissions being re-delegated by Delegated-applications to other applications last? (Does it last until reboot or until revoked or any other condition?)
5. Which app can revoke delegated permissions? (Can the Owner-app, Manifest-app or Delegated-app revoke the delegated permissions?)
6. How can permissions be revoked? (This is mainly the syntax for revocation.)

Example Test 1 considers API 22 in Android and answers these 6 questions with regards to the other 2 parameters. The answers to these questions reveal some peculiar and some bizarre

observations which are described in the results section.

Before explaining the results for our testing, let us look at a few concepts in Android which help to better understand the testing done, namely the app stack and activity instances.

App-stack: The app-stack consists of all the activities for any app which have started and not finished. These activities remain in the app stack until the user presses the back button or uses the multi-tasking window to manually end these activities. Whenever dealing with app-stack we keep in mind that the stack behaves in a last-in-first-out (LIFO) manner.

Activity instances: applications in Android usually are launched by the user after pressing the app icon. However, applications can also be launched by other applications and sometimes this leads to multiple instances for activities being spawned in the stack. To make testing easier we opted for a singleInstance mode on each of our activities to avoid confusion when dealing with permission revocation.

The results of tests performed are described below. While the detailed results for all APIs are of a considerable size and cannot be displayed here, we will observe the results from API 22's point of view so that we can understand the effectiveness of the testing done.

3.1.4 Working of URI Permissions in Android

The results of our tests occupy more than a few excel sheets and cannot be efficiently displayed here, but instead we will look at a few special observations for the URI permission system. While displaying the results we should note that these results are valid for APIs 10,16-19 and 21-22 as respectively indicated.

The questions which arose when we described the example scenario in section 4 are answered by our tests and are noted below.

Q1: Which app is allowed to delegate permissions to other applications?

Solution: Prior to API 16, only those applications which requested URI permissions in the Manifest file were allowed to delegate this further to other applications. After API 16 the Owner-app was given special access to its own URI and did not require to be requested at

Table 3.1: Test Results (API 22)

<table>
<tr><th rowspan="2">Manifest Attributes
(Parameter 1)</th><th colspan="3">Scenario → 1 to 3</th></tr>
<tr><th>Implicit Intent</th><th>Explicit Intent</th><th>GrantUriPermission</th></tr>
<tr><td>Exported = False
GrantUriPermission = False</td><td colspan="3">Solution 1, 2, 3, 4, 5, 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 (in fact no other app except Owner-app gets access to the URI and owner cannot delegate any permission)</td></tr>
<tr><td rowspan="8">Exported = False
GrantUriPermission = True</td><td colspan="3">Solution 1: As exported attribute is false the Manifest-app does not have access to the URI however Owner-apps have permissions and can delegate them since GrantUriPermission is true (Owner has permissions irrespective or permissions requested in manifest or not)</td></tr>
<tr><td colspan="3">Solution 2: The Delegated-apps can receive permission only from Owner-app which it keeps until →</td></tr>
<tr><td colspan="2">Device is Rebooted, App-stack finishes or the RevokeUriPermission method is invoked in case of permissions granted with Intents</td><td>Device is Rebooted, RevokeUriPermission method is invoked in case of permissions granted with the GrantUriPermission method</td></tr>
<tr><td colspan="3">Solution 3: The Delegated-apps receive permission from Owner-app and can delegate this permission using Intents and GrantUriPermission method</td></tr>
<tr><td colspan="3">Solution 4: The Delegated-app can receive chain permission from Owner-app→DA-1→DA-2 which they keeps until:</td></tr>
<tr><td colspan="2">Device Reboots, the App-Stack finishes, or the RevokeUriPermission method is called in case of permissions granted with Intents</td><td>Device Reboots, the RevokeUriPermission method is called in case of permission granted with the GrantUriPermission()</td></tr>
<tr><td colspan="3">Solution 5: Manifest-apps do not have permission to URI and cannot revoke access to such permission however, the Owner-app can execute RevokeUriPermission method which revokes all access to the said URI from all apps which have been granted the permission via Intents or via the GrantUriPermission method</td></tr>
<tr><td colspan="3">Solution 6: Only the Owner-app can run the RevokeUriPermission command to revoke permission from all other apps to which they were granted during run time.</td></tr>
<tr><td rowspan="2">Exported = True
GrantUriPermission = False</td><td colspan="3">Solution 1: As exported is true the Manifest-app and Owner app has access to URI but neither can grant permission as grantUriPermission is false.</td></tr>
<tr><td colspan="3">Solution 2, 3, 4, 5, 6: There is no Delegated-app as permissions cannot be delegated due to GrantUriPermission attribute being false.</td></tr>
<tr><td rowspan="6">Exported = True
GrantUriPermission = True</td><td colspan="3">Solution 1: As the exported and GrantUriPermission attributes are true, the Manifest-app can grant the URI permissions via Implicit Intent using syntax {intent.setAction("Action string")}</td></tr>
<tr><td colspan="3">Solution 2: The Delegated-apps keeps the URI permission until their app-stack (all activities) finish or until RevokeUriPermission method is invoked by either Manifest-app or Owner-app or until the device reboots.</td></tr>
<tr><td colspan="3">Solution 3: The Delegated-apps can re-delegate the URI permissions via Implicit/Explicit Intents and the GrantUriPermission method.</td></tr>
<tr><td colspan="3">Solution 4: The Delegated-apps keeps permission until app stack (all activities) finishes or until the Manifest-app or Owner-app invoked the revokeUriPermission method or until Device Reboots</td></tr>
<tr><td colspan="3">Solution 5: Only Manifest-apps and Owner-apps can revoke permission via the RevokeUriPermission method</td></tr>
<tr><td colspan="3">Solution 6: Manifest-apps and Owner-apps can execute the revokeUriPermission method which revokes URI permission from ALL apps (Except Manifest-apps and Owner-app itself)</td></tr>
</table>

install time.

Q2: Once permissions are delegated to iCloud or the Google Drive app, how long will the permissions remain with the app?

Solution: Once a URI permission is delegated to another app with Intents, it remains with that app until all its activities have ended which means that the user pressed back button while on its activities and none are currently executing in the background. If, however the permission was granted via the GrantUriPermissions method, they remain with the app until manually revoked by executing the RevokeUriPermissions method. Point to be noted is that all such permissions are automatically lost when a device is rebooted.

Q3: Can the iCloud or Google Drive app re-delegate these permissions to other applications, and whether any conditions exist on the scope of re-delegation?

Solution: The URI permisisons which these Delegated-applications have can be further re-delegated by those applications using Intents and GrantUriPermissions method.

Q4: What changes do the different API levels in Android bring to such URI permissions, their delegation, re-delegation and revocation?

Solution: It would be surprising to know that when we tested API 10 for URI permissions, we found a peculiar condition. The Owner app did not get any sort of permissions to its own content provider to delegate the permission to other applications. This meant that the Owner app must request permissions to its own content provider to enable it to delegate it to other applications. From API 10 to API 16 a major change was implanted in the URI permission system. This change allowed the Owner-applications to gain permanent access to its own URI and hence allowed it to delegate its URI permissions to other applications. No major changes were detected after API 16 to API 22.

3.2 Formalization of Access Control in Android

Post obtaining a better understanding pertaining to URI permissions in Android, a comprehensive formal model of access control in Android is described in this section. ACiA_α was built via reading the developer/source code documentation [5, 44], reading the source code itself and verifying our findings via inter-app tests. The ACiA_αmodel is specified below.

In the normal course of action, the Android users download many applications from the Google Play Store; 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 applications the OS uses many different APIs which we summarize as APK Extractor Functions, and, as shown in the Figure 3.2, these functions assist in the installation procedure. Upon successful installation, all the necessary device entity sets and relations are updated as shown in Table 4 (InstallApp operation).

3.2.1 Experimental Setup for Testing Android Permissions

After model extraction 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 application installation, uninstallation, permission grants and revocation etc. A brief overview on the testing methodology is explained in the section below.

A simple three app test environment was designed, and was adapted for each individual test. The applications used for these tests were dummy applications with two activities and one service component; and, where required, the applications 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 4 test parameters (TP) are considered (see Table3.2) which include app installation, app un-installation, install sequence & un-installation sequences; a worst case testing was done for validating issue #1.

Example Test 1. For brevity, we demonstrate a simple test that we conducted to check whether permission definitions were changed in accordance with the applications that were present on a

Table 3.2: Test parameters used for ACiA$_\alpha$model evaluation

TP1[1]	*Install Procedure* e.g.:$adb push & 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

device. The 3 above mentioned applications were designed to define a single permission, but into 3 distinct permission groups i.e.: pgroup1, pgroup2 & pgroup3. Upon installation of app1, the permission p1 was defined on the device into the permission-group "pgroup1"; following this, applications 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 applications, 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 applications were removed using the command line (something which only developers use anyway). This makes the issue more relevant, since users normally uninstall applications using the GUI and not the command line tools.

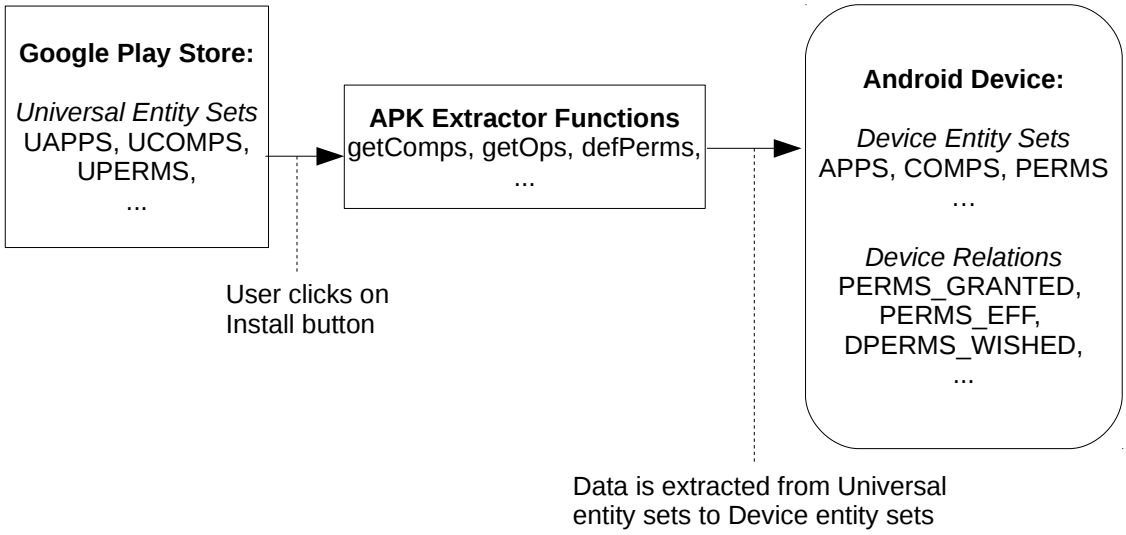


Figure 3.2: Building blocks of the ACiA

3.2.2 Building blocks of ACiA$_\alpha$

ACiA$_\alpha$ operations utilize certain element sets, functions and relations that are listed in Tables 3.3,3.4 & 3.5. Table 3.3 shows primary data sets from the Google Play Store (Universal Entity Sets - column 1) and any generic Android device (Device Entity Sets - column 2). It is to be noted that not all universal entity sets have corresponding device entity sets, and vice-versa.

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 work.

- 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 work we only consider the base protection-levels i.e.: $\mathbf{normal}$, $\mathbf{dangerous}$ and $\mathbf{signature}$.

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.

Table 3.3: 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 3.4: 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
`getAppSign`: UAPPS $\rightarrow$ USIG
`getCusPerms`: UAPPS $\nrightarrow 2^{\text{UPERMS}}$
`defPgroup`: UAPPS $\nrightarrow 2^{\text{UPGROUP}}$
`getCusPermProtlvl`: UAPPS $\times$ UPERMS $\nrightarrow$ UPROTLVL
`defPgroupPerm`: UAPPS $\times$ UPERMS $\nrightarrow$ UPGROUP
`wishList`: UAPPS $\nrightarrow 2^{\text{UPERMS}}$

Table 3.5: 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	`defAppsPerms`: 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
DROLES_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 $\rightarrow$ {**SemiPermanent**, **Temporary**, **NotGranted**}
	`uriPrefixCheck`: APPS $\times$ URI $\times$ DATAPERMS $\rightarrow \mathbb{B}$

Table 3.6: 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 $\rightarrow \mathbb{B}$

- 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, $\text{DATAPERMS}_\text{b}$ = { **dpread**, **dpwrite** } and $\text{DATAPERMS}_\text{m}$ = { **mpersist**, **mprefix** , **none**} and therefore, DATAPERMS = $\text{DATAPERMS}_\text{b} \times \text{DATAPERMS}_\text{m}$ = { (**dpread**, **none**), (**dpwrite**, **none**), (**dpread**, **mpersist**), (**dpread**, **mprefix**), (**dpwrite**, **mpersist**), (**dpwrite**, **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 3.4.

- `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).

- `getAppSign`, 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).

- `getCusPerms`, 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).

- `getCusPermProtlvl`, 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.
  `getCusPermProtlvl`$(ua, up) = pl_1 \Rightarrow$ `getCusPermProtlvl`$(ua, up) \neq pl_2$.

- `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[1]. Note that, $\forall a \in$ UAPPS, $\forall p \in$ UPERMS, $\forall pg_1 \neq pg_2 \in$ UPGROUP.
  `defPgroupPerm`$(ua, up) = pg_1 \Rightarrow$ `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.

[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!

Relation names are two words separated by an underscore and in upper case. 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 3.5

- APP_COMPS, a one-to-many relation mapping application to its 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

  - `ownerApp`, a function mapping application component to their owner application. Note that, a component can only belong to a single application. So, $\forall c \in$ COMPS. (`ownerApp`(c), c) $\in$ 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, `appComps`(a) = {$c \in$ COMPS | (a, c) $\in$ APP_COMPS}.

- COMP_PROTECT, a relation that maintains the permissions that are required for operations to be performed on application components. Not all components may be protected by permission. 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. A single application can define (and this own) multiple authorities however multiple applications may not own the same authority. In fact an application that tries to re-define an already defined authority on an Android device, 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, $\forall a \in \text{APPS},\ \forall p \in \text{PERMS},\ \forall pg_1 \neq pg_2 \in \text{PGROUP},\ \forall pl_1 \neq pl_2 \in \text{PROTLVL}.\ (a,\ p,\ pg_1,\ pl_1) \in \text{PERMS\_DEF} \Rightarrow \big((a,\ p,\ pg_2,\ pl_1) \notin \text{PERMS\_DEF} \wedge (a,\ p,\ pg_1,\ pl_2) \notin \text{PERMS\_DEF}\big)$

  - `defAppsPerms`, 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, $\texttt{defAppsPerms}(p) = \{a \in \text{APPS} \mid (a,\ p,\ pg,\ pl) \in \text{PERMS\_DEF}\}$

  - `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, $\forall a \in \text{APPS}, \forall p \in \text{PERMS}, \forall pg \in \text{PGROUP}.\ \texttt{defPgroup}(\texttt{a}, \texttt{p}) = \texttt{pg} \Rightarrow \big((\texttt{a}, \texttt{p}, \texttt{pg}, \texttt{pl}) \in \text{PERMS\_DEF} \wedge \texttt{pl} \in \text{PROTLVL}\big)$
- `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, $\forall a \in \text{APPS}, \forall p \in \text{PERMS}, \forall pl \in \text{PROTLVL}.\ \texttt{defProtlvl}(\texttt{a}, \texttt{p}) = \texttt{pl} \Rightarrow \big((\texttt{a}, \texttt{p}, \texttt{pg}, \texttt{pl}) \in \text{PERMS\_DEF} \ \wedge \ \texttt{pg} \in \text{PGROUP}\big)$

- 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. Multiple applications from the same developer having the same signature may attempt to define the same permission; in this case, the first application that gets installed is recognized as the defining application for such a permission. 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, $\texttt{effPerms}(\texttt{a}) = \{\texttt{p} \in \text{PERMS} \mid (\texttt{a}, \texttt{p}, \texttt{pg}, \texttt{pl}) \in \text{PERMS\_EFF} \wedge \texttt{pg} \in \text{PGROUP} \wedge \texttt{pl} \in \text{PROTLVL}\}$

- `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, $\forall p \in \text{PERMS}.\ \texttt{effPgroup}(\texttt{p}) = \texttt{pg} \Rightarrow \big((\texttt{a}, \texttt{p}, \texttt{pg}, \texttt{pl}) \in \text{PERMS\_EFF} \wedge \texttt{pg} \in \text{PGROUP} \wedge \texttt{pl} \in \text{PROTLVL}\big)$
- `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, $\forall p \in \text{PERMS}.\ \texttt{effProtlvl}(\texttt{p}) = \texttt{pl} \Rightarrow \big(\texttt{a}, \texttt{p}, \texttt{pg}, \texttt{pl}) \in \text{PERMS\_EFF} \wedge \texttt{pg} \in \text{PGROUP} \wedge \texttt{pl} \in \text{PROTLVL}\big)$

- DROLES_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, $\forall a \in \text{APPS},\ \forall p \in \text{PERMS}.\ (a,\ p) \in \text{DROLES\_WISHED} \Rightarrow p \in \texttt{wishList}(a) \wedge \texttt{effProtlvl}(p) = \text{dangerous}$
  - `wishDperms`, the mapping of an application to a set of dangerous permissions requested by it in the manifest. Formally, $\texttt{wishDperms}(a) = \{p \in \text{PERMS} \mid (a,\ p) \in \text{DROLES\_WISHED}\}$.
- PERMS_GRANTED, a many-to-many relation mapping applications to the permissions granted to them. Note that, $\forall a \in \text{APPS}, \forall p \in \text{PERMS}.\ (a, p) \in \text{PERMS\_GRANTED} \Rightarrow p \in \texttt{wishList}(a)$
  - `grantedPerms`, the mapping of an application to a set of permissions granted to it. Formally, $\texttt{grantedPerms}(a) = \{\ p \in \text{PERMS} \mid (a, p) \in \text{PERMS\_GRANTED}\}$.

- 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, $\forall a \in \text{APPS}$, $\forall uri \in \text{URI}$, $\forall dp_b \in \text{DATAPERMS}_\text{b}$, $\forall dp_m \in \text{DATAPERMS}_\text{m}$, $\forall dp \in \text{DATAPERMS}$.
    $\texttt{grantNature}(a,\ uri,\ dp) = \textbf{SemiPermanent} \Rightarrow \big(dp_m = \textbf{mpersist} \wedge (a,\ uri,\ dp) \in \text{GRANTED\_DATAPERMS}\big) \quad \vee$
    $\texttt{grantNature}(a,\ uri,\ dp) = \textbf{Temporary} \Rightarrow \big(dp_m = \emptyset \ \wedge \ (a,\ uri,\ dp) \in \text{GRANTED\_DATAPERMS}\big) \quad \vee$
    $\texttt{grantNature}(a,\ uri,\ dp) = \textbf{NotGranted} \Rightarrow (a,\ uri,\ dp) \notin \text{GRANTED\_DATAPERMS}$
  - `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, $\forall a \in \text{APPS}$, $\forall uri \in \text{URI}$, $\forall dp_b \in \text{DATAPERMS}_\text{b}$, $\forall dp_m \in \text{DATAPERMS}_\text{m}$, $\forall dp = (dp_b,\ dp_m) \in \text{DATAPERMS}$. $\texttt{uriPrefixCheck}(\texttt{a},\ \texttt{uri},\ \texttt{dp}) = \mathbb{T} \Rightarrow \big(\texttt{dp}_\texttt{m} = \textbf{mprefix} \wedge \texttt{prefixMatch}(\texttt{a},\ \texttt{uri},\ \texttt{dp})\big)$

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 3.6.

- `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, $\forall a_2 \neq a_1 \in \text{APPS}$, $\forall uri \in \text{URI}$, $\forall dp \in \text{DATAPERMS}$.

  $\texttt{requestApproval}(a_2,\, a_1,\, uri) \neq \emptyset \;\Rightarrow \bigwedge_{dp \,\in\, \texttt{requestApproval}(a_1,\, a_1,\, uri)} \texttt{appAuthorized}(a_2,\, uri,\, dp)$

- `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, $\forall a_1 \neq a_2 \in \text{APPS}$, $\forall uri \in \text{URI}$, $\forall dp \in \text{DATAPERMS}$.

  $\texttt{grantApproval}(a_1,\, a_2,\, uri) = \mathbf{T} \;\Rightarrow \bigwedge_{dp \,\in\, \texttt{grantApproval}(\mathtt{a}_1,\, \mathtt{a}_2,\, \mathtt{uri})} \texttt{appAuthorized}(a_2,\, uri,\, dp)$

- `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, $\forall a \in \text{APPS}$, $\forall uri \in \text{URI}$, $\forall dp \in \text{DATAPERMS}$.

$$\texttt{appAuthorized}(\texttt{a},\ \texttt{uri},\ \texttt{dp}) \Rightarrow (\texttt{a},\ \textit{uri},\ \textit{dp}) \in \text{GRANTED\_DATAPERMS} \quad \vee$$
$$\texttt{ownerOf}\big(\texttt{belongingAuthority}(\textit{uri})\big) = a \quad \vee \exists p \in \texttt{grantedPerms}(a).\ (\textit{uri},\ \textit{dp}) \in$$
$$\texttt{corrDataPerm}(p) \quad \vee \quad \texttt{uriPrefixCheck}(\texttt{a},\ \texttt{uri},\ \texttt{dp})$$

Understanding the ACiA$_\alpha$model: We mention a few pointers that are required to be understood prior to reading the formalized ACiA$_\alpha$model.

1. 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.

2. 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.

3.2.3 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 3.7. All 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.

Table 3.7: ACiA$_\alpha$ User Initiated Operations

Operation: **AddApp**(ua : UAPPS)

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

$$\texttt{getAppSign}\big(\texttt{effApp}(up)\big) = \texttt{getAppSign}(ua) \quad \wedge$$

$$\texttt{getAuthorities}(ua) \cap \bigcup_{a \in \text{APPS}} \texttt{getAuthorities}(a) = \emptyset$$

Updates:

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

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

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

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

$$\text{PERMS\_DEF}' = \text{PERMS\_DEF} \quad \cup \bigcup_{up \in \texttt{getCusPerms}(ua)} \Big\{\big(ua,\ up, \texttt{defPgroupPerm}(ua,\ up),\ \texttt{getCusPermProtlvl}(ua,\ up)\big)\Big\}$$

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

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

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

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

$$\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \quad \cup$$

$$\bigcup_{\substack{a' \in \text{APPS}\\ up \in \texttt{wishList}(a') \cap \text{PERMS such that}\\ \texttt{effProtlvl}(up) = \textbf{normal}}} \{(a',\ up)\} \cup \bigcup_{\substack{a' \in \text{APPS};\ up \in \texttt{wishList}(a') \cap \text{PERMS such that}\\ \big(\texttt{effProtlvl}(up) = \textbf{signature} \wedge\\ \texttt{getAppSign}(\texttt{effApp}(up)) = \texttt{getAppSign}(a')\big)}} \{(a',\ up)\}$$

$$\text{DROLES\_WISHED}' = \text{DROLES\_WISHED} \quad \cup \bigcup_{\substack{a' \in \text{APPS};\ up \in \texttt{wishList}(a') \text{ such that}\\ \texttt{effProtlvl}(up) = \textbf{dangerous}}} \{(a',\ up)\}$$

Operation: **DeleteApp**(a : APPS)

Authorization Requirement: **T**

Updates:

$\text{COMP\_PROTECT}' = \text{COMP\_PROTECT} \quad \setminus$

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

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

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

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

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

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

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

$$\text{DROLES\_WISHED}' = \text{DROLES\_WISHED} \quad \setminus \left(\{a\} \times \texttt{wishDperms}(a) \quad \cup \bigcup_{a' \in \text{APPS} \setminus \{a\};\, p \in \texttt{effPerms}(a)} \{(a',\ p)\} \right)$$

$$\text{PERMS\_EFF}' = \left(\text{PERMS\_EFF} \quad \setminus \bigcup_{p \in \texttt{effPerms}(a)} \left\{ \left(a,\, p,\, \texttt{effPgroup}(p), \texttt{effProtlvl}(p) \right) \right\} \right) \quad \cup$$

$$\bigcup_{p \in \texttt{effPerms}(a')} \left\{ \left(a',\, p,\, \texttt{defPgroup}(a',\, p), \texttt{defProtlvl}(a',\, p) \right) \right\}, \text{ where } a' \in \texttt{defAppsPerms}(p) \setminus \{a\}$$

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

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

$$\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \quad \cup$$

$$\bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} \\ p \in \texttt{wishList}(a') \cap \text{PERMS s. t.} \\ \texttt{effProtlvl}(p) = \textbf{normal}}} \{(a',\ p)\} \quad \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\} p \in \texttt{wishList}(a') \cap \text{PERMS s. t.} \\ \Big(\texttt{effProtlvl}(p) = \textbf{signature} \ \wedge \\ \texttt{getAppSign}\Big(\texttt{effApp}(p)\Big) = \texttt{getAppSign}(a') \Big)}} \{(a',\ p)\}$$

$$\text{DROLES\_WISHED}' = \text{DROLES\_WISHED} \quad \cup \bigcup_{\substack{a' \in \text{APPS} \setminus \{a\};\, p \in \texttt{wishList}(a') \text{ such that} \\ \texttt{effProtlvl}(p) = \textbf{dangerous}}} \{(a',\ p)\}$$

$$\text{APPS}' = \text{APPS} \setminus \{a\}$$

Operation: **GrantDangerPerm**(a : APPS, p : PERMS)

Authorization Requirement: $p \in \texttt{wishDperms}(a)$

Update: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \cup \{(a, p)\}$

Operation: **GrantDangerPgroup**(a : APPS, pg : 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 : APPS, p : PERMS)

Authorization Requirements: $p \in \texttt{grantedPerms}(a) \ \wedge \ \texttt{p} \in \texttt{wishDperms}(a)$

Update: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \setminus \{(a,\ p)\}$

Operation: **RevokeDangerPgroup**(a : APPS, pg : PGROUP)

Authorization Requirements: $\exists p \in \texttt{grantedPerms}(a).\ \texttt{effPgroup}(p) = pg \ \wedge \ \texttt{p} \in \texttt{wishDperms}(a)$

Update: $\text{PERMS\_GRANTED}' = \text{PERMS\_GRANTED} \setminus \bigcup_{p \in \texttt{grantedPerms}(a) \text{ such that } \left(\texttt{effPgroup}(p) = pg \ \wedge p \in \texttt{wishDperms}(a)\right)} \{(a,\ p)\}$

– **GrantDangerPerm/GrantDangerPgroup**: These operations resemble the user granting a dangerous permission/permission-group to an app via the Settings app; and, the execution of these operations 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.

Table 3.8: ACiA_α Application Initiated Operations

Operation: **RequestPerm**$(a : \text{APPS},\ p : \text{PERMS})$

Authorization Requirement: $(a,p) \in \text{DROLES\_WISHED} \quad \wedge$

$\Big(\big(\exists p' \in \text{PERMS} \setminus \{p\}.\ \texttt{effPgroup}(p') = \texttt{effPgroup}(p) \wedge (a,p') \in \text{PERMS\_GRANTED} \big) \quad \vee$

$\texttt{userApproval}(a,p) \Big)$

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: $\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \quad \cup$

$$\bigcup_{dp \in \texttt{requestApproval}(a_{src},\ a_{tgt},\ uri)} \{(a_{src},\ uri,\ dp)\}$$

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

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 $\psi : \equiv \quad a_{src} = \texttt{ownerOf}\big(\texttt{belongingAuthority}(uri)\big) \quad \vee$

$\exists p \in \texttt{grantedPerms}(a_{src}).\ (uri,\ dp) \in \texttt{corrDataPerm}(p)$

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

Authorization Requirement: ψ

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

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:

$\texttt{ownerApp}(c_{src}) = \texttt{ownerApp}(c_{tgt}) \quad \vee$

$\Big(op \in \texttt{allowedOps}(c_{tgt}) \ \wedge \ \texttt{requiredPerm}(c_{tgt},\ op) \in \texttt{grantedPerms}\big(\texttt{ownerApp}(c_{src})\big) \quad \wedge$

$\big(op = \textbf{sendbroadcast} \ \wedge \ \texttt{brReceivePerm}(c_{src})\big) \ \Rightarrow$

$\texttt{brReceivePerm}(c_{src}) \subseteq \texttt{grantedPerms}\big(\texttt{ownerApp}(c_{tgt})\big) \Big)$

Update: -

Operation: **AppShutdown**(a : APPS)

Authorization Requirement: **T**

Updates: $\text{GRANTED\_DATAPERMS}' = \text{GRANTED\_DATAPERMS} \quad \setminus$

$$\bigcup_{\substack{(a,\ uri,\ dp) \in \text{GRANTED\_DATAPERMS such that} \\ \texttt{grantNature}(a,\ uri,\ dp) = \textbf{Temporary}}} \{(a,\ uri,\ dp)\}$$

3.2.4 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 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.

– **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.

3.2.5 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 [3, 7]. 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 of ACiA. Below we note a few such important observations and the related operations where they were encountered.

- **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 3.7), the PERMS_EFF updates in the **AddApp** and **DeleteApp** operations. While an app (App1) is being installed (see Figure 3.3), 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 defines the permission p1. Upon installing two additional apps (App2 & 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 un-installation 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: It is our belief, that Android should remember the order of app installations and modify permission definitions in-order rather than take a random approach to the same. This will enable developers to definitively know, which definition of a custom permission is active.

- **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

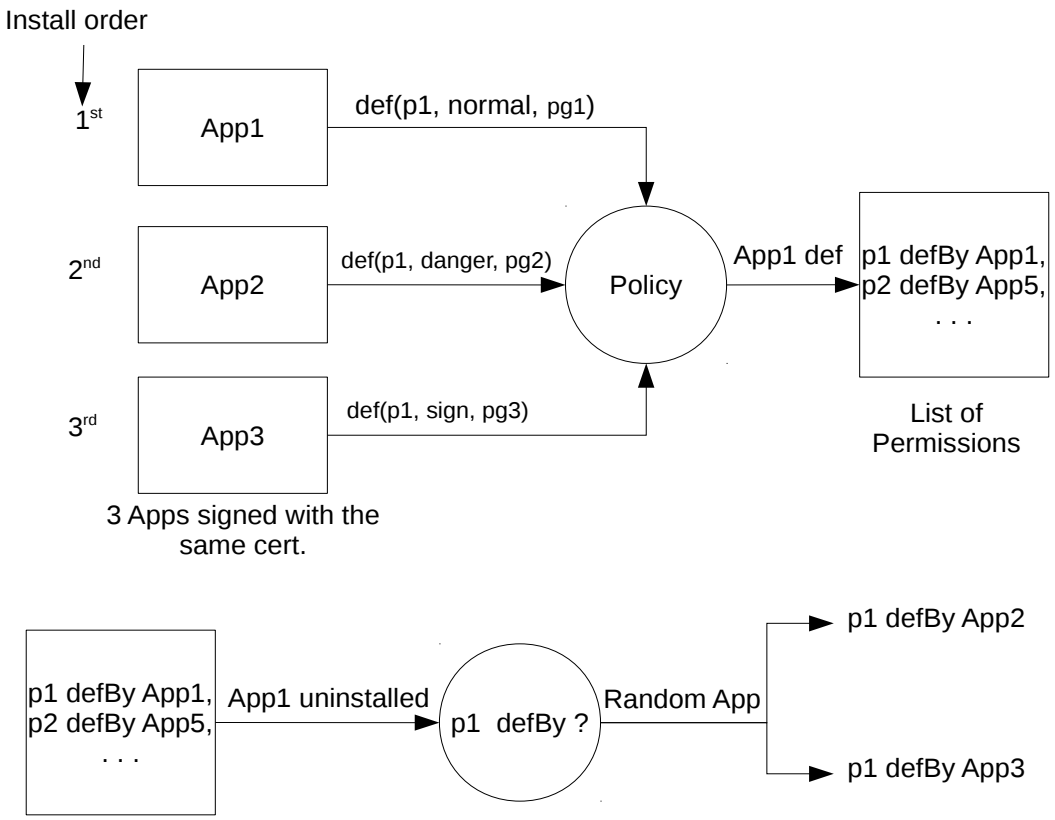


Figure 3.3: Anomaly in Android Custom Permissions

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 3.7), its definition is effective i.e.: protection-level of p1 is **normal**. If, at this juncture, App1 is un-installed, 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 [3].

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 its re-grants the **signature** permissions, so that, the behavior of permissions can be correctly predicted by developers.

- **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 [33].

  Explanation using ACiA$_\alpha$ model: It can be seen from Table 3.8, 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.

- **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

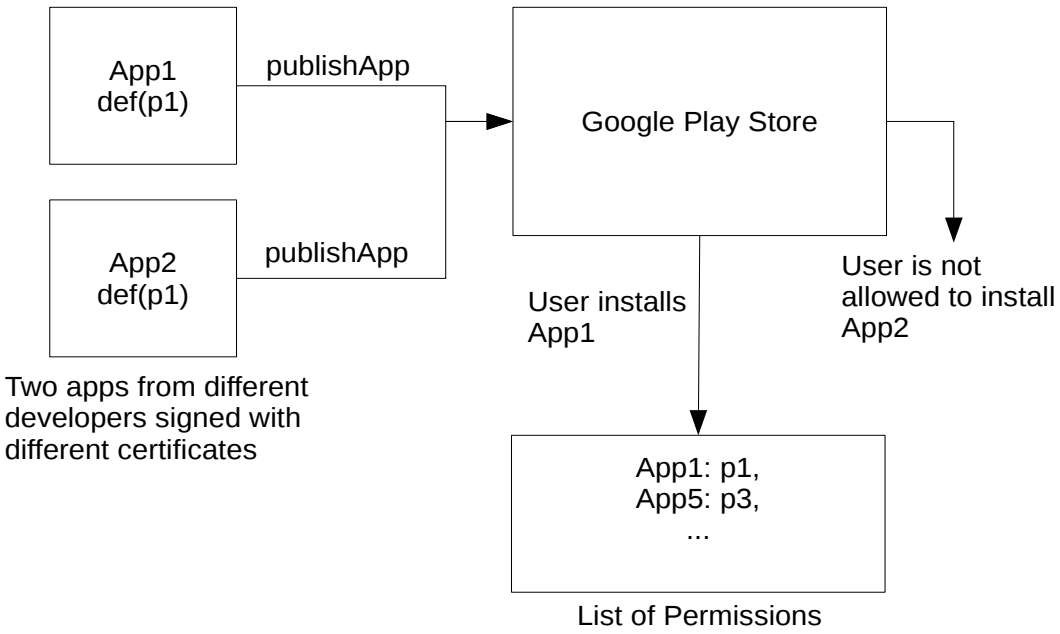


Figure 3.4: Drawback of Not Associating Custom Permission Names to App Signatures

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.

- **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 3.7, 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 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 & 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. Thus, it is crucial to model this complex operation of app un-installation to correctly predict any issues that may arise from inconsistencies in this operation.

CHAPTER 4: RBAC IN ANDROID

In this chapter, the role based access control models for Android are described. Role based access control (RBAC) [31] is used in enterprise scenarios due to the administrative ease it provides, and it would be useful in managing Android permissions as well. RBAC assigns objects to roles, and then enables administrators to grant these roles to the appropriate subjects. This results in a significant decrease in the administrative burden, because it is easier to grant a few roles rather than granting significantly higher number of permissions to the subjects. Apart from this, RBAC offers inherent advantages through sessions, which enable subjects to limit the number of permissions that are exposed at runtime.

4.1 RBAC Models for Android

In this section we define three new models for RBAC in Android, differing primarily in the substitutions for the set of users in RBAC with corresponding entities in Android. We have also provided a rigorous formal specification of these models below.

4.1.1 RiA_u (Users in RBAC Replaced with Users in Android)

In this model users in RBAC are substituted with users in Android. Permissions in Android grant a blanket access to the resource they protect, but it is intuitive to maintain the separation of data for each user. For example, access to user specific data such as contacts, photos, videos and calendar needs to be granted only to the user who owns that data; this requires perms and roles to be parameterized. Many works in the literature describe the concept of parameterized perms and roles that can achieve such a selective access control mechanism [34, 37, 45, 46]. As we can see from Fig.4.1, our model makes use of parameterized roles and perms to control selective access to resources, however, it should be noted that not all perms and roles are parameterized.

Design. According to this model, parameterized roles are granted directly to the users (UA). This UA can be done by the owner of the Android device or in an enterprise scenario, an admin-

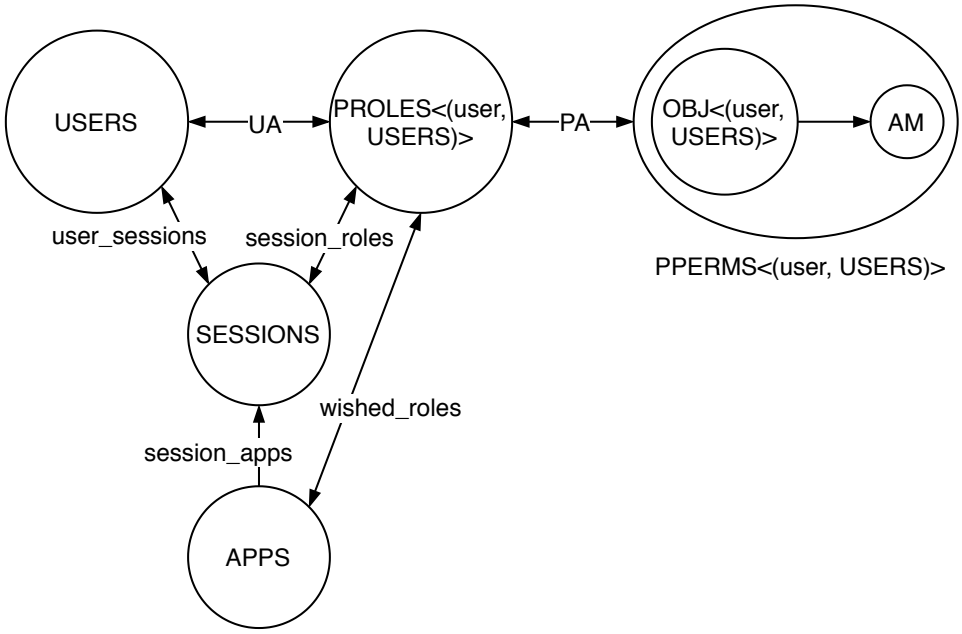


Figure 4.1: RBAC Users substituted with Android Users(RiA_u)

istrator for the device. Once users receive the required roles, they can launch new sessions when needed, and can activate any role they have been granted in that session. The user can control whether the new app launched should be added to an active session or to launch it in a different session (new session).

RiA_u *Model.* In this model for RiA_u, we define parameterized sets where P_{name} is the name of the parameter and P_{domain} is the domain of the parameter SET$\big\langle(P_{name},\ P_{domain})\big\rangle$. For all intents and purposes our model only contains the parameter of username, however this can be modified in the future to allow more granularity based on location, time and other parameters. The entity sets, relations and functions are described in Table 4.1, and the key operations for this model can be found in the Table 4.2.

Table 4.1: RiA_u Entity sets, Relations and Functions

Entity Sets
USERS and APPS the sets of all users and applications on an Android device.
OBJ$\langle$(user, USERS)$\rangle$, a set containing all the objects required to be distinct for the users of a given Android device. For example, – OBJ$\langle$(user, USERS)$\rangle$ = {Contacts$\langle$(user, Alice)$\rangle$, Contacts$\langle$(user, Bob)$\rangle$, WhatsAppPhotos$\langle$(user, Alice)$\rangle$, WhatsAppPhotos$\langle$(user, Bob)$\rangle$,...}
AM = {read, write}, a set of access modes for all objects on a given Android device.
PROLES$\langle$(user, USERS)$\rangle$, the set of all parameterized roles in a given Android device. A few example parameterized roles are given below. – PROLES$\langle$(user, Alice)$\rangle$ = {Parent$\langle$(user, Alice)$\rangle$, Guest$\langle$(user, Alice)$\rangle$, Child$\langle$(user, Alice)$\rangle$} – PROLES$\langle$(user, Bob)$\rangle$ = {Parent$\langle$(user, Bob)$\rangle$, Guest$\langle$(user, Bob)$\rangle$, Child$\langle$(user, Bob)$\rangle$}
SESSIONS, the set of all sessions on an Android device.
Relations
PPERMS$\langle$(user, USERS)$\rangle$ = $2^{\mathrm{OBJ}\langle\text{(user, USERS)}\rangle \times \mathrm{AM}}$, a set of all parameterized permissions on a given Android device. For example, – PPERMS$\langle$(user, Alice)$\rangle$ = {(WhatsAppPhotos$\langle$(user, Alice)$\rangle$, read) (WhatsAppPhotos$\langle$(user, Alice)$\rangle$, write), (Contacts$\langle$(user, Alice)$\rangle$, read), (Contacts$\langle$(user, Alice)$\rangle$, write)}
UA $\subseteq$ USERS $\times$ PROLES$\langle$(user, USERS)$\rangle$, a many-to-many mapping user-to-parameterized role assignment relation.
PA $\subseteq$ PROLES$\langle$(user, USERS)$\rangle$ $\times$ PPERMS$\langle$(user, USERS)$\rangle$, a many-to-many mapping permission-to-role assignment relation.
Functions

`user_proles`: USERS $\rightarrow 2^{\text{PROLES(USERS)}}$, the mapping of a user u:USERS onto a set of parameterized roles assigned to that user. Formally, `user_proles`(u) = $\{pr\langle(\text{user}, u)\rangle \in \text{PROLES}\langle(\text{user}, u)\rangle \mid (u, pr\langle(\text{user}, u\rangle)) \in \text{UA}\}$

`assigned_users`: $\text{PROLES}\langle(\text{user}, \text{USERS})\rangle \rightarrow 2^{\text{USERS}}$, the mapping of a parameterized role $\text{pr}\langle(\text{user}, \text{u})\rangle$:$\text{PROLES}\langle(\text{user}, \text{USERS})\rangle$ onto a set of users that it has been assigned to. Formally, `assigned_users`$(pr\langle(\text{user}, u\rangle))|_{u \in \text{USERS}} = \{u \mid (u, pr\langle(\text{user}, u)\rangle) \in \text{UA}\}$

`assignedPerms`: $\text{PROLES}\langle(\text{user}, \text{USERS})\rangle \rightarrow 2^{\text{PPERMS}\langle(\text{user, USERS})\rangle}$, the mapping of $\text{pr}\langle(\text{user}, \text{u})\rangle$:$\text{PROLES}\langle(\text{user}, \text{USERS})\rangle$ onto a set of parameteried permissions for a particular user u:USERS. Formally, `assignedPerms`$(pr\langle(\text{user}, u)\rangle) = \{pp\langle(\text{user}, u)\rangle \in \text{PPERMS}\langle(\text{user}, \text{USERS})\rangle \mid (pp\langle(\text{user}, u)\rangle, pr\langle(\text{user}, u)\rangle) \in \text{PA}\}$

`user_sessions`: USERS $\rightarrow 2^{\text{SESSIONS}}$, the mapping of user u onto a set of sessions. Note that, $\forall u_1 \neq u_2 \in \text{USERS}.$ `user_sessions`$(u_1) \cap$ `user_sessions`$(u_2) = \emptyset$

`session_apps`: SESSIONS $\rightarrow 2^{\text{APPS}}$, the mapping of session s onto a set of applications.

`session_proles`: $\text{SESSIONS} \rightarrow 2^{\text{PROLES}\langle(\text{user, USERS})\rangle}$, the mapping of session s onto a set of roles. Formally, `session_proles`$(s_i) = \{pr\langle(\text{user}, u)\rangle \in \text{PROLES}\langle(\text{user}, u)\rangle \mid u \in \text{USERS} \wedge (\texttt{session\_users}(s_i), pr\langle(\text{user}, u)\rangle) \in \text{UA}\}$. Note that, $\forall s_1 \neq s_2 \in \text{SESSIONS}.$ `session_proles`$(\text{s}_1) \neq$ `session_proles`(s_2)

`availSessPerms`: $\text{SESSIONS} \rightarrow 2^{\text{PERMS}\langle(\text{user, USERS})\rangle}$, the permissions available to a user in a session, $\bigcup_{pr\langle(\text{user}, u)\rangle \in SESSION\_ROLES(s)}$ `assignedPerms`$(pr\langle(\texttt{user}, u)\rangle)$.

`wished_proles`: $\text{APPS} \rightarrow 2^{\text{PROLES}\langle(\text{user, USERS})\rangle}$, the mapping of an app a:APPS onto a set of roles wished by that app.

Table 4.2: RiA_u Operations

Operation: **CreateSession**(u : USERS, a : APPS, $ars : 2^{\mathrm{PROLES}\langle(\text{user},\ u)\rangle}$, s : NAME)

Authorization Requirement: $ars \subseteq \texttt{user\_proles}(u) \wedge ars \subseteq \texttt{wished\_proles}(a) \wedge s \notin \text{SESSIONS}$

Updates:

$\text{SESSIONS}' = \text{SESSIONS} \cup \{s\}$

$\texttt{user\_sessions}' = \texttt{user\_sessions} \cup \{u,\ s\}$

$\text{SESSION\_ROLES}' = \text{SESSION\_ROLES} \cup \{s\} \times ars$

Operation: **DeleteSession**(u : USERS, s : SESSIONS)

Authorization Requirement: $\texttt{sessUser}(s) = u$

Updates:

$\texttt{user\_sessions}' = \texttt{user\_sessions} \setminus \{u,\ s\}$

$\text{SESSION\_ROLES}' = \text{SESSION\_ROLES} \cup \{s\} \times ars$

$\text{SESSIONS}' = \text{SESSIONS} \setminus \{s\}$

Operation: **RequestAccess**$\big(a : \text{APPS},\ u : \text{USERS},\ pr\langle(\text{user},\ u)\rangle : \text{PROLES}\langle(\text{user},\ u)\rangle\big)$

Authorization Requirement: $\exists s \in \text{SESSIONS}.\ a \in \texttt{session\_apps}(s) \wedge u = \texttt{session\_users}(s) \wedge \big(u,\ pr\langle(\text{user},\ u)\rangle\big) \in \text{UA} \wedge \texttt{pr}\langle(\text{user},\ \texttt{u})\rangle \notin \texttt{SESSION\_ROLES}(\texttt{s})$

Updates: $\text{SESSION\_ROLES}' = \text{SESSION\_ROLES} \cup \{s,\ pr\langle(\text{user},\ u)\rangle\}$

Operation: **RevokeRole**$\big(u :\ \text{USERS},\ s : \text{SESSIONS},\ pr\langle(\text{user},\ u)\rangle : \text{PROLES}\langle(\text{user},\ u)\rangle\big)$

Authorization Requirement: $s \in \texttt{user\_sessions}(u)$

Updates: $\text{SESSION\_ROLES}' = \text{SESSION\_ROLES} \setminus \{s,\ pr\langle(\text{user},\ u)\rangle\}$

Operation: **CheckAccess**(a : APPS, s : SESSIONS, $pp\langle(\text{user},\ u)\rangle : \text{PPERMS}\langle(\text{user},\ u)\rangle$, $outresult$: BOOLEAN)

Authorization Requirement: $\exists pr\langle(\text{user},\ u)\rangle \in \text{PROLES}\langle(\text{user},\ u)\rangle.\ u = \texttt{session\_users}(s) \wedge pr\langle(\text{user},\ u)\rangle \in SESSION\_ROLES(s) \wedge \big(pr\langle(\text{user},\ u)\rangle,\ pp\langle(\text{user},\ u)\rangle\big) \in \text{PA}$

Updates: -

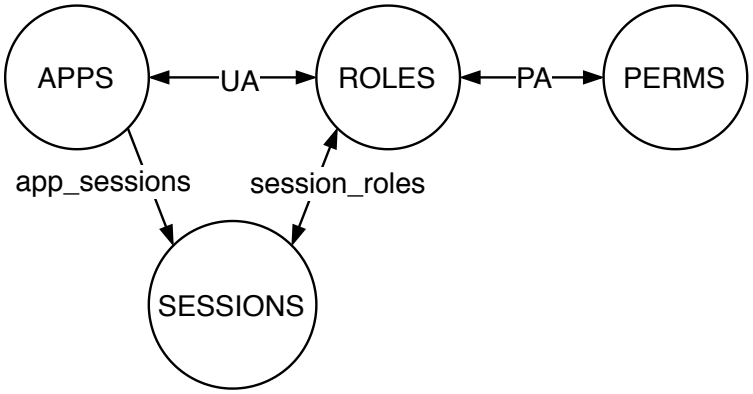


Figure 4.2: RBAC Users substituted with Android Applications(RiA_a)

4.1.2 RiA_a (Users in RBAC Replaced with Applications in Android)

In this model, users in RBAC are substituted with applications in Android (see Fig.4.2). This model dictates entrusting decision policies such as what roles to activate and when, with the applications themselves, and by extension with the app developers. In the current Android, prompts are only shown for permission groups and within these groups, only the permissions belonging to dangerous protection level are controlled with the groups themselves. By assigning roles to applications, the administrative burden of users is reduced without a disproportionate increase in the number of user prompts[1].

Design. Since applications are tasked with managing active roles, app developers are responsible for defining, requesting and activating roles. A few default roles are built into the devices for use by applications. These roles are generated by top-down (semantic meaning) and bottom-up (algorithms for mining roles) approaches [21] using applications from the Play store and the information on which permissions are requested by them. Although applications are designated as subjects, keeping in line with the requirement to safeguard user data, role requests still need to be granted by users. Once roles are granted, applications are free to activate any role as required and can be set to launch in a pre-active session or launch distinct sessions as per developer discretion.

[1]This depends on the quality of the roles that exist in the system.

Table 4.3: RiA_a Entity sets, Relations and Functions

Entity Sets
APPS, ROLES and PERMS the set of all apps, roles and permissions on a given Android device.
SESSIONS, the set of all sessions on an Android device.
Relations
UA $\subseteq$ APPS $\times$ ROLES, a many-to-many mapping app-to-role assignment relation.
PA $\subseteq$ PERMS $\times$ ROLES, a many-to-many mapping perm-to-role assignment relation.
Functions
`assigned_apps`: ROLES $\rightarrow 2^{\text{APPS}}$, the mapping of role r:ROLES onto a set of apps. Formally: `assigned_apps`(r) = $\{a \in \text{APPS} \mid (a, r) \in \text{UA}\}$.
`app_roles`: APPS $\rightarrow 2^{\text{ROLES}}$, the mapping of app a:APPS onto a set of roles assigned to it. Formally: `app_roles`(a) = $\{r \in \text{ROLES} \mid (a, r) \in \text{UA}\}$.
`assigned_permissions`: ROLES $\rightarrow 2^{\text{PERMS}}$, the mapping of role r:ROLES onto a set of permissions. Formally: `assigned_permissions`(r) = $\{p \in \text{PERMS} \mid (p, r) \in \text{PA}\}$.
`app_sessions`: APPS $\rightarrow 2^{\text{SESSIONS}}$, the mapping of app a:APPS onto a set of sessions.
`session_roles`: SESSIONS $\rightarrow 2^{\text{ROLES}}$, the mapping of session s:SESSIONS onto a set of roles. Formally: `session_roles`(s_i) $\subseteq \{r \in \text{ROLES} \mid (\texttt{session\_apps}(s_i), r) \in \text{UA}\}$.
`avail_session_perms`: SESSIONS $\rightarrow 2^{\text{PERMS}}$, the permissions available to an app in a session, $\bigcup_{r \in \texttt{session\_roles}(s)} \texttt{assigned\_permissions}(r)$.
`wished_roles`: APPS $\rightarrow 2^{\text{ROLES}}$, the mapping of an app a:APPS onto a set of roles wished by that app.

RiA_a *Model.* The model for RiA_a is described below along with the entity sets, relations and functions in Table 4.3, and, its key operations in Table 4.4.

Table 4.4: RiA_a Operations

Operation: **CreateSession**$(a : \text{APPS},\ ars : 2^{\text{ROLES}}, s : \text{NAME})$
Authorization Requirement: $ars \subseteq \texttt{app\_roles}(a) \wedge ars \subseteq \texttt{wished\_roles}(a) \wedge s \notin \text{SESSIONS}$
Updates:
$\text{SESSIONS}' = \text{SESSIONS} \cup \{s\}$
$\texttt{app\_sessions}' = \texttt{app\_sessions} \cup \{(a,\ s)\}$
$\texttt{session\_roles}' = \texttt{session\_roles} \cup \{s\} \times \text{ARS}$
Operation: **DeleteSession**$(a : \text{APPS},\ s : \text{SESSIONS})$
Authorization Requirement: $s \in \texttt{app\_sessions}(a)$
Updates:
$\texttt{app\_sessions}' = \texttt{app\_sessions} \setminus \{a,\ s\}$
$\texttt{session\_roles}' = \texttt{session\_roles} \cup \{s\} \times ars$
$\text{SESSIONS}' = \text{SESSIONS} \setminus \{s\}$
Operation: **RequestAccess**$(a : \text{APPS},\ r : \text{ROLES})$
Authorization Requirement: $\exists s \in \text{SESSIONS}.\ a \in \texttt{session\_apps}(s) \wedge (a,\ r) \in \text{UA} \wedge r \notin \texttt{session\_roles}(s)$
Updates: $\texttt{session\_roles}' = \texttt{session\_roles} \cup \{s,\ r\}$
Operation: **RevokeRole**$(s : \text{SESSIONS},\ r : \text{ROLES})$
Authorization Requirement: $s \in \texttt{app\_sessions}(a)$
Updates: $\texttt{session\_roles}' = \texttt{session\_roles} \setminus \{s,\ r\}$
Operation: **CheckAccess**$(a : \text{APPS},\ s : \text{SESSIONS},\ p : \text{PERMS},\ outresult : \text{BOOLEAN})$
Authorization Requirement: $\exists r \in \text{ROLES}.\ a = \texttt{session\_apps}(s) \wedge r \in \texttt{session\_roles}(s) \wedge (\texttt{r},\ \texttt{p}) \in \text{PA}$
Updates: -

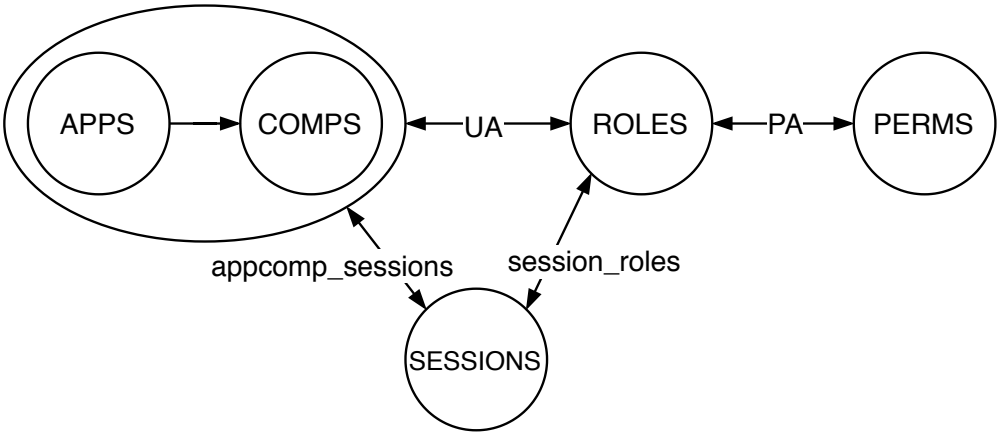


Figure 4.3: RBAC Users substituted with Android App-components (RiA_{ac})

4.1.3 RiA_{ac} (Users in RBAC Replaced with App-components in Android)

In this model, users in RBAC are replaced with app-components in Android (see Fig.4.3). This model supports a highly granular access control system, by assigning roles directly to app-components. This also limits the exposure of sensitive system and user resources.

Design. Since roles are granted to app-components, the roles need to be defined by the applications. This puts the onus of defining, requesting and activating roles with the app developers. Users would be required to accept role prompts prior to them being granted to the app-components. Although this model presents an additional burden on the app developers and users, due to increased granularity associated with granting roles directly to app-components, it is heavily mitigated by the use of the RBAC system. Also, this model mitigates the issue of third party app components receiving the entire battery of permissions assigned to the app themselves, because roles are granted to app-components themselves. On launch, app-components can be programmed to activate any of the granted roles to obtain the required permissions for providing full functionality.

Table 4.5: RiA_{ac} Element sets, Relations and Functions

Entity Sets
APPS, COMPS (the sets of all apps and components on an Android device)
ROLES the set of all roles on an Android device
PERMS, the set of all permissions that exist on a given Android device.
SESSIONS, the set of all sessions that exist on an Android device.
Relations
APP_COMPS $\subseteq$ APPS $\times$ COMPS, a one-to-many mapping applications-to-component assignment relation.
UA $\subseteq$ APP_COMPS $\times$ ROLES, a many-to-many mapping app_components-to-role assignment relation.
PRA $\subseteq$ PERMS $\times$ ROLES, a many-to-many mapping permission-to-role assignment relation.
Functions
`assignedComps`: ROLES $\nrightarrow 2^{\mathrm{APP\_COMPS}}$, the mapping of role *r*:ROLES onto a set of app_components. Formally, `assignedComps`(*r*) = {*ac* $\in$ APP_COMPS $\mid$ (*ac, r*) $\in$ UA}.
`appcomp_roles`: APP_COMPS $\rightarrow 2^{\mathrm{ROLES}}$, the mapping of app-comp *ac*:APP_COMPS onto a set of roles. Formally, `appcomp_roles`(*ac*) = { *r* $\in$ ROLES $\mid$ (*ac, r*) $\in$ UA}.
`assigned_permissions`:ROLES $\rightarrow 2^{\mathrm{PERMS}}$, the mapping of *r*:ROLES onto a set of permissions. Formally, `assigned_permissions`(*r*) = {*p* $\in$ PERMS $\mid$ (*p, r*) $\in$ PRA}.
APPCOMP_SESSIONS $\subseteq$ APP_COMPS $\times$ SESSIONS, a many-to-many mapping app_components - to - session assignment relation.
`sessAC`: APP_COMPS $\nrightarrow 2^{\mathrm{SESSIONS}}$, the mapping of *ac*:APP_COMPS onto a set of sessions. Formally, `sessAC`(*ac*) = {*s* $\in$ SESSIONS $\mid$ (*ac, s*) $\in$ APPCOMP_SESSIONS}.
`appCompsS`: SESSIONS $\rightarrow$ APP_COMPS, the mapping of *s*:SESSIONS to an app_component *ac*. Note that, $\forall s \in$ SESSIONS. (`appCompsS`$(s),\ s) \in$ COMP_SESSIONS.
`session_roles` $\subseteq$ SESSIONS $\times$ ROLES, a many-to-many mapping session-to-roles assignment relation.

sessRoles: SESSIONS $\rightarrow 2^{\text{ROLES}}$, the mapping of s:SESSIONS onto a set of roles. Formally, sessRoles(s) = $\{r \in \text{ROLES} \mid (\text{appCompsS}(s), r) \in \text{UA}\}$.
avail_session_perms: SESSIONS $\twoheadrightarrow 2^{\text{PERMS}}$, the permissions available to a component in a session, $\bigcup_{r \in \text{sessRoles}(s)} \text{assigned\_permissions}(r)$.
isAuthorized: SESSIONS $\times$ PERMS $\rightarrow \mathbb{B}$, a s:SESSIONS is authorized to exercise a permission p if isAuthorized(s, p). Also, $\forall s \in \text{SESSIONS}$, $\forall p \in \text{PERMS}$. $\text{isAuthorized}(s, p) \rightarrow \exists r \in \text{ROLES}. ((s, r) \in \text{session\_roles} \wedge (p, r) \in \text{PRA})$
wished_roles: APP_COMPS $\rightarrow 2^{\text{ROLES}}$, the mapping of an app-comp ac:APP_COMPS onto a set of roles wished by that app component.

RiA_{ac} *Model.* In this subsection, we define a model for RBAC in Android with users in RBAC substituted with components in Android. The entity sets, relations and functions are described in Table 4.5, and the key operations for this model can be found in Table 4.6.

4.2 Adapting RBAC into the Android OS

In an enterprise, the IT employees can meet with the department heads to create roles based on job functions. However, in Android, there is no equivalent entity to job functions; neither the applications are designed with a single task in mind, nor do the permissions dictate the manner in which they should be used i.e.: a location permission maybe utilized by applications to show the device location on the map, send the location information to another person or to alert the user by using the geo-fencing capabilities. In order for RBAC to work in Android, it is required to build the two main assignment relations, namely the UA or the user assignment and the PA or the Permission assignment. After several attempts to build a PA based on semantically related permissions, we resorted to role-mining, a technique with which PA can be automatically generated utilizing algorithms. Several such algorithms are discussed in this section.

Prior to the implementation of RBAC in Android, it is required to define roles, and asso-

Table 4.6: RiA_{ac} Operations

Operation: **CreateSession**(ac : APP_COMPS, $ars : 2^{\mathrm{ROLES}}$, s : NAME)

Authorization Requirement: $ars \subseteq \texttt{appcomp\_roles}(ac) \ \wedge \ ars \subseteq \texttt{wished\_roles}(ac)$

Updates:

$\mathrm{SESSIONS}' = \mathrm{SESSIONS} \cup \{s\}$

$\mathrm{APPCOMP\_SESSIONS}' = \mathrm{APPCOMP\_SESSIONS} \cup \{(ac,\ s)\}$

$\texttt{session\_roles}' = \texttt{session\_roles} \cup \bigcup_{r \in ars} \{(s,\ r)\}$

Operation: **DeleteSession**(ac : APP_COMPS, s : SESSIONS)

Authorization Requirement: *ac* = $\texttt{appCompsS}(s)$

Updates:

$\mathrm{APPCOMP\_SESSIONS}' = \mathrm{APPCOMP\_SESSIONS} \setminus \{(ac,\ s)\}$

$\texttt{session\_roles}' = \texttt{session\_roles} \setminus \bigcup_{r \in \texttt{session\_roles}(s)} \{(s,\ r)\}$

$\mathrm{SESSIONS}' = \mathrm{SESSIONS} \setminus \{s\}$

Operation: **RequestAccess**(ac : APP_COMPS, s : SESSIONS, r : ROLES)

Authorization Requirement: *ac* = $\texttt{appCompsS}(s) \wedge$

$(ac,\ r) \in \mathrm{UA} \wedge \forall dsdpair \in \mathrm{DSD}.\ dsdpair = (rs_1,\ n),$

$\forall rset \in 2^{\mathrm{ROLES}}.\ rset \subseteq rs_1 \wedge\ rset \subseteq ars \Rightarrow |rset| < n$

Updates: $\texttt{session\_roles}' = \texttt{session\_roles} \cup \{(s,\ r)\}$

Operation: **RevokeRole**(ac : APP_COMPS, s : SESSIONS, r : ROLES)

Authorization Requirement: *ac* = $\texttt{appCompsS}(s) \wedge r \in \texttt{sessRoles}(s)$

Updates: $\texttt{session\_roles}' = \texttt{session\_roles} \setminus \{(s,\ r)\}$

Operation: **CheckAccess**(ac : APP_COMPS, s : SESSIONS, p : PERMS,

$outresult$: BOOLEAN)

Authorization Requirement: $\exists r \in \mathrm{ROLES}.\ ac = \texttt{appCompsS}(s) \wedge r \in \texttt{session\_roles}(s) \wedge$

$(r,\ p) \in \mathrm{PA}$

Updates: -

ciate them with certain permissions to form the Permission assignment relation (PA). Multiple approaches exist for achieving this, some rely on prior knowledge of the subject job functions in the system, whereas, some automate the process via algorithms that produce such a mapping (i.e.: the mapping between roles and permissions). Mechanisms that use knowledge of job functions to manufacture roles belong to the group known as top-down approaches, whereas those using algorithms to automate the process are known as bottom-up approaches; all mechanisms produce a set of roles, and the mapping PA.

Role mining is a bottom up approach [72] of role engineering [21] in which, algorithms are used to analyze and extract roles from a pre-existing user permission assignment (UPA) matrix. Various algorithms to mine roles from provided data sets have been published, and we have analyzed and implemented five such algorithms i.e.: Fast Miner/ Complete Miner [72], Basic RMP [69], Delta RMP [71] and the Min Noise RMP algorithm [40]. The algorithms and the resultant generated roles are discussed briefly below.

4.2.1 Criteria for Role-Mining Algorithm Selection

There are two types of algorithms that are considered in this work; one, that outputs a large set of candidate roles, from which, useful roles need to be extracted for building the PA; and second, that outputs the UA and PA. Before proceeding further, the principles that outline the quality requirements of the element sets and relations ensuring that RBAC in Android is advantageous, are stated below; these principles guide the selection of appropriate role-mining algorithm.

Principle 1 (Cardinality limit on ROLES)*: The number of roles should ideally be significantly lower than the number of requested-permissions from the UPA.*

$$|\text{ROLES}| \ll |\text{RPERMS}|$$

Explanation: Roles are used to ease the administration of a large set of permissions with applications (in Android there are 161 requested-permissions and modern Android devices can have hundreds of applications); mining an equally large set of roles undermines the very basic motive

of having RBAC in Android, which is to provide a level of abstraction between applications and permissions. For practical purposes, the number of roles should not be higher than 3/4th of the number of requested-permissions

$$\text{Practically: } |\text{ROLES}| \leq (0.75)|\text{RPERMS}|$$

Principle 2 (Coverage of all requested-permissions)*: All the requested-permissions from the UPA must be assigned to at least one role,*

$$|R_1 \cup R_2 \cup \ldots R_n| = |RPERMS|$$

where R1, R2, Rn are the roles generated by the algorithm.

Explanation: This principle specifies that all the permissions that are ever requested by any application, must be covered by ROLES (assigned to at least 1 role).

Android dataset. We downloaded top 500 free applications from the Google Play Store & analyzed each for its requested-permissions to acquire the UPA (using AAPT - Android Asset Packaging Tool [42]). The cardinality of the set of permissions used by at-least 1 application was found to be 161, this means out of 500+ permissions found on the built-in Android emulator, only 161 were ever requested. The role-mining algorithms that were implemented from their pseudo-code to generate UA and PA are described below.

4.2.2 Fast-Miner and Complete-Miner algorithm

The Fast-Miner and Complete-Miner [72] are algorithms, that generate a large set of candidate roles (CR) from which, useful roles need to be extracted to build the PA. The Fast-Miner restricts the intersections between roles to a maximum of 2, and its output consists of more than 14,000 roles. The Complete-Miner has no such restriction, and so, it takes a significant amount of time to execute generating over 200,000 roles. However, due to the high time complexity in running the Complete-Miner, this algorithm is un-suitable for use in generating roles for our purposes. The Fast-Miner on the other hand executes fully within a reasonable amount of time, and, we use its output to extract roles for building the PA. In order to facilitate this extraction, the roles in CR

need to be sorted using a metric, that prioritizes roles according to their characteristics, so that, the resulting PA satisfies the aforementioned principles. Such a prioritization is achieved with the help of formulas that are stated below.

Prioritization Formula 1 (RolePriority)*: The roles in CR are prioritized using a metric called RolePriority which is defined as,*

$$RolePriority = IRCount \times PriorityConstant + Count$$

where IRCount is associated with the roles in IR, Count is associated with the roles in GR, and PriorityConstant is a constant that biases the RolePriority towards IR.

Prioritization Formula 2 (Subset Commonality Factor)*: The Subset Commonality Factor for a role indicates how frequently it is subsumed by other roles in CR, and is defined as,*

$$SCF_{r_i} = \frac{|\{r_n \mid \forall r_n \subseteq CR.\ r_i \subseteq r_n\}|}{|CR|}$$

The SCF is calculated for individual roles and is obtained by dividing the number of roles that subsume the role in consideration, with the total number of roles.

Prioritization Formula 3 (Usefulness-Factor of a Role)*: The Usefulness-Factor of a role with respect to CR is given by,*

$$UF(r_i,\ CR) = \frac{\sum_{j\ =\ 1}^{n} \frac{|r_i \cap r_j|}{r_i \cup r_j}}{n}$$

where n is the number of roles in CR.

The above prioritization formulas were applied to the output of FM/CM algorithm, however, they failed to satisfy either one of the two requirements stated above. Hence the output of FM/CM algorithms cannot directly be used to construct meaningful roles in Android.

4.2.3 Basic-RMP algorithm

The Basic-RMP algorithm [69] is adapted from the largest uncovered tile mining algorithm (LUTM) [35] defined for databases, for use in role-mining; it greedily discovers large, uncovered tiles from the UPA, to construct the UA and PA. We implemented this algorithm and used it to generate UA

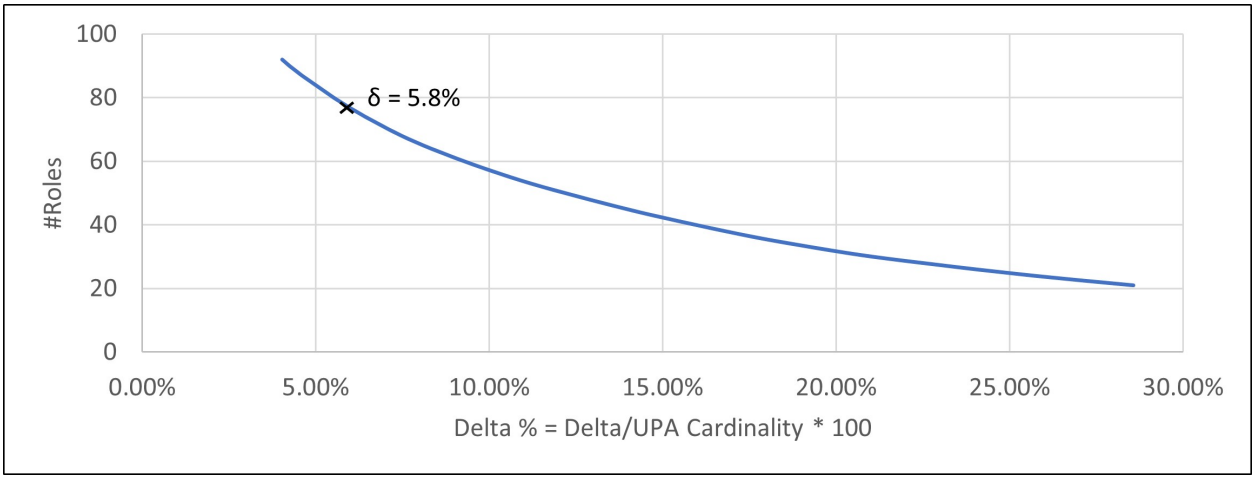


Figure 4.4: Delta % to number of Roles

and PA for Android using the UPA extracted from applications. The output of prioritized roles (see Appendix A) from this algorithm contains many roles having 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 role, the tiles considered by the algorithm are contiguous and do not provide optimal area with respect to the entire UPA. The single permission roles generated by this algorithm do not satisfy either of our principles and therefore the output of this algorithm cannot directly be considered for generating the PA or the UA.

4.2.4 δ Approx-RMP algorithm

The δ-approximation algorithm [70] generates PA by using the candidate role set generated by the Fast Miner algorithm as input, and, greedily picking the best candidate role 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 roles. Delta is the approximation factor between the UPA and the reconstructed UPA (from PA and UA), and, can be adjusted to change the properties of the generated PA, and, is inversely proportional to the cardinality of the set of ROLES generated by the algorithm.

Approximation factor δ

$$0 < \delta < |UPA|$$

$$\delta \propto \frac{1}{|ROLES|}$$

If the δ used is too high, the algorithm finishes prematurely before any significant number of roles are generated, whereas, a δ that is too low will increase the time complexity of the algorithm. From the results 4.4, we found that having the value of δ to be 5.8% to be optimal, as this would ensure a comparatively lower |ROLES| compared with |PERMS| with good coverage of permissions and a small under-privilege. Once the PA is generated, we can use a secondary algorithm to obtain UA by assigning roles to applications if the ratio of matching permission between application and a role crosses a certain threshold.

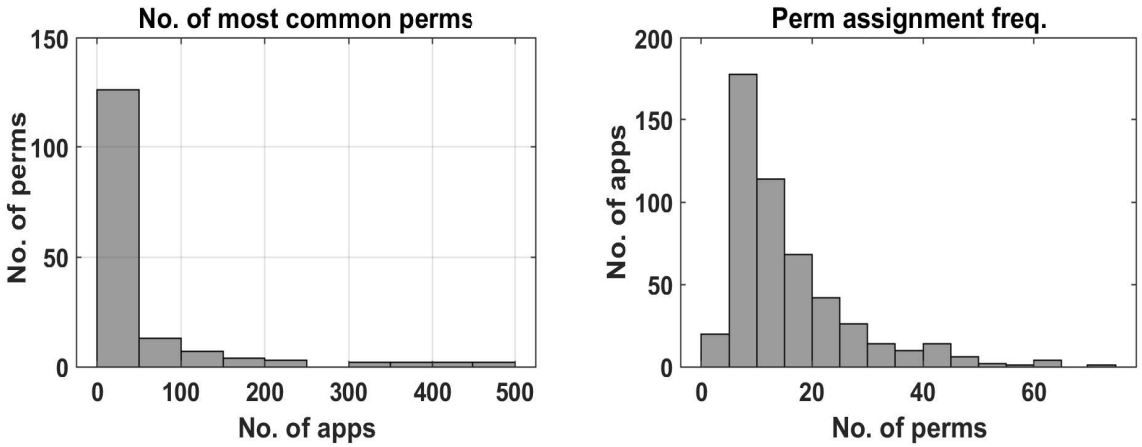
4.2.5 Min-Noise RMP Algorithm

The Min-Noise RMP algorithm works similarly to the δ-approximation RMP algorithm with one crucial difference - instead of minimizing the difference between the output matrices and the input matrix, the number of roles are fixed, and then the PA is generated, based on this number. This algorithm performs better than the δ-approximation RMP algorithm for generating roles for Android, since it outputs the best possible PA, given the number of roles. A few sample roles generated by this algorithm are shown in Appendix A Table A.4.

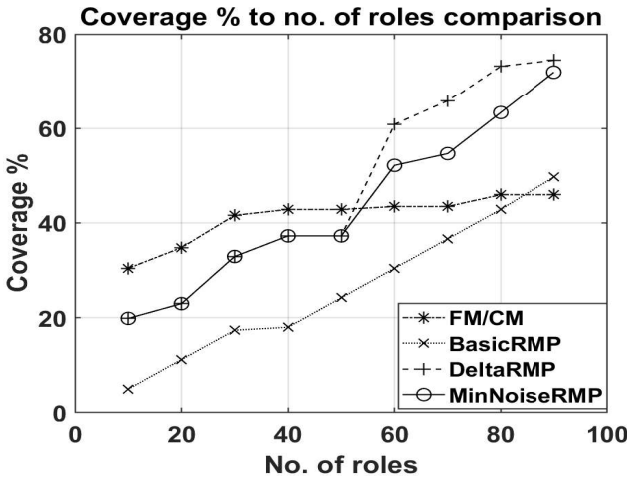
4.3 Selecting the RMP Algorithm for Role-Mining in Android

It should be noted that the issue of engineering good roles has been studied extensively, and is outside the scope of this work; we merely present a few techniques from the literature to automatically generate roles based on the UPA matrix from our data set. This does not confer a finality to this work, and roles need to be engineered according to the requirements for each different system.

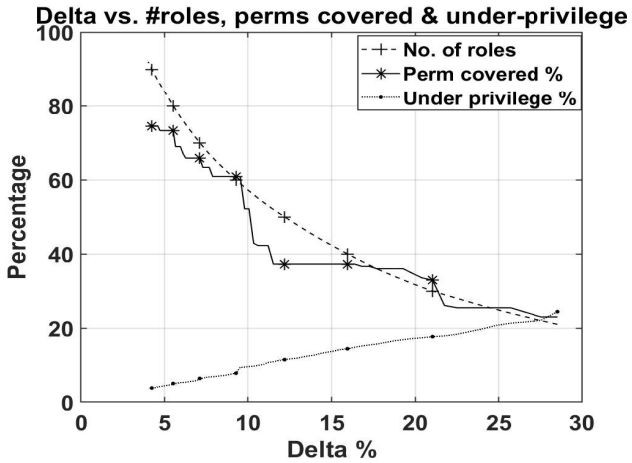
The purpose of mining roles, is to provide a few basic roles which can be pre-included with the RBAC in Android system, and it is not our intention to assign all of Android's perms (582 perms in API 29) to roles. It is shown below, that even when we consider as low as 10 roles, a significant number of perms requested by apps are covered to them. The onus of assigning the remaining perms to roles, and requesting these roles from the user lies with the app developers.



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

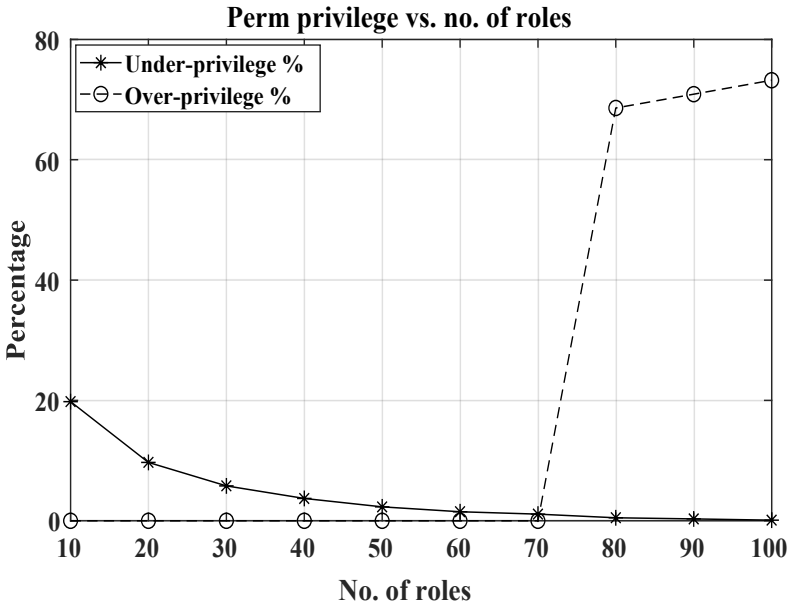


(b) Cvrg. of role mining alg.

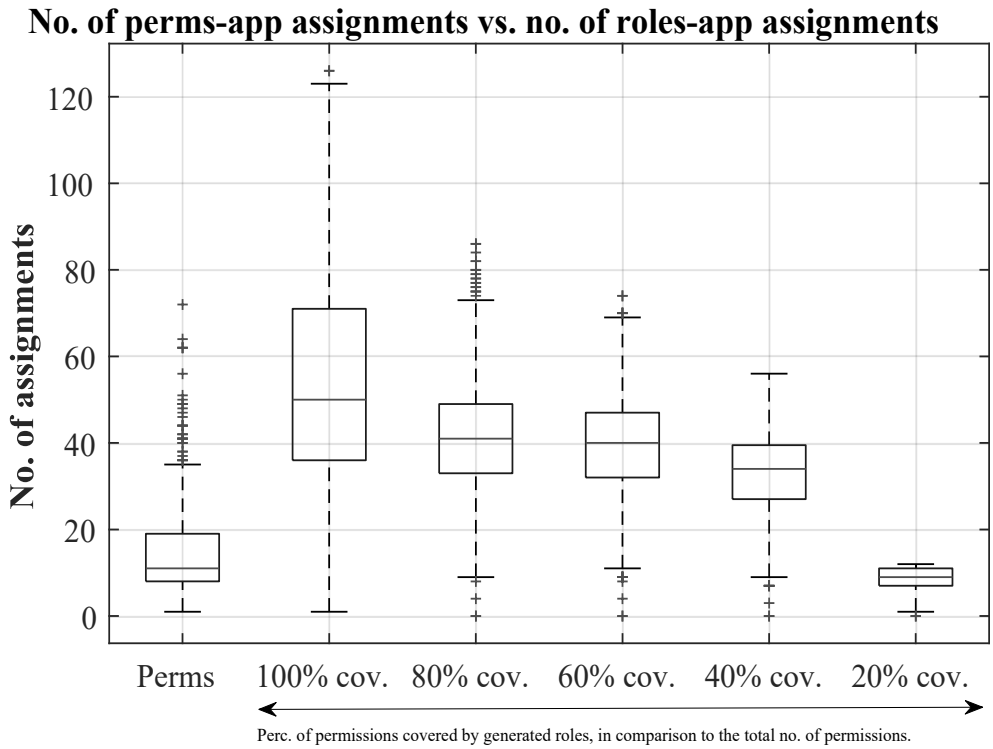


(c) Delta RMP perm privilege

Figure 4.5: Results from role mining for Android



(a) MinNoise RMP perm privilege



(b) Comparison of no. of user assignments (stock Android vs. RBAC in Android)

Figure 4.6: Results from role mining for Android (contd. . .)

After prefacing this, the results from our role mining for Android are described in brief.

The above-mentioned algorithms are run on our data set consisting of top 500 free apps from the Google Play store (obtained from APK Pure[2]). 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 perms are rarely requested by any app. It can be seen from Fig. 4.5(a), 125 perms are requested by 0 to 50 apps in our data set, and about 175 apps need between 5 to 10 perms. Fig.4.5(b) shows the percentage increase in the coverage of perms, when a greater number of generated roles are successively considered. This graph is obtained from the results of all five of the role mining algorithms. Coverage of perms is obtained by dividing the total number of unique perms assigned to any role, in a set of a certain number of roles, to the total number of perms ever requested by any app (which is known to be 161). This figure shows, that the algorithms known as Delta RMP and MinNoise RMP are the most efficient in mining roles.

The Fig.4.5(c) obtained from the results for the Delta RMP algorithm, shows the percentage of the number of roles generated, perms covered and the under-privilege of perms with respect to an increase in delta. Delta is the difference between the UPA matrix and the generated UA and PA matrices [71]. Under-privilege of perms occurs when there is a reduction in the number of perms assigned to apps in comparison to the requested number of perms. It can be observed from this graph (approx.) that when a delta of 6% is considered, the under-privilege is 4%, the perms covered are 70% however the number of roles that need to be considered are 80 (it should be noted that in the graph, the number of roles considered are not a percentage). According to the total number of perms requested by apps in our data set, which is 161, needing to consider 80 roles is a disadvantage. Next, Fig.4.6(a) which is obtained from the results of the MinNoise RMP algorithm, shows the under-privilege and over-privilege percentage of perms (over-privilege is the assignment of more than requested perms to apps) when an increasing number of successively mined roles are considered. Firstly, this graph shows that even with 20 roles mined by this algorithm, the under-privilege percentage is merely 20%; secondly, it shows the sharp rise in the over-privilege

[2]https://apkpure.com/

percentage above 120 mined roles which is noteworthy.

Finally, the Fig.4.6(b) is obtained by comparing the number of assignments between the non-RBAC, UPA based Android, to the RBAC based Android with roles generated by the MinNoise RMP algorithm. It can be observed that when the coverage is 20%, the number of role assignments drop below the number of perm assignments. This 20% coverage reflects the consideration of about 20 roles (from Figure 4.5(b)), and a corresponding under-privilege of 20% (from Figure 4.6(a)). This implies that with 10 generated roles, the under-privilege of perms is only about 1 in every 5 perms requested by the apps, and is considered by us as a positive outcome of the role mining algorithm. As stated earlier, the remaining perms required by apps can be obtained by, firstly assigning them to custom-developer-defined roles, and then by requesting those roles from the user.

CHAPTER 5: ADMINISTRATION OF RBAC IN ANDROID

This chapter describes three novel models for administration of RBAC in Android (For the RBAC in Android model under consideration - see Fig. 5.1). Since a standardization of the constituencies of roles (i.e.: the permission that are assigned to the roles (PA)) is unavailable, the administrative ease RBAC offers, cannot be envisioned accurately. This is because a proportionally higher number of roles with respect to the number of permissions, exacerbates the complexity involved in the administration of RBAC in Android, and undermines the administrative ease offered by RBAC in Android. We assume that the user of the Android device, the application developers and Google are responsible for administering the UA and PA. Also, a few roles are assumed to be present on the Android device, and other roles are assumed to be defined by the application developers, that consist of role identities and their permission assignments (PA).

Acknowledging the uncertainty of constituencies of roles, it is non trivial to design an administrative model that provides the user with sufficient control over the UA and PA operations. Borrowing from the NIST RBAC paper [31], constraints placed on the UA and PA operations alleviate some of the administrative burden from the user, by potentially reducing the frequency of prompts. Another way of reducing administrative burden, is with an automated access control system such as RAdAC [47]. RAdAC functions by allowing the user to define security tolerances (**RiskTol**) for the UA and PA operations. Then, the Android OS determines the level of risk the operations under consideration pose, and RAdAC then makes access control decisions based on this risk. The administrative models designed to manage RBAC in Android are described below.

5.1 Administrative Models for RBAC in Android

The models for administration of RBAC in Android are described in this section. Each of these models comprise of several entity sets, helper functions, relationships and convenience functions. This is followed by the operations in the model, for modification of the UA and PA relations. The models are denoted by ARiA_{ax} which stands for administration of RBAC in Android - model

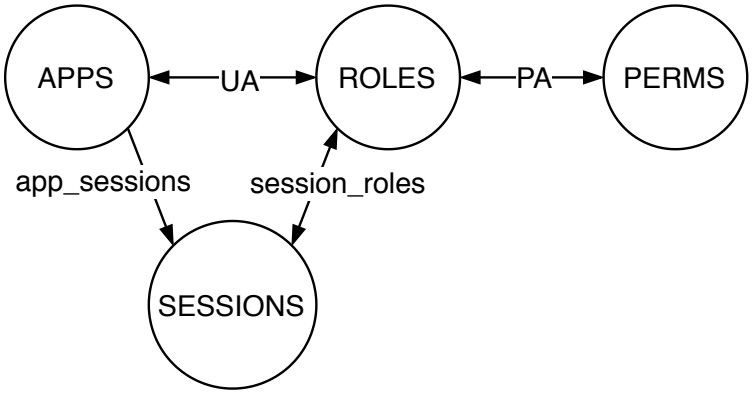


Figure 5.1: RBAC in Android

x, where x is the number distinguishing different models. The entity sets and relations from all the three models are denoted in Tables 5.1, 5.2 and 5.3.These models have been put forth in an incremental fashion, that is, the base model contains entities and relations that are common to all the three models. The rest of the models only include those additional entities and relations not already noted in the base model.

5.1.1 ARiA$_0$ (Base Model)

This is the base model for ARiA, and consists of entity sets, helper functions, relations and convenience functions.

Entity Sets for ARiA$_0$: The entity sets are designed to mimic the information stored on an Android device (see Table 5.1). These entities are populated in accordance with the policies described further in the work.

– APPS, ROLES, PERMS, the sets of applications, roles and permissions on an Android device.

– OWNER, the set of all device owners on an Android device. So, OWNER = {owner$_1$, owner$_2$, . . . , owner$_n$}

– DEV, the set of all developers for applications installed on an Android device. So, DEV = $\{\mathrm{dev}_1, \mathrm{dev}_2, \ldots, \mathrm{dev}_n\}$

– ANDROID, the set containing all the "Android" users. So, ANDROID = {android$_1$, android$_2$,

Table 5.1: Entity Sets

APPS
ROLES
PERMS
OWNER
DEV
ANDROID
AE

Table 5.2: Helper Functions

`protlvl`: ROLES $\rightarrow$ {**normal**, **dangerous**, **signature**}
`dev_of`: APPS $\rightarrow$ DEV
`usr_approved`: AE $\times$ PERMS $\times$ ROLES $\rightarrow \mathbb{B}$
`usr_approved`: AE $\times$ APPS $\times$ ROLES $\rightarrow \mathbb{B}$
`wished_roles`: APPS $\rightarrow 2^{\text{ROLES}}$
`user_sel` : $\texttt{cvar}_{\texttt{protlvl}} \rightarrow 2^{\{\textbf{normal}, \textbf{dangerous}\}}$
`pgrant_approved`: PERMS $\times$ ROLES $\rightarrow \mathbb{B}$
`rgrant_approved`: APPS $\times$ ROLES $\rightarrow \mathbb{B}$

Table 5.3: Relations and Convenience Functions

UA $\subseteq$ APPS $\times$ ROLES	`assigned_apps`: ROLES $\rightarrow 2^{\text{APPS}}$
PA $\subseteq$ PERMS $\times$ ROLES	`assigned_permissions`: ROLES $\rightarrow 2^{\text{PERMS}}$
ROLE_OWNER $\subseteq$ ROLES $\times$ AE	

$\ldots$, android_n}

– AE, the set of all administrative entities on an Android device. So, AE = OWNER $\cup$ DEV $\cup$ ANDROID.

Helper Functions for ARiA$_0$: The helper functions facilitate access control decisions by extracting data from the device itself (see Table 5.2). It should be noted that, these access control decisions refer to administrative operations themselves i.e.: whether a device administrator is allowed to modify a certain device relation (UA or PA), or not.

– `protlvl`, a function that gives the protection level for a role.

– `dev_of`, a function that returns the developer of an app *a* installed on an Android device.

– `usr_approved`, a function that depicts user approval for addition of a permission to a role.

– `usr_approved`, a function that depicts user approval for assigning a role to an app.

– `wished_roles`, a function that provides wished roles for an app installed on an Android device.

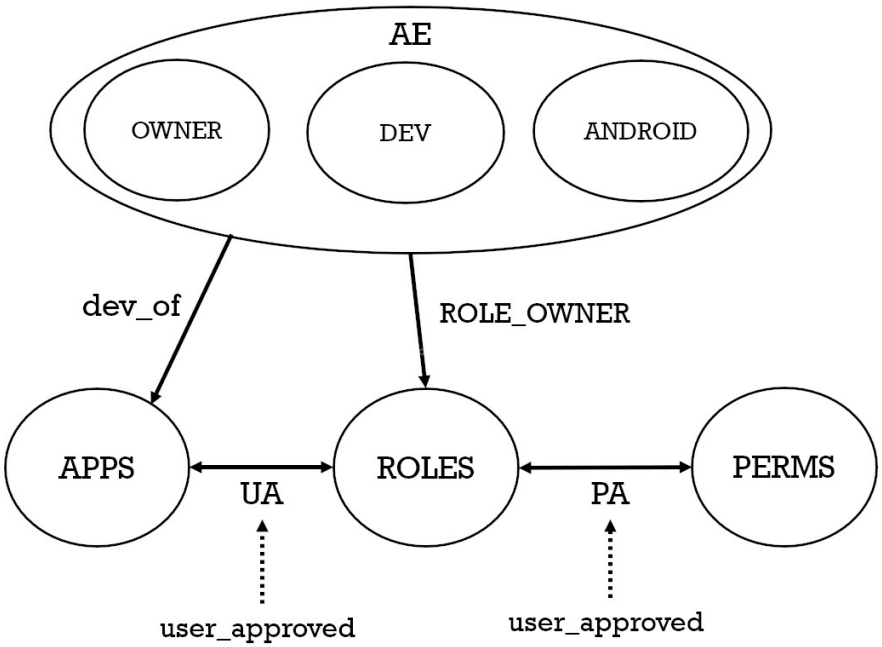


Figure 5.2: Base Model for Administration of RBAC in Android ($ARiA_0$)

Relations and Convenience Functions for $ARiA_0$: The relations denote the information stored on an Android device, and are used to make access control decisions (see Table 5.3). Convenience functions extract data from the relations stored on the device, and facilitate the access control decisions in accordance with policies defined by us.

– UA, a many-to-many mapping application to role assignment relation.

  ◦ `assigned_apps`, the mapping of a role *r*:ROLES onto a set of applications assigned to it. Formally, $\texttt{assigned\_apps}(r) = \{a \in \text{APPS} \mid (a, r) \in \text{UA}\}$.

– PA, a many-to-many mapping permission to role assignment relation.

  ◦ `assigned_permissions`, the mapping of role *r*:ROLES onto a set of permissions assigned to it. Formally, $\texttt{assigned\_permissions}(r) = \{p \in \text{PERMS} \mid (p, r) \in \text{PA}\}$.

– ROLE_OWNER, a relation mapping roles and the administrative entity that owns these roles on an Android device. Note that, $\forall r \in \text{ROLES}, \forall ae_1 \neq ae_2 \in \text{AE}.\ (r, ae_1) \in \text{ROLE\_OWNER} \rightarrow (r, ae_2) \notin \text{ROLE\_OWNER}$

Table 5.4: ARiA_0 Operations

Operation: **AssignPerm**$(ae : \text{AE}, p : \text{PERMS}, r : \text{ROLES})$ Authorization Requirement: $\Big((r,\ ae) \in \text{ROLE\_OWNER} \wedge \Big(\big(ae \in \text{DEV} \wedge \texttt{usr\_approved}(\textit{ae, p, r})\big) \vee \texttt{ae} \in \text{ANDROID}\Big)\Big)$ Updates: $\text{PA}' = \text{PA} \cup \{p,\ r\}$
Operation: **RevokePerm**$(ae : \text{AE}, p : \text{PERMS}, r : \text{ROLES})$ Authorization Requirement: $(r,\ ae) \in \text{ROLE\_OWNER}$ Updates: $\text{PA}' = \text{PA} \setminus \{p,\ r\}$
Operation: **AssignApp**$(ae : \text{AE}, a : \text{APPS}, r : \text{ROLES})$ Authorization Requirement: $r \in \texttt{wished\_roles}(a) \wedge$ $\Big(\big(\texttt{protlvl(r)} \in \{\textbf{normal}, \textbf{signature}\} \wedge \texttt{ae} \in \text{ANDROID}\big) \vee$ $\big(\texttt{protlvl(r)} = \textbf{dangerous} \wedge \texttt{ae} \in \text{DEV} \wedge \texttt{usr\_approved}(\textit{ae, a, r})\big) \vee$ $\big(\texttt{protlvl(r)} = \textbf{dangerous} \wedge \texttt{ae} \in \text{OWNER}\big)\Big)$ Updates: $\text{UA}' = \text{UA} \cup \{a,\ r\}$
Operation: **RevokeApp**$(ae : \text{AE}, a : \text{APPS}, r : \text{ROLES})$ Authorization Requirement: $ae \in \text{OWNER} \vee ae = \texttt{dev\_of}(a)$ Updates: $\text{UA}' = \text{UA} \setminus \{a,\ r\}$

Administrative Operations for ARiA_0: The administrative operations denote the modification of device relations, and to succeed, all the requisite authorization checks presented in the model must be satisfied (see Table 5.4). The operations **AssignPerm** and **RevokePerm** represent modification to the device PA relation, whereas, **AssignApp** and **RevokeApp** represent modification to the device UA relation. The operations that add a permission to a role, or a role to a user are

deemed more security sensitive than those operations that remove them; evidently, the operations that perform additions to the UA or PA relations, are fitted with more stringent security checks than the removal operations.

The **AssignPerm** operation adds a permission to a role, and can succeed if the administrative entity owns the role, and is either the Android device itself, or an app developer for an app installed on the device. If the administrative entity is an app developer, prior approval from the device owner is required before this operation can succeed. The **RevokePerm** operation removes a permission from a role, and can succeed if the administrative entity performing such an operation, owns the role under consideration.

The **AssignApp** operation assigns a role to an app, and can succeed if the role being assigned to the app, is requested by that app, and upon satisfaction of any one of the following conditions:

- If the role being assigned belongs to the **normal** or **signature** protection level, then the administrative entity performing the operation is required to be the Android device itself.
- If the role being assigned belongs to the **dangerous** protection level, then
  - if the administrative entity is the device owner themselves, the operation can succeed, or
  - if the administrative entity is an app developer for an app installed on the device, then the assignment operation requires the express approval from the device owner.

The **RevokeApp** operation de-assigns a role from an app, and can succeed if the administrative entity performing the operation is the device owner themselves, or is the developer of the app under consideration.

5.1.2 $ARiA_1$ (Constraint Based Model)

This is the second model for administration of RBAC in Android, and it uses constraints set by the device owner, to filter request prompts in order to minimize user burden. As mentioned before, the

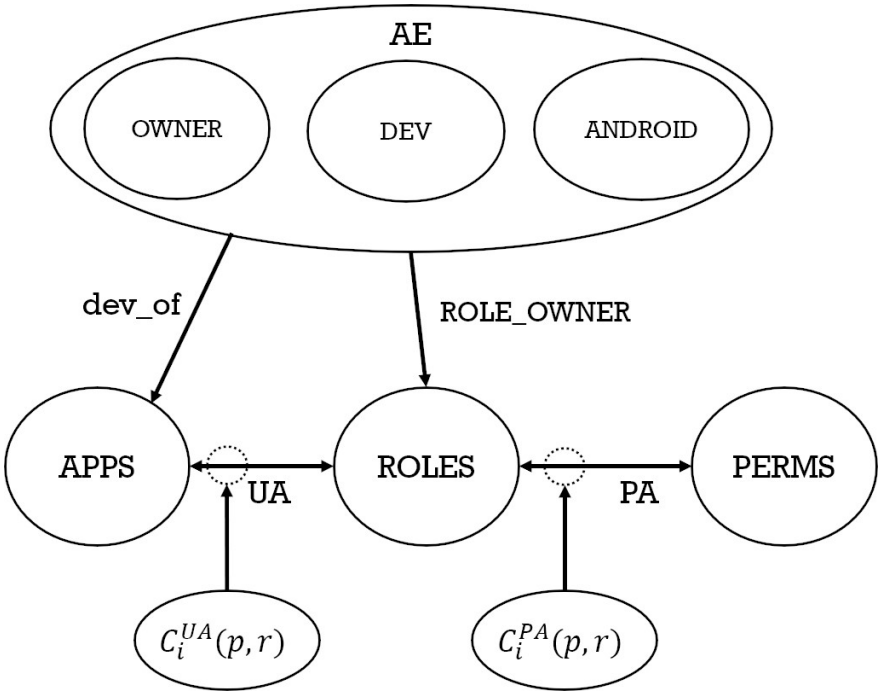


Figure 5.3: Constraint Model for Administration of RBAC in Android ($ARiA_1$)

entities and relations in addition to the base model, are stated below. These entities are followed by the administrative operations in Table 5.7.

Constraints for $ARiA_1$: As mentioned before, these constraints facilitate the mitigation of risks associated with addition operations to the device UA and PA relations. It should be noted, however, that the satisfaction of any of these constraints does not prevent risk altogether.

- *Cvars*: User defined constraint variables that set limits on PA assignments.
  - $\texttt{cvar}_{\texttt{card}}$, is the constraint on role-permission cardinality. Its values can range from 0 to |PERMS|.
  - $\texttt{cvar}_{\delta_{\texttt{card}}}$, is the constraint on the number of permissions that can be added to any role in a short amount of time. Values range from 0 to |PERMS|.
  - $\texttt{cvar}_{\texttt{protlvl}}$, is the constraint on the permissions that are allowed to be added to the roles based on their protection levels; its values range from 0 to 3. So, a value of 0 sets the protection level requirement to **normal**, 1 sets it to **dangerous** and 2 sets it to **normal** or **dangerous**. So, if the user selected protection level requirement is

Table 5.5: Constraints for the PA Relation

Constraint	Statement	Explanation
$\mathrm{C}^{\mathrm{PA}}_{card}(p,\ r)$	$\lvert\texttt{assigned\_permissions}(r)\rvert < \texttt{cvar}^{\texttt{PA}}_{\texttt{card}}$	A constraint that limits the maximum number of permissions that can be assigned to a role to *x*.
$\mathrm{C}^{\mathrm{PA}}_{\delta_{card}}(p,\ r)$	$\delta_{\texttt{assigned\_permissions}(r)} < \texttt{cvar}^{\texttt{PA}}_{\delta_{\texttt{card}}}$	A constraint that limits the number of permissions that can be added to a role in a certain time frame to *y*.
$\mathrm{C}^{\mathrm{PA}}_{protlvl}(p,\ r)$	$\texttt{protlvl}(p) \in \texttt{user\_sel}(\texttt{cvar}^{\texttt{PA}}_{\texttt{protlvl}})$	A constraint which directs that only the permissions that have a certain protection level may be assigned to a role.

Table 5.6: Constraints for the UA Relation

Constraints	Statements	Explanation
$\mathrm{C}^{\mathrm{UA}}_{card}(a,\ r)$	$\mid \texttt{app\_roles}(a) \mid < \texttt{cvar}^{\texttt{UA}}_{\texttt{card}}$	A constraint that limits the maximum number of roles that can be assigned to an app to *x*.
$\mathrm{C}^{\mathrm{UA}}_{\delta_{card}}(a,\ r)$	$\delta_{\texttt{app\_roles}(a)} < \texttt{cvar}^{\texttt{UA}}_{\delta_{\texttt{card}}}$	A temporal constraint that limits the number of roles that can be added to an app in a certain time frame, to *y*.
$\mathrm{C}^{\mathrm{UA}}_{protlvl}(a,\ r)$	$\texttt{protlvl}(r) \in \texttt{user\_sel}(\texttt{cvar}^{\texttt{UA}}_{\texttt{protlvl}})$	A constraint which directs that only the roles that have a certain protection level may be assigned to an app.

normal then the permissions belonging to the **dangerous** protection level may not be assigned to that role. It should be noted that, when the constraint fails, the user can still approve such an operation explicitly, as shown in the function `pgrant_approved`.

- $\mathrm{C}^{\mathrm{PA}}_{i}(p,\ r)$, a constraint statement particular to the PA relation, that evaluates to either *true* or *false*. These constraints are defined in Table 5.5; and, need to be satisfied prior to a permission being assigned to a role.

- $\mathrm{C}^{\mathrm{UA}}_{i}(p,\ r)$, a constraint statement particular to the user assignment (UA), that evaluates to either *true* or *false*. These constraints are defined in Table 5.6; and, need to be satisfied prior to a permission being assigned to a role.

Helper Functions for ARiA$_1$: These functions extract data stored on the device, and facilitate

Table 5.7: ARiA_1 Operations

Operation: **AssignPerm**$(ae : \text{AE}, p : \text{PERMS}, r : \text{ROLES})$ Authorization Requirement: $(r,\ ae) \in \text{ROLE\_OWNER} \wedge \Big(ae \in \text{DEV} \wedge \texttt{pgrant\_approved}(ae, p, r) \vee ae \in \text{ANDROID}\Big)$ Updates: $\text{PA}' = \text{PA} \cup \{p,\ r\}$
Operation: **RevokePerm**$(ae : \text{AE}, p : \text{PERMS}, r : \text{ROLES})$ Authorization Requirement: $(r,\ ae) \in \text{ROLE\_OWNER}$ Updates: $\text{PA}' = \text{PA} \setminus \{p,\ r\}$
Operation: **AssignApp**$(ae : \text{AE}, a : \text{APPS}, r : \text{ROLES})$ Authorization Requirement: $r \in \texttt{wished\_roles}(a) \wedge \Bigg(\Big(\texttt{protlvl(r)} \in \{\textbf{normal}, \textbf{signature}\} \wedge ae \in \text{ANDROID}\Big) \vee \Big(\texttt{protlvl(r)} = \textbf{dangerous} \wedge ae = \texttt{dev\_of}(a) \wedge \texttt{rgrant\_approved}(a, r)\Big) \Big(\texttt{protlvl(r)} = \textbf{dangerous} \wedge ae \in \text{OWNER}\Big)\Bigg)$ Updates: $\text{UA}' = \text{UA} \cup \{a,\ r\}$
Operation: **RevokeApp**$(ae : \text{AE}, a : \text{APPS}, r : \text{ROLES})$ Authorization Requirement: $ae \in \text{OWNER} \vee ae = \texttt{dev\_of}(a)$ Updates: $\text{UA}' = \text{UA} \setminus \{a,\ r\}$

access control decisions. The particular helper functions for this model enable provide assistance to place constraints on the modifications of the UA and PA relations.

- $\delta_{\texttt{assigned\_permissions}(r)}$ and $\delta_{\texttt{app\_roles}(a)}$, these functions track the number of assignments (from the co-domain of the function to the domain) for a function, within a certain time frame. The time frame is arbitrated by the user.

- `user_sel`, a function that maps the user input to the protection level requirement for assign-

ing permissions to roles or roles to applications.

- `pgrant_approved`, a function that seeks user approval and ensures certain constraints are satisfied for modifications done to the PA relation. The constraints for this function are shown in Table 5.5. Note that, $\forall ae \in \text{AE}, \forall r \in \text{ROLES}, \forall p \in \text{PERMS}.$ `pgrant_approved`$(p, r) \rightarrow \Big(\bigwedge_{i=1}^{n} \text{C}_i^{\text{PA}}(p, r) \vee$ `usr_approved`$(ae, p, r) \Big)$. If all the constraints are satisfied, then user prompt is not shown, however, if any of the constraints are not satisfied, the user is required to approve the request.

- `rgrant_approved`, a function that seeks user approval and ensures certain constraints are satisfied for modifications done to the UA relation. The constraints for this function are shown in Table 5.6. Note that, $\forall ae \in \text{AE}, \forall a \in \text{APPS}, \forall r \in \text{ROLES}.$ `rgrant_approved`$(ae, a, r) \rightarrow \big(\bigwedge_{i=1}^{n} \text{C}_j^{\text{UA}}(a, r) \wedge$ `usr_approved`$(ae, a, r) \big)$.

Note the difference between `pgrant_approved` and `rgrant_approved` functions, in that the latter requires the user's express approval for assigning a role to an app, even when the constraints are satisfied. This is because, applications utilize the roles to access components on the device; whereas, roles are merely a tool, to organize permissions, and do not grant access to the device components themselves.

Administrative Operations for ARiA$_1$: The administrative operations for ARiA$_1$ are shown in Table 5.7. The authorization requirements for the **RevokeApp** and the **RevokePerm** operations are identical to the ones for ARiA$_0$. The authorization requirements for **AssignPerm** and the **AssignApp** operations are based on constraints described earlier.

The **AssignPerm** operation assigns a permission to a role, and can succeed when the administrative entity performing the operation owns the role under consideration, and is either the Android device itself or an app developer for an app installed on that device. If the administrative entity is an app developer, then the operation succeeds either if all the constraints are satisfied, or upon express approval from the device owner. The **AssignApp** operation assigns a role to an app, and can succeed if the role being assigned is requested by the app under consideration, and either of

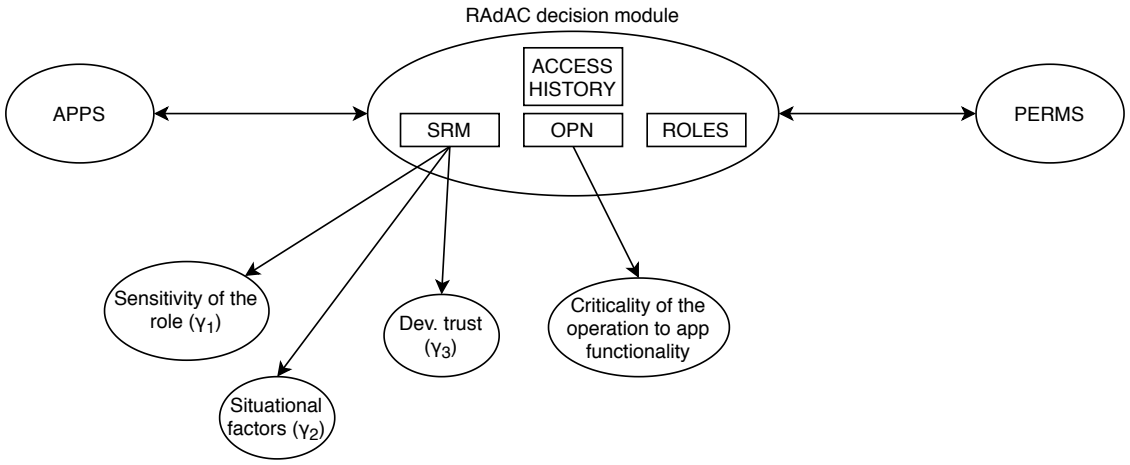


Figure 5.4: Administration of RBAC in Android using Risk-adaptive approach (RAdAC)

the following conditions are satisfied.

- If the role being assigned belongs to the **normal** or the **signature** protection level, then the administrative entity is required to be the Android device itself.
- If the role being assigned belongs to the **dangerous** protection level, then either
  - the administrative entity is the device owner, or
  - the administrative entity is the app developer for the app under consideration, and all the constraints are satisfied along with express user approval.

5.1.3 ARiA$_2$ (RAdAC Based Model)

This model is based on the RAdAC model [47], and takes into consideration the operational risk before automatically making an access control decision (see Figure 5.4). The main building blocks for this model are the **SecRisk** and the **OpNeed** modules.

Operational Need Module (OPN): This module provides the quantifiable role request rationale (see Table 5.8). The application developer can choose to provide the value of the operation to the app-functionality. If the security risk posed by the operation can be challenged by this value, the operation may still succeed.

Security Risk Module (SRM): This module calculates the quantifiable risk associated with each operation based on the operation itself and a number of situational factors. The operational security risk is always elevated for any *assign* operations and is reduced for corresponding *revoke* operations. This is because assign operations are more security sensitive than the revoke operations. The situational factors are - Location of the device, Time at which operation is initiated, whether the device owner is busy using another app, proximity of the device to other Bluetooth/WiFi/NFC devices, and, in case of enterprise scenario, whether the owner is logged in to the enterprise network. Total risk for any given operation is given by $\sum_{i=1}^{n} \mathrm{R}_i$.

Risk Threshold (RiskTol): The relative security budget defined by the user for any particular operation. Users can define a security budget value for all operations. If an operation's security risk exceeds its budget value, that operation can be denied by the RAdAC system.

The term OP is the set of app operations for the Android device. Note that, OP = {**AssignPerm**, **RevokePerm**, **AssignApp**, **RevokeApp**}.

Administrative Operations for ARiA$_2$: The administrative operations for the ARiA$_2$ are authorized by the RAdAC system (see Table 5.9). Initially, the device owner defines a risk threshold denoted by **RiskTol**. This is the overall security risk, the device owner is willing to take, for that particular operation. The RAdAC system then subtracts the calculated security risk via the SRM module, which is based on situational factors such as location of the device, time of the day and proximity to certain wifi networks. Then, the operational need provided by the app developer is added to this quantity to obtain the final operational qualifying number. If this is greater than zero, the operation can succeed, and if not, the operation fails.

Table 5.8: Operational need for applications defined by developers

Op-need	Explanation
0	Role not critical to app functionality, but complements it.
1	App needs the role, but if not available, does not impact user experience.
2	App can still function, but user experience is severely hampered.
3	App category prevents major functionality without role grant.

Table 5.9: ARiA$_2$ Operations

Operation: **AssignPerm**$(ae : \text{AE}, p : \text{PERMS}, r : \text{ROLES})$ Authorization Requirement: $\texttt{radac}_{\textbf{AssignPerm}}(ae,\ p,\ r)$ Updates: $\text{PA}' = \text{PA} \cup \{p,\ r\}$
Operation: **RevokePerm**$(ae : \text{AE}, p : \text{PERMS}, r : \text{ROLES})$ Authorization Requirement: $\texttt{radac}_{\textbf{RevokePerm}}(ae,\ p,\ r)$ Updates: $\text{PA}' = \text{PA} \setminus \{p,\ r\}$
Operation: **AssignApp**$(ae : \text{AE}, a : \text{APPS}, r : \text{ROLES})$ Authorization Requirement: $\texttt{radac}_{\textbf{AssignApp}}(ae,\ p,\ r)$ Updates: $\text{UA}' = \text{UA} \cup \{a,\ r\}$
Operation: **RevokeApp**$(ae : \text{AE}, a : \text{APPS}, r : \text{ROLES})$ Authorization Requirement: $\texttt{radac}_{\textbf{RevokeApp}}(ae,\ p,\ r)$ Updates: $\text{UA}' = \text{UA} \setminus \{a,\ r\}$

$$\texttt{radac}_{Op}(ae,\ p,\ r) \rightarrow \Big(\mathbf{RiskTol(op)} - \sum \mathbf{SecRisk}(\gamma) + \mathbf{OpNeed(op)}\Big) > \mathbf{0}$$

where $op \in \text{OP}$, the **RiskTol** is the threshold which is selected by the user for that operation.

5.2 Discussion

In this section, we discuss the operations involved in the administrative RBAC in Android. As we use constraints in ARiA$_1$, we argue the rationale for such constraints, and their effects on user burden for administering RBAC in Android. A few examples for administrative operations are also provided in this section.

Table 5.10: Permissions required by WhatsApp in Android

ACCESS_NETWORK_STATE
ACCESS_WIFI_STATE
CAMERA
INTERNET
READ_CONTACTS
RECEIVE_BOOT_COMPLETED
RECORD_AUDIO
STORAGE
VIBRATE
WRITE_EXTERNAL_STORAGE
READ_PHONE_STATE
READ_SYNC_SETTINGS
READ_SYNC_STATS
WRITE_SYNC_SETTINGS
WAKE_LOCK
MODIFY_AUDIO_SETTINGS

(a) Permissions used frequently

ACCESS_COARSE_LOCATION
ACCESS_FINE_LOCATION
CALL_PHONE
CHANGE_WIFI_STATE
GET_ACCOUNTS
GET_TASKS
MANAGE_ACCOUNTS
READ_PROFILE
USE_CREDENTIALS
WRITE_CONTACTS
WRITE_SETTINGS

(b) Permissions used on occasion

AUTHENTICATE_ACCOUNTS
RECEIVE_SMS
SEND_SMS
INSTALL_SHORTCUT

(c) Permissions used rarely

5.2.1 Rationale for the Constraints on Modifications to PA and UA

App developers are required to use the minimum necessary permissions for functionality in line with principle of least privilege. But they do not adhere to this principle and statistically request more permissions than their applications need; prior research heavily indicates over-privilege in Android [29, 36, 73, 75, 76]. This equates to the developers over-privileging their applications even when permissions are assigned directly to applications. Introduction of roles can compound this issue, due to the reduction in granularity. So, constraints on the assignment of permissions to roles and consequently the assignment of roles to applications are necessary.

Out of all the permissions that are legitimately required by any app, not all permissions are needed all the time. For example, WhatsApp requests 31 permissions (that are **normal** or **dangerous**)(from Table 5.10), however almost 13% permissions are very rarely used. Hence, it should be required for developers to assign permissions to roles based on the frequency of usage of such permissions. However, Android does not provide an easy way to monitor permission usage, hence, a cardinality based constraint can mitigate this issue by limiting the maximum number of

permissions that are assigned to roles, and further by limiting the maximum number of roles that can be assigned to applications. Apart from this, a higher number of roles, equates more prompts shown to the user. Due to this, users can develop fatigue and choose to simply uninstall the app under consideration. This can prompt the developer to reduce the total number of roles they assign their permissions to, which can further exacerbate the issue of over-privileged applications.

Furthermore, legitimate applications can get tricked by malicious applications to reveal user data, resulting in the well known privilege escalation attacks [18, 25, 51]. Many applications on the Play Store use ad libraries to provide additional income to developers; these libraries can then gain access to user data if applications remain over-privileged. These leaks of user data can be problematic and can result in a monetary loss for the user. While solving the issue of permission over-privilege is outside the scope of this work, these constraints attempt to mitigate the over-privilege to help limit the damage that can be caused by such an over-exposure of permissions. In the following subsection, a few examples of the administrative operations are presented.

5.2.2 Example Operation - AssignApp

Consider the scenario where an app is installed on an Android device, called WhatsApp. The app requests access to one of the dangerous roles present on the device. This corresponds to the **AssignApp** administrative operation. The app has requested access to one other role within the past five minutes, and has been granted four roles in total. The constraint variables (for $ARiA_1$) that have been set by the user are shown in Table 5.11.

The risk threshold and security risk values, for the **AssignApp** operation (for $ARiA_2$) are shown below.

- **RiskTol**(**AssignApp**) = 16
- $\sum$**SecRisk** = 12
- **OpNeed**(**AssignApp**) = 3

The behavior for this operation according to all the three models presented in this work is as

follows.

AssignApp operation for ARiA$_0$: The third line in the authorization requirements (see Table 5.7) states that such a request is basically forwarded to the user as a role prompt. The device owner can accept this prompt to grant the role to that application. The manner in which this model forwards such requests to the users, results in prompts for every role request, which increases user burden in administration of the device.

AssignApp operation for ARiA$_1$: In this case, the function `rgrant_approved` needs to return true for the role to be granted. As we can see from the Fig. 5.5 (see also function `rgrant_approved`), that apart from the satisfaction of all the constraints put forth by the owner, his (owner's) express approval is required for the role to be granted to the application. This is because, such an operation is highly security sensitive, since granted roles enable the app complete access to the corresponding resources. The constraints (see Table 5.6) are intended to act as a filter to incoming role request prompts.

The first constraint limits the maximum number of roles that can be assigned to any app. As $\texttt{cvar}^{\texttt{UA}}_{\texttt{card}} = 5$, and $|\texttt{app\_roles}(\text{WhatsApp})| = 4$, the first constraint is satisfied. The second constraint limits the number of roles that can be granted in quick succession. This constraint is intended to mitigate the issue of app requesting multiple roles in quick succession that causes the device owner to get fatigued by successive role prompts. As $\texttt{cvar}^{\texttt{UA}}_{\delta_{\texttt{card}}} = 2$, and since WhatsApp has been granted only one other role recently, this constraint is satisfied as well. It should be noted that the temporal constraint (i.e.: the exact time constraint) within which an app must request roles, is not decided by this work. Further research is needed to select an ideal time for such a constraint. The third constraint sets a limit on the maximum allowable protection level for a permission.

Table 5.11: UA Constraints for ARiA$_1$, for the AssignApp Operation

Constraint	Set value	Calculated value
$\texttt{cvar}^{\texttt{UA}}_{\texttt{card}}$	5	4
$\texttt{cvar}^{\texttt{UA}}_{\delta_{\texttt{card}}}$	2	1
$\texttt{cvar}^{\texttt{UA}}_{\texttt{protlvl}}$	**normal**	**dangerous**

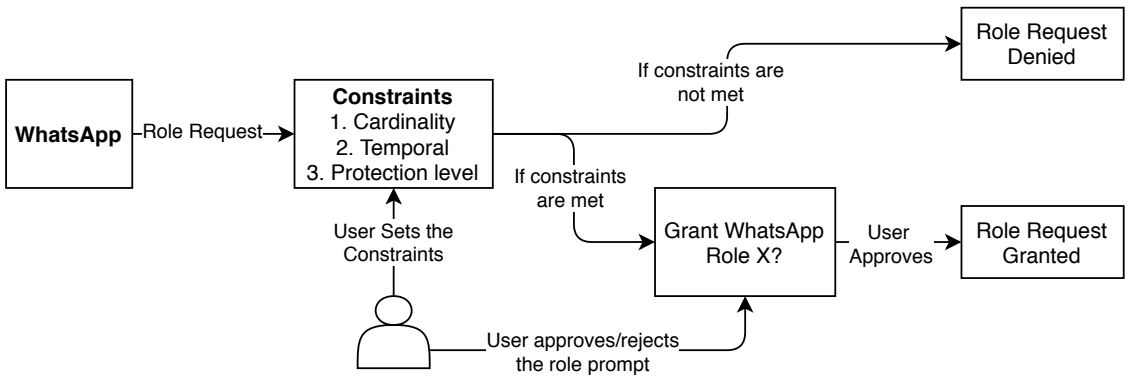


Figure 5.5: ARiA$_1$: UA Constraints Based Administration of RBAC - **AssignApp** Operation

As $\texttt{cvar}^{\texttt{UA}}_{\texttt{protlvl}}$ = **normal**, and the currently requested role is **dangerous**, this constraint is not satisfied and the role request is thus automatically declined.

AssignApp operation for ARiA$_2$: The RAdAC model states that the user defined security tolerance should be higher than the security risk for an operation, for the operation to allowed to complete. A small factor called **OpNeed** allows the app developers to indicate if the role request is critical for the app functionality. As the **RiskTol** is set at 16, and the **SecRisk** calculated by Android is 12, with the **OpNeed** specified by the app developer to be 3, the operation would succeed (16 - 12 + 3 = 5 which is greater than zero).

5.2.3 Example Operation - AssignPerm

The **AssignPerm** operation is used to assign a permission to a role. Consider a scenario where an app, WhatsApp, attempts to add a permission "android.permission.CAMERA" which is a dangerous permission to one of the roles assigned to it, r1. The role r1 has 9 other permissions assigned to it. The values for the constraints on the permission-assignment relation are stated in Table 5.12.

The risk threshold and security risk values, for the **AssignPerm** operation (for ARiA$_2$) are

Table 5.12: PA Constraints for ARiA$_1$, for the AssignPerm Operation

Constraint	Set value	Calculated value
$\texttt{cvar}^{\texttt{PA}}_{\texttt{card}}$	10	9
$\texttt{cvar}^{\texttt{PA}}_{\delta_{\texttt{card}}}$	5	2
$\texttt{cvar}^{\texttt{PA}}_{\texttt{protlvl}}$	**normal**	**dangerous**

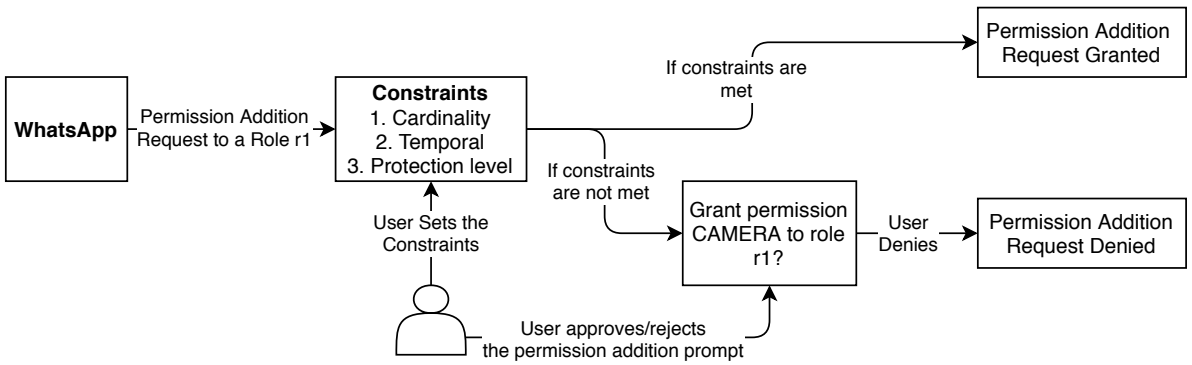


Figure 5.6: ARiA$_1$ Constraint Based Administration of RBAC - **AssignPerm** Operation

shown below.

- **RiskTol**(**AssignPerm**) = 12
- $\sum$**SecRisk** = 9
- **OpNeed**(**AssignPerm**) = 2

AssignPerm operation for ARiA$_0$: The authorization requirement for the **AssignPerm** operation states that, if the request for adding a permission to role originates from the app (and by extension the app developer) such an operation requires prior user approval in the form of a permission addition prompt. If the user accepts such a prompt, the permission may be added to the role.

AssignPerm operation for ARiA$_1$: The authorization requirement for the **AssignPerm** operation states that the function `pgrant_approved` needs to return true, for the permission to be added to the role. Further, it can be seen from Fig.5.6 that this function states that either all the constraints set by the owner (see Table 5.12) needs to be satisfied, or the owner should accept the incoming permission addition prompt. As mentioned earlier, the role r1 has nine other permissions assigned to it, so the cardinality constraint is satisfied. The second constraint is a temporal constraint, which is also satisfied, since only two permissions were added to this role recently. The third constraint states that if the permission being assigned to the role should belong to an equal or higher protection level, as compared to the role. In this case, however, since the permission is a dangerous one, and the role it is being added to is a normal role, this constraint is not satisfied.

However, it should be noted that the user can still accept the resultant permission addition prompt, for the permission to be assigned to the role r1.

AssignPerm operation for $ARiA_2$: The RAdAC model calculates security risk for the operation based on a number of situational factors. In this case, the operation is approved, since the addition of the security risk and the operational need exceeds the risk threshold defined by the owner (12 - 9 + 2 = 5 which is greater than zero).

CHAPTER 6: IMPLEMENTATION

This chapter describes our implementation of RBAC and its administrative components in the Android OS. The Android OS contains a vast array of individual system components, each performing specific tasks, and learning the internal mechanics of these components required considerable time and effort. Most of the documentation available, pertains to publicly released APIs, that are used to develop applications on Android; the documentation related to Android's source code is severely limited. Therefore, this chapter serves as additional purpose, of providing crucial information to beginner researchers attempting to modify framework components in Android.

We discuss the system files in Android that were modified, the Role Manager developed by us to manage role-permission (PA) and app-role (UA) assignments, and the Role Manager service, that enables installed applications to request and activate roles. A list of all the system files modified and their path in the Android Open Source Project (AOSP) directory can be found in the Appendix B. We modified and compiled the source code for API 29 (current at the time). The models RiA_a and ARiA_0 were implemented in Android and described below, a high level overview is detailed in Fig. 6.1. These models were implemented at the framework level in Android, and they utilize Android's internal permission checking mechanism to allow or deny apps from accessing the corresponding resources. The source code for the Role Manager, the Role Manager Service, and the modified files in AOSP is at [62].

6.1 Role Manager

The Role Manager is a system component, that is built at the framework level to manage all role to permission (PA), and role to app (UA) assignments. Some default roles are built into the Role Manager, for use by applications. The Role Manager receives role requests from applications, and prompts the user to accept such requests. If the user accepts the request the roles are granted to the applications. However, this does not grant the application access to the resources in accordance with the permissions assigned to that role, because the role needs to be activated by the application.

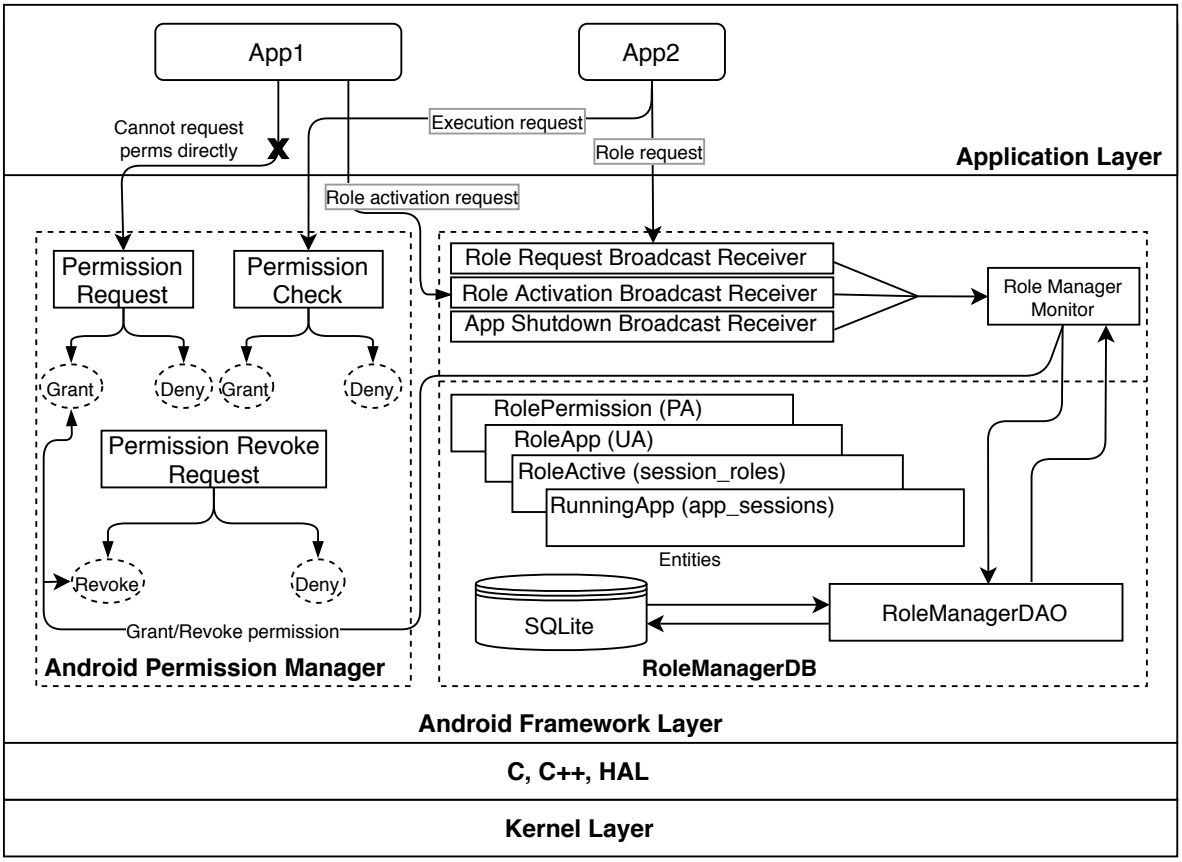


Figure 6.1: RiA_a implementation

Once the application launches, it can activate the role, thus granting all the permissions for that role to the application. The Role Manager records granted and activated roles in a database, built using the Android Room Persistence library, and the files used to build this database are denoted in Fig. 6.2, and are described below.

RoleManagerDB.java The Role Manager stores the UA and PA relations in an SQLite database, which is constructed using the Android Room Persistence library. The Room library provides a layer of abstraction over SQLite, and eliminates the need for most boilerplate SQL code. This file creates a database with the provided string name, that includes all the tables corresponding to the entities that are provided to the database using the @Database annotation.

RoleManagerDAO.java The DAOs or Data Access Objects is a class where we define methods for database queries. The Android Room library provides four annotations for simplifying database

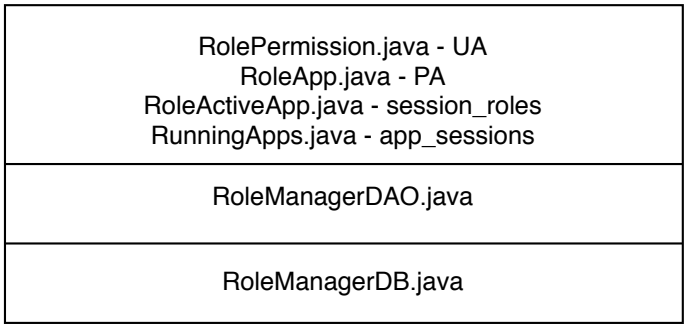


Figure 6.2: Role Manager Files

operations, namely, @Insert, @Update, @Delete and @Query. We utilized these annotations to perform CRUD operations on the RoleManagerDB.

RolePermission.java, RoleApp.java, RoleActiveApp.java, RunningApp.java These are the entities that are built into the Role Manager, and each entity corresponds to a table in the Role Manager database. These files contain several fields, and each field corresponds to a column in the table formed by the entity. This file also contains the setters and getters for the entities.

RoleRequestReceiver.java This broadcast receiver is used to receive role request broadcasts from the Role Manager Service. When applications request roles, this request is sent to the Role Manager Service, which then forwards this request to the Role Manager via a system wide broadcast. The Role Manager obtains the package name for the request, and displays a prompt to the user via Prompt.java; this prompt contains the package name of the application, the role being requested, and a short description of the type of access allowed by the role.

RoleActivationReceiver.java This broadcast receiver is used to receive role activation broadcasts from the Role Manager Service. When applications request role activation, this request is forwarded to the Role Manager Service, which in turn sends a system wide broadcast to the Role Manager. The Role Manager activates the requested role, grants the permissions mapped to that role, and displays the role activation confirmation dialog (see Fig. 6.12b).

AppUninstallReceiver.java The package manager service is modified to send a system wide broadcast whenever any application is uninstalled from an Android device. This broadcast is received by the AppUninstallReceiver. When the Role Manager receives such broadcasts, it revokes

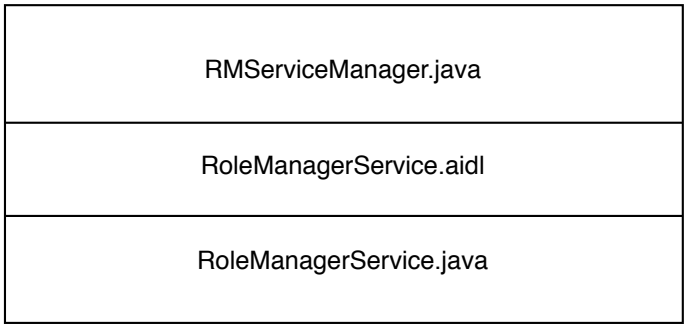


Figure 6.3: Role Manager Service Files

all roles from the application, and revokes all the permissions mapped to the roles.

Prompt.java The role request prompt is displayed by this file. The Role Manager presents the user with a role prompt to accept (see Fig. 6.12a), and if the user chooses to accept it, the role is granted to the application.

6.2 Role Manager Service

The Role Manager Service is the front-end to the Role Manager, and is responsible for receiving role prompts and role activation prompts, and, forward these requests to the Role Manager via protected system wide broadcasts. The files with which the Role Manager Service is designed are described below (see Fig. 6.3).

RMServiceManager.java The RMService manager is responsible for receiving role requests and role activation requests and forwarding it to the Role Manager Service. The RMService manager does this by using an AIDL (Android Interface Definition Language), called as RoleManagerService.aidl.

RoleManagerService.aidl Due to sandboxing, applications or services in Android cannot talk to each other without using interprocess communication (IPC). Designing and implementing IPC is tedious, hence, Andoid provides the AIDL to manage IPC. We define two methods in the AIDL file for the Role Manager Service, one to request roles, and another to activate roles.

RoleManagerService.java This is a system service built by us, to responds to role requests and role-activation requests from the applications, and forwards these requests to our Role Manager.

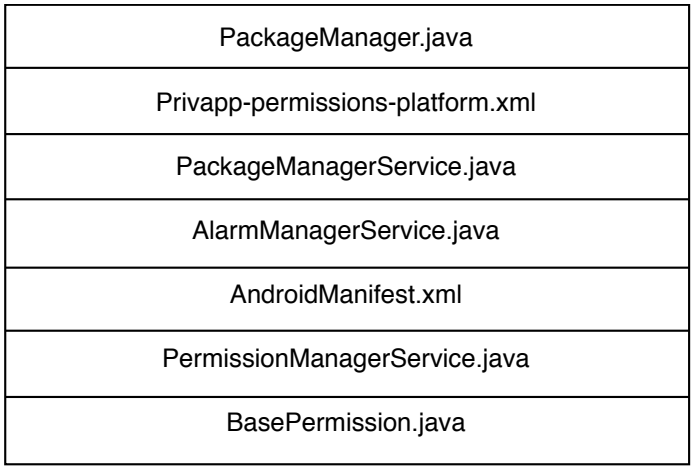


Figure 6.4: Framework Files Modified to Facilitate Role Manager Operations

This service was implemented using the Android Interface Definition Language (AIDL), so that applications can request roles and request role activation. The Role Manager Service is built as a system service, to enable it to obtain access to hidden APIs in Android. To achieve this, we modified several system files as stated below.

6.3 Framework Modifications to Facilitate Role Manager Operations

System files were modified to facilitate Role Manager operations like granting and revoking roles, and activate and de-activating roles. Android contains system APIs that are hidden from third party applications. Since we used Android Studio to compile the Role Manager, we had to expose these hidden APIs to the SDK so that Android Studio could compile the Role Manager as a system application; this is because, in the downloaded version of Android SDK (any API), these methods are missing. This section also includes all the other modifications to the system as well, to facilitate Role Manager functions such as application shutdown detection, application uninstallation detection and to granting of normal permissions along with dangerous permissions. All the files that were modified in the Android OS are described below (see Fig. 6.4).

PackageManager.java The Role Manager utilizes the package manager to grant and revoke permissions, corresponding to the role grants and revocations. Typically, the grantRuntimePermission and revokeRuntimePermission methods are unavailable to external applications. The Role

```
<privapp-permissions package="com.samir.samirrolemanager">
<permission name="android.permission.GRANT_RUNTIME_PERMISSIONS"/>
<permission name="android.permission.REVOKE_RUNTIME_PERMISSIONS"/>
</privapp-permissions>
```

Figure 6.5: Permission Whitelist for the Role Manager

Manager was built using Android Studio (AS), and the AS does not recognize the above mentioned methods, and so would not compile the Role Manager, without a custom Android SDK with those methods exposed. To achieve this, we removed the @Systemapi and @hide annotations from these methods, in this file. We also added these methods to the android.txt file in the prebuilts folder, while commenting the lines with these methods in the android.test.mock.txt file (see Appendix B).

Privapp-permissions-platform.xml We added permission entries for the Role Manager in this file, to enable it to obtain system-only permissions. For the Role Manager to be able to grant and revoke permissions, the system permissions denoted in Fig. 6.5 are required. Although, these permissions are only granted to system components and applications, this file serves as a whitelist for system permissions; any such permissions added to this file are automatically granted to the package described in the entry.

PackageManagerService.java When applications are uninstalled from the Android device, all the granted roles for that application needed to be revoked, and activated roles needed to be de-activated. To achieve this, we added the functionality to send out protected system wide broadcast on app un-installation, from the package manager service. This broadcast is received by the Role Manager, which then revokes and de-activates all the roles from the app that is un-installed. The broadcast that is sent when applications are uninstalled is stated below.

- Broadcast sent on app uninstallation: **Intent.ACTION_PACKAGE_FULLY_REMOVED**

AlarmManagerService.java The timer set with the alarm manager keeps roles de-activated for all but the running applications. The alarm manager, however, has a built in mechanism to prevent alarms from being scheduled for less than 60 seconds. This is done to prevent applications from setting alarms too frequently, which can drain the device battery. We overrode this mechanism

to allow our Role Manager to be able to monitor running applications more frequently. When the Role Manager detects the application being shutdown or killed, it de-activates the roles and revokes all the permissions from the application under consideration.

AndroidManifest.xml The AndroidManifest.xml file for the platform, contains all the permission definitions and permission group assignments for that device. We removed the permission-group associations for all the dangerous permissions defined in this file, since permission-groups are not used by the Role Manager. This was done so that, all the dangerous permissions can be granted using the Role Manager's internal PA table instead of permission groups.

PermissionManagerService.java In a stock Android OS, dangerous permissions are granted using the method grantRuntimePermission, and revoked using the revokeRuntimePermission method. We modified these methods in this file, to grant and revoke normal permissions as well as dangerous ones. This allows the Role Manager to grant and revoke dangerous as well as normal permissions with the two stated methods.

BasePermission.java The enforceDeclaredUsedAndRuntimeOrDevelopment method in this file was modified, to allow the Role Manager to grant and revoke normal permissions along with dangerous ones. In the stock Android OS, normal permissions are granted and revoked differently than dangerous or signature ones. These permissions are enforced at the kernel level, and when a normal permission is granted to an application, Android (zygote) associates the app process with the GID of the permission. If the GID of any process changes at runtime, that process is killed automatically. This is one of the issues we encountered while building the Role Manager, and it results in the app process being killed, if any of the roles it activates contain normal permissions. We remedy this issue, by detecting normal permissions in roles that are about to be activated, and then simply re-starting the app process after the normal permission is granted.

6.4 Role Manager Installation as a System Application

In order to install the Role Manager in the privapp partition of the Android emulator, the following files needed to be changed. System applications can access hidden APIs that third party

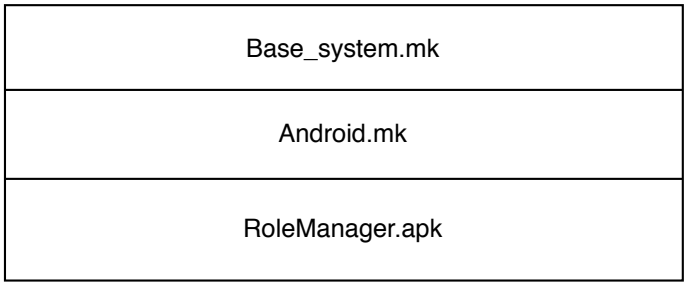


Figure 6.6: Files Modified for Installing Role Manager as a System Application

applications do not have access to, and it is for this reason we chose to install the Role Manager as a system application. All the modifications to the system files are described below (see Fig. 6.6). The Android.mk and the apk file for the Role Manager are inserted into the <AOSP root>/packages/RoleManager directory.

Base_system.mk This file serves as the main file for the NDK build system, and lists the packages to be built, that are common to all the targets platform versions in the <AOSP root>/device directory. We added the package name for the Role Manager to this file (under the "PRODUCT_PACKAGES" variable), so that it would be included along with other packages built when compiling the AOSP.

Android.mk A file labelled Android.mk needs to be added to the root folder of the system-app being installed. This file relays important information about the application being installed, such as the package name, source directory, the signature the package is to be signed with (for the Role Manager, it is signed with the "platform" signature), and whether any dependencies need to be resolved before the application is compiled (for the Role Manager, since the package is already compiled, we do not have any more dependencies to be inserted here). The Android.mk file used by us is denoted in Fig. 6.7.

RoleManager.apk This is the compiled application package for the Role Manager. As stated before, it was compiled with the Android Studio software, and includes all the required files for the Role Manager to run. This file was added to the <AOSP root>/packages/RoleManager directory, along with the Android.mk files described above.

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE_TAGS := optional

LOCAL_MODULE := samirrolemanager

LOCAL_CERTIFICATE := platform

LOCAL_PRIVILEGED_MODULE := true

LOCAL_SRC_FILES := samirrolemanager.apk

LOCAL_MODULE_CLASS := APPS

LOCAL_MODULE_SUFFIX :=
$(COMMON_ANDROID_PACKAGE_SUFFIX)

include $(BUILD_PREBUILT)
```

Figure 6.7: Android.mk file for the Role Manager

6.5 Role Manager Service Installation as a System Service

Since the Role Manager Service is built as a system service, we needed to modify several files to register this service with the service manager, instruct SELinux to allow the Role Manager to communicate with this service, and to allow other applications to send role requests and role activation requests to the service. All the files that were modified to achieve this are described below (see Fig. 6.8).

Android.bp This is the Blueprint file for the Soong build system (in the <AOSP root> /frameworks/base/services/core directory), which is equivalent to the Android.mk file used for by the NDK build system. We added dependencies for the Role Manager Service, that were not already included in this file by default. Namely, we added the AndroidX-Annotations library as a dependency for the Role Manager Service. These libraries are required because, the Service uses @Nullable annotation for the package-name and the role-name that is passed with the role request and the role activation requests.

Android.mk This is the framework makefile (in the <AOSP root>/frameworks/base directory) that includes all the framework components that need to be compiled. We added an entry for the

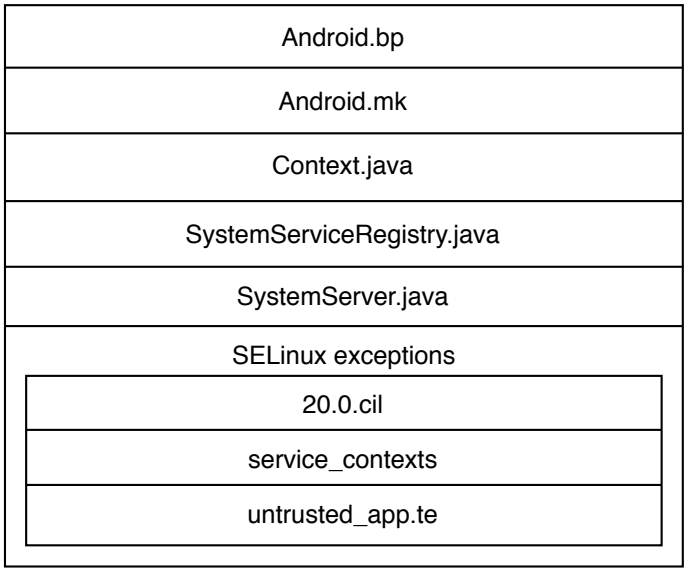


Figure 6.8: Files Modified to Install Role Manager Service as a System Service

Role Manager Service AIDL to this file. The entry that was added can be seen in Fig. 6.9.

Context.java We defined the Role Manager Service in this file, so that applications are able to request roles and activate them. The applications can access the service via the getSystemService method of the Activiy.java class by passing "Context.ROLE_MANAGER_SERVICE" to it.

SystemServiceRegistry.java Prior to the Role Manager Service being allowed to run as a system service, we need to let the system server know that it exists so that it can be launched at device boot with other system services. We do this using the "registerService" method in this file.

SystemServer.java Once the system knows that a service exists, we need to instruct it to launch the service. We achieve this, by adding the Role Manager Service entry with the "ServiceManager.addService()" method, inside the "startOtherServices" method, in this file.

SELinux exceptions Before any service is allowed to run, and other applications are allowed to discover and communicate with the service, we need to add exceptions for the service to the SELinux in Android. We do this by modifying the following files in the AOSP.

```
LOCAL_AIDL_INCLUDES := /home/<user>/<AOSP root>/frameworks/base/core/java/android/app/RMServiceManager.aidl
```

Figure 6.9: Entry for the RmServiceManager AIDL in the Android.mk file

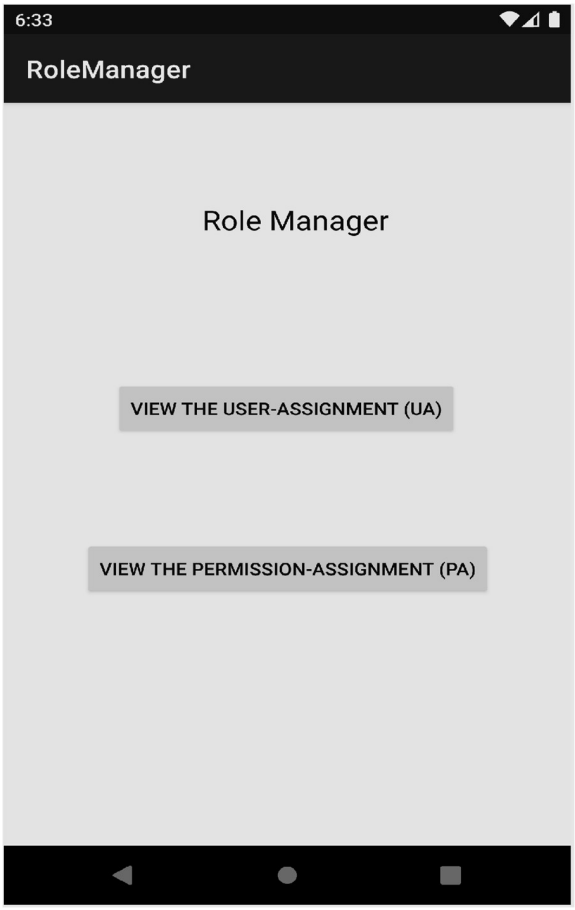
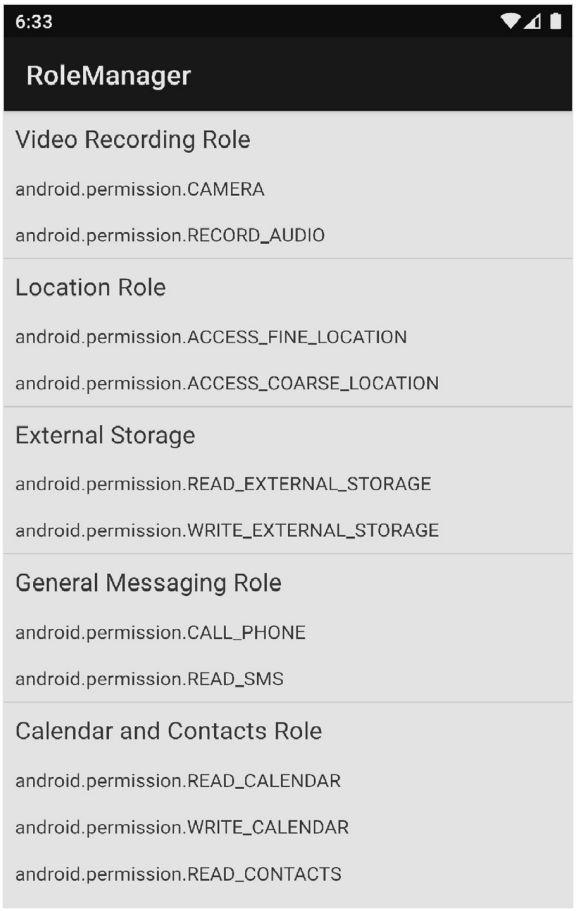
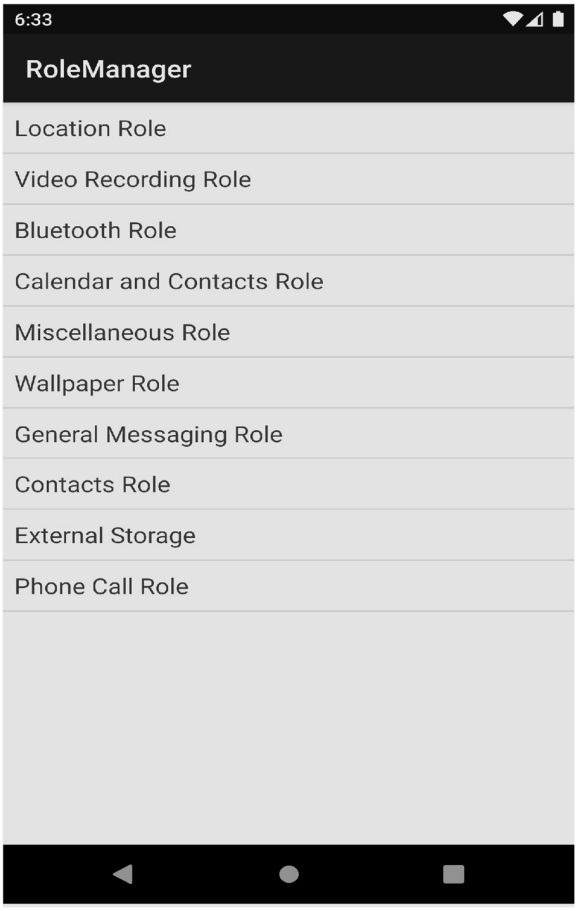


Figure 6.10: Role Manager Main Activity

- **29.0.cil**: We add our service to this file, to enable it to run as a system service in the background.
- **service_contexts**: This file is responsible for assigning a label to the Role Manager Service so that processes can add and find a binder reference for the service. Without this entry, third party applications cannot request roles since they cannot obtain a binder reference to the Role Manager Service.
- **untrusted_app.te**: We add a "find" rule to allow third party applications to find our service, so that they may request roles and role activations.



(a) Role Manager Permission Assignment Activity (PA)



(b) Role Manager User Assignment Activity (UA)

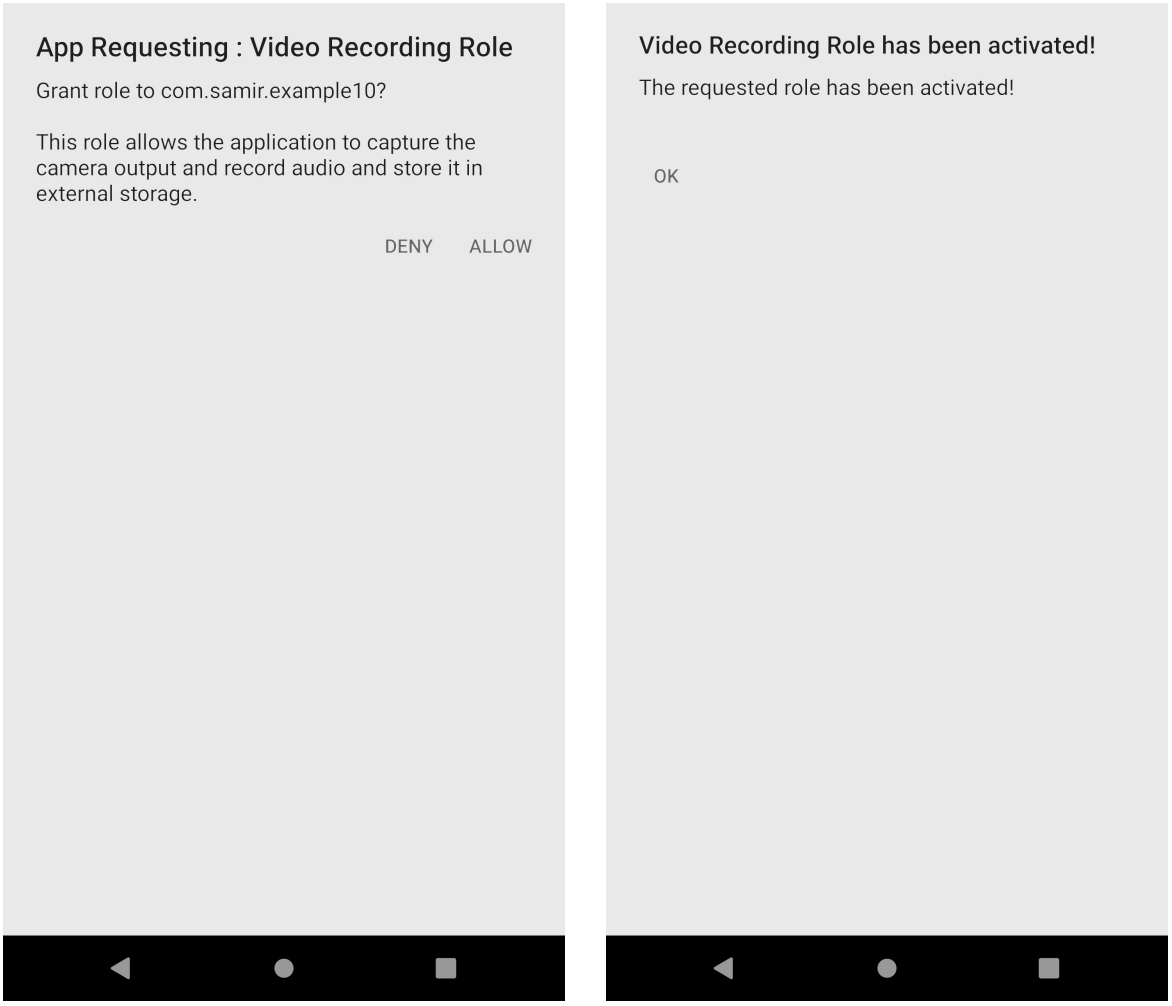
Figure 6.11: Role Manager UI

6.6 Implementation Demonstration

In this section, the implementation of RiA_a and **ARiA$_0$** is demonstrated. Screenshots are included for showing the Role Manager interface, and to indicate UI elements visible to the end user during their interaction with the modified Android OS.

6.6.1 Role Manager User Interface

The Role Manager system component has three parts, the main activity (see Figure 6.10) to choose which assignment to display (UA or PA), an activity for displaying the permission to role mapping (PA) (see Figure 6.11a) and an activity for displaying the role to application mapping (UA) (see



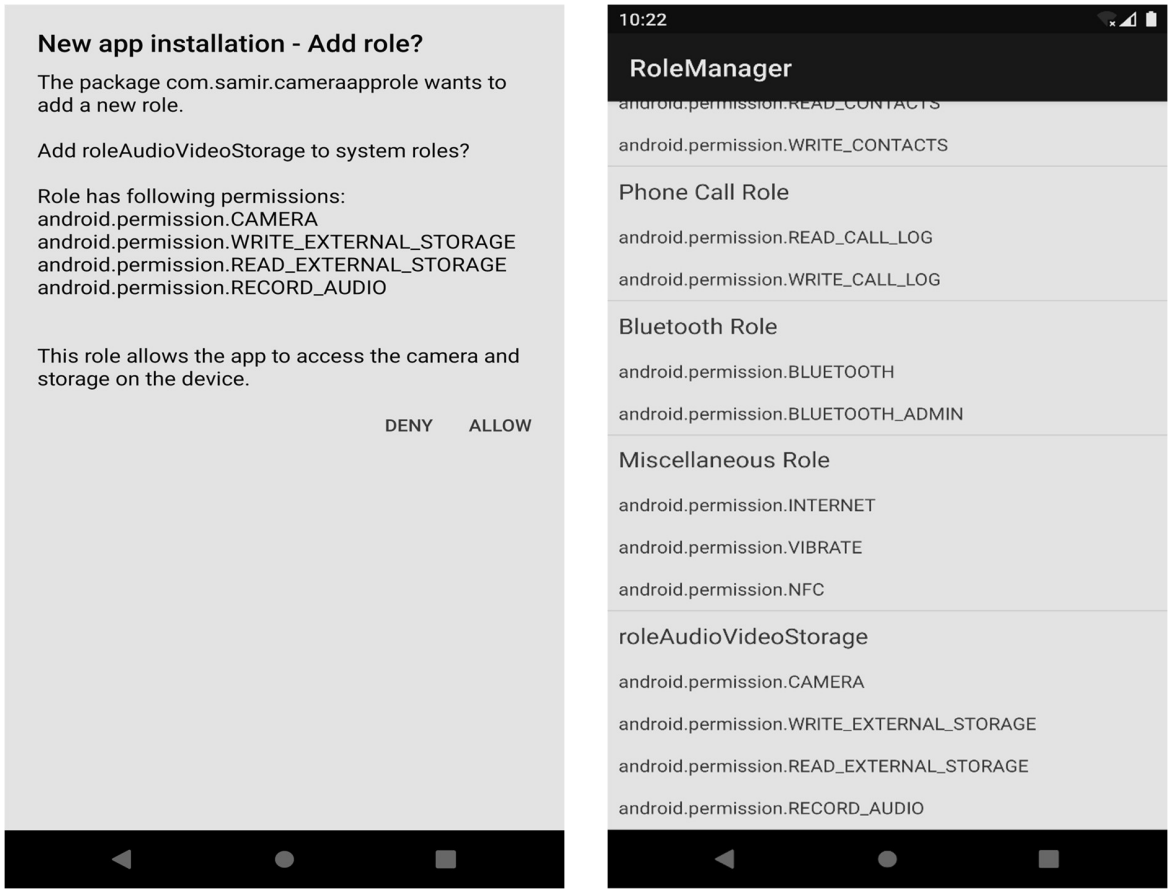
(a) Role Request Prompt (b) Role Activation Confirmation

Figure 6.12: Role Prompts Shown to the User

Figure 6.11b). When a new application is installed, and its request to add a new role to the PA is accepted by the user, this role gets populated in the PA activity of the Role Manager. Similarly, when a role is granted to an application, this assignment is populated in the UA activity of the Role Manager application.

6.6.2 User Prompt for Role Requests (AssignApp Operation)

When an application requests a role, and the system deems it necessary to seek user interaction, a role request prompt (see Figure 6.12a) is shown to the user. If the user accepts the request, by clicking on "ALLOW", the role gets assigned to that application.



(a) New Role Addition Prompt (b) New Role Added to the PA

Figure 6.13: New Role Prompt and Updated PA

6.6.3 Role Activation Confirmation Dialog (CreateSession Operation)

When an application requests to activate a role, the Role Manager activates that role for the app, and grants all associated permissions to the application. A role activation confirmation dialog is shown to the user, to indicate the role activation (see Figure 6.12b).

6.6.4 New Role Add Prompt (AssignPerm Operation)

Applications can add a new role-permission assignment to the PA, if, the user accepts the role addition prompt shown (see Figure 6.13a).

CHAPTER 7: CONCLUSION

We studied the Android's URI permissions and system permissions that revealed some peculiarities. Based on this, we built a model for access control in Android ($ACiA_{\alpha}$) and present it in brief, in this dissertation. Upon the analysis of this model, we discovered two significant issues within the permission system for Android, which were communicated to Google. Google has responded that one of these issues would be fixed in a later version of the Android OS. Some of the additional questions that could be answered from this work, are stated below.

- Given an app and a system permission it does not possess, is there a way in which that app receives that permission? Which ways?
- Given an app and a system permission it does possess, is there a way in which that permission is revoked from this app? Which ways?
- Given an app and its 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 applications, 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?

Post analyzing our ACiA model, we identified the need to explore a different approach for controlling access to resources in order to mitigate some of the discovered issues. RBAC offered potential advantages over the current access control model by being easy to configure for the end-user, and app developers; this enhanced Android's permission-based approach with a role based one. Prior to the implementation of RBAC in Android, we used role-mining algorithms from the literature to generate roles from the user permission assignment (UPA) matrix from the top 500 free apps from Google's Play Store. Analysis of the roles generated by these algorithms indicated that, the issue of administering permissions could be significantly remedied using these roles; the analysis has been described in this work. We implemented the RiA_a model in Android API 29, which was current at the time, and we have described this implementation in detail.

We have also presented several administrative models for RBAC in Android, to facilitate the user management of user-assignment (UA) and permission-assignment (PA) operations. Without a standardized PA, it was difficult and non-trivial to design an administrative model. To realize the administrative models, we used several techniques such as placing constraints on the UA and PA operations, and a RAdAC based administration of RBAC. The models presented aim to mitigate the user burden in the administration of RBAC in Android, by either automatically processing role-assignment and permission-assignment requests, or by placing constraints based on the cardinality, time and protection level of roles. A few example operations are also explained which describe the operation to assign a role to an application, and the operation to assign a permission to a role.

7.1 Future Work

In this work, we have analyzed roles generated from role-mining algorithms; it would be interesting to explore roles generated with the help of machine learning algorithms. Apart from this, a usability analysis of RBAC in Android, could uncover some potential drawbacks of the system, that could yield a better UI design. While RBAC in Android mitigates the issues we discovered, attribute based access control (ABAC) has been one of the most promising model for managing access

rights in recent times; it would be interesting to see how Android permissions can benefit from an attribute based access control system. It may also be potential to design and build an access control system for Android, with the help of deep learning. If devices can learn from user behavior, and adapt how they works, it could significantly enhance the usability of Android.

APPENDIX A: MINES ROLES

The output from the role-mining algorithm produce many roles, the top ten roles from each such algorithm are shown below.

Table A.1: FM/CM Mined Roles

Roles	Assigned permissions
R1	INTERNET, ACCESS_NETWORK_STATE
R2	com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, WAKE_LOCK
R3	INTERNET, READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, WAKE_LOCK
R4	INTERNET
R5	INTERNET, READ_PHONE_STATE, WRITE_EXTERNAL_STORAGE, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE
R6	READ_EXTERNAL_STORAGE, WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, WAKE_LOCK
R7	WRITE_EXTERNAL_STORAGE, com.google.android.c2dm. permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, WAKE_LOCK
R8	com.google.android.c2dm.permission.RECEIVE, INTERNET, ACCESS_NETWORK_STATE, WAKE_LOCK
R9	WRITE_EXTERNAL_STORAGE, INTERNET, ACCESS_NETWORK_STATE, WAKE_LOCK
R10	WRITE_EXTERNAL_STORAGE, INTERNET, ACCESS_NETWORK_STATE

Table A.2: Basic RMP Mined Roles

Roles	Assigned permissions
R1	android.permission.WRITE_CONTACTS, android.permission.GET_ACCOUNTS, android.permission.READ_CONTACTS,
R2	android.permission.WRITE_CONTACTS, android.permission.GET_ACCOUNTS, android.permission.READ_CONTACTS, android.permission.READ_CALL_LOG,
R3	android.permission.WRITE_CONTACTS,
R4	android.permission.GET_ACCOUNTS,
R5	android.permission.READ_CONTACTS,
R6	android.permission.READ_CALL_LOG,
R7	android.permission.ANSWER_PHONE_CALLS,
R8	android.permission.READ_PHONE_NUMBERS, android.permission.READ_PHONE_STATE,
R9	android.permission.READ_PHONE_STATE,
R10	android.permission.CALL_PHONE,

Table A.3: Delta RMP Mined Roles

Roles	Assigned permissions
R1	android.permission.WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.WAKE_LOCK,
R2	android.permission.READ_EXTERNAL_STORAGE, android.permission.WRITE_EXTERNAL_STORAGE, android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.VIBRATE, android.permission.ACCESS_WIFI_STATE,
R3	android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.ACCESS_WIFI_STATE,

Table A.4: MinNoise RMP Mined Roles

Roles	Assigned permissions
R1	android.permission.WRITE_EXTERNAL_STORAGE, com.google.android.c2dm.permission.RECEIVE, android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.WAKE_LOCK
R2	android.permission.READ_EXTERNAL_STORAGE, android.permission.WRITE_EXTERNAL_STORAGE, android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.VIBRATE, android.permission.ACCESS_WIFI_STATE
R3	android.permission.INTERNET, android.permission.ACCESS_NETWORK_STATE, android.permission.ACCESS_WIFI_STATE
R4	android.permission.GET ACCOUNTS, android.permission.READ PHONE STATE, android.permission.CAMERA, android.permission.ACCESS FINE LOCATION, android.permission.ACCESS COARSE LOCATION, android.permission.WRITE EXTERNAL STORAGE, com.google.android.c2dm.permission.RECEIVE, android.permission.INTERNET, android.permission.ACCESS NETWORK STATE, android.permission.VIBRATE, android.permission.WAKE LOCK

APPENDIX B: RBAC IN ANDROID - MODIFIED FILES

This is a list of Android framework files that were modified for the RBAC implementation.

Names of Files Modified	File path
29.0.cil	<AOSP root>/system/sepolicy/private/compat/29.0/
28.0.cil	<AOSP root>/system/sepolicy/private/compat/28.0/
27.0.cil	<AOSP root>/system/sepolicy/private/compat/27.0/
26.0.cil	<AOSP root>/system/sepolicy/private/compat/26.0/
Android.bp	<AOSP root>/frameworks/base/services/core/
Android.mk	<AOSP root>/frameworks/base/
android.test.mock.txt	<AOSP root>/prebuilts/sdk/29/system/api/
android.txt	<AOSP root>/prebuilts/sdk/29/public/api/
AlarmManagerService.java	<AOSP root>/frameworks/base/services/core/java/com/android/server
AndroidManifest.xml	<AOSP root>/frameworks/base/core/res
base_system.mk	<AOSP root>/build/make/target/product
BasePermission.java	<AOSP root>/frameworks/base/services/core/java/com/android/server/pm/permission
Context.java	<AOSP root>/frameworks/base/core/java/android/content
PackageManager.java	<AOSP root>/frameworks/base/core/java/android/content/pm
PackageManagerService.java	<AOSP root>/frameworks/base/services/core/java/com/android/server/pm
PermissionManagerService.java	<AOSP root>/frameworks/base/services/core/java/com/android/server/pm/permission
platform.xml	<AOSP root>/frameworks/base/data/etc
privapp-permissions-platform.xml	<AOSP root>/frameworks/base/data/etc
service.te	<AOSP root>/system/sepolicy/prebuilts/api/29.0/public
service.te	<AOSP root>/system/sepolicy/public
service_contexts.te	<AOSP root>/system/sepolicy/private
SystemServer.java	<AOSP root>/frameworks/base/services/java/com/android/server
SystemServiceRegistry.java	<AOSP root>/frameworks/base/core/java/android/app

BIBLIOGRAPHY

[1] How Game Apps That Captivate Kids Have Been Collecting Their Data. https://nyti.ms/2MpFuoG, 2018. [Online; accessed 12-March-2019].

[2] How Game Apps That Captivate Kids Have Been Collecting Their Data. https://nyti.ms/2MpFuoG, 2018. [Online; accessed 12-March-2019].

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

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

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

[6] Background Execution Limits | Android Developers. https://developer.android.com/about/versions/oreo/background, 2019. [Online; accessed 21-March-2019].

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

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

[9] J Abdella, Mustafa Özuysal, and Emrah Tomur. CA - ARBAC: Privacy preserving using context-aware role-based access control on Android permission system. *Security and Communication Networks*, 9(18):5977–5995, 2016.

[10] J Abdella, Mustafa Özuysal, and Emrah Tomur. Ca-arbac: Privacy preserving using context-aware role-based access control on Android permission system. *Security and Communication Networks*, 9(18):5977–5995, 2016.

[11] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the Android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[12] Hamid Bagheri, Eunsuk Kang, Sam Malek, and Daniel Jackson. A formal approach for detection of security flaws in the Android permission system. *Formal Aspects of Computing*, 30(5):525–544, sep 2018.

[13] Hamid Bagheri, Alireza Sadeghi, Joshua Garcia, and Sam Malek. COVERT: Compositional Analysis of Android Inter-App Permission Leakage. *IEEE Transactions on Software Engineering*, 41(9):866–886, sep 2015.

[14] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to Android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[15] Gustavo Betarte, Juan Campo, Carlos Luna, and Agustín Romano. Formal Analysis of Android's Permission-Based Security Model 1. *Scientific Annals of Computer Science*, 26(1):27–68, 2016.

[16] Gustavo Betarte, Juan Campo, Carlos Luna, Camila Sanz, Felipe Gorostiaga, and Maximiliano Cristiá. A formal approach for the verification of the permission-based security model of Android.

[17] Gustavo Betarte, Juan Diego Campo, Carlos Luna, and Agustín Romano. Verifying Android's Permission Model. pages 485–504. Springer, Cham, oct 2015.

[18] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, volume 17, page 19, 2012.

[19] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 131–146, 2013.

[20] Supriyo Chakraborty, Chenguang Shen, Kasturi Rangan Raghavan, Yasser Shoukry, Matt Millar, and Mani Srivastava. ipShield: A framework for enforcing context-aware privacy. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, pages 143–156, 2014.

[21] Edward J Coyne. Role engineering. In *Proceedings of the first ACM Workshop on Role-based access control*, pages 4–es, 1996.

[22] Jason Crampton. Administrative scope and role hierarchy operations. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 145–154, 2002.

[23] Jason Crampton. Understanding and developing role-based administrative models. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 158–167, 2005.

[24] Jason Crampton and George Loizou. Administrative scope: A foundation for role-based administrative models. *ACM Transactions on Information and System Security (TISSEC)*, 6(2):201–231, 2003.

[25] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *International Conference on Information Security*, pages 346–360. Springer, 2010.

[26] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, 2009.

[27] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android security. *IEEE security & privacy*, 7(1):50–57, 2009.

[28] Zheran Fang, Weili Han, and Yingjiu Li. Permission based Android security: Issues and countermeasures. *computers & security*, 43:205–218, 2014.

[29] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[30] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, page 3. ACM, 2012.

[31] David F Ferraiolo, Ravi Sandhu, Serban Gavrila, D Richard Kuhn, and Ramaswamy Chandramouli. Proposed NIST standard for role-based access control. *ACM Transactions on Information and System Security (TISSEC)*, 4(3):224–274, 2001.

[32] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing Android's permission system. In *European Symposium on Research in Computer Security*, pages 1–18. Springer, 2012.

[33] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and Enhancing Android's Permission System. pages 1–18. Springer, Berlin, Heidelberg, 2012.

[34] Mei Ge and Sylvia L Osborn. A design for parameterized roles. In *Research Directions in Data and Applications Security XVIII*, pages 251–264. Springer, 2004.

[35] Floris Geerts, Bart Goethals, and Taneli Mielikäinen. Tiling databases. In *International conference on discovery science*, pages 278–289. Springer, 2004.

[36] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, and Paquale Stirparo. A permission verification approach for Android mobile applications. *Computers & Security*, 49:192–205, 2015.

[37] Luigi Giuri and Pietro Iglio. Role templates for content-based access control. In *Proceedings of the second ACM workshop on Role-based access control*, pages 153–159, 1997.

[38] Google. Android open source project. `https://source.android.com/`, 2020.

[39] Michael C Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock Android smartphones. In *NDSS*, volume 14, page 19, 2012.

[40] Qi Guo. *A formal approach to the role mining problem*. PhD thesis, Rutgers University-Graduate School-Newark, 2010.

[41] Tao Guo, Puhan Zhang, Hongliang Liang, and Shuai Shao. Enforcing multiple security policies for Android system. In *2nd International Symposium on Computer, Communication, Control and Automation*. Atlantis Press, 2013.

[42] Google Inc. AAPT2 | Android developers. `https://developer.android.com/studio/command-line/aapt2`, 2019.

[43] Google Inc. Permissions overview | Android Developers. `https://developer.android.com/guide/topics/permissions/overview`, 2019.

[44] Google Inc. Request app permissions | Android developers. `https://developer.android.com/training/permissions/requesting/`, 2019.

[45] Ninghui Li and Ziqing Mao. Administration in role-based access control. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 127–138, 2007.

[46] Emil Lupu and Morris Sloman. Reconciling role based management and role based access control. In *Proceedings of the second ACM workshop on Role-based access control*, pages 135–141, 1997.

[47] R McGraw. Risk-adaptable access control (RAdAC). In *NIST Privilege (Access) Management Workshop*, 2009.

[48] Markus Miettinen, Stephan Heuser, Wiebke Kronz, Ahmad-Reza Sadeghi, and N Asokan. Conxsense: Automated context classification for context-aware access control. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 293–304, 2014.

[49] Sejong Oh and Ravi Sandhu. A model for role administration using organization structure. In *Proceedings of the seventh ACM symposium on Access control models and technologies*, pages 155–162, 2002.

[50] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in Android. *Security and Communication Networks*, 5(6):658–673, 2012.

[51] Mohammed Rangwala, Ping Zhang, Xukai Zou, and Feng Li. A taxonomy of privilege escalation attacks in Android applications. *International Journal of Security and Networks*, 9(1):40–55, 2014.

[52] Bingfei Ren, Chuanchang Liu, Bo Cheng, Shuangxi Hong, Jie Guo, and Junliang Chen. Easyprivacy: Context-aware resource usage control system for Android platform. *IEEE Access*, 6:44506–44518, 2018.

[53] Felix Rohrer, Yuting Zhang, Lou Chitkushev, and Tanya Zlateva. DR BACA: Dynamic role based access control for Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 299–308. ACM, 2013.

[54] Ahmad-Reza Sadeghi. Droidauditor: Forensic analysis of application-layer privilege escalation attacks on Android (short paper). In *Financial Cryptography and Data Security: 20th International Conference, FC 2016, Christ Church, Barbados, February 22–26, 2016, Revised Selected Papers*, volume 9603, page 260. Springer, 2017.

[55] Ravi Sandhu, Venkata Bhamidipati, Edward Coyne, Srinivas Ganta, and Charles Youman. The ARBAC97 model for role-based administration of roles: Preliminary description and outline. In *Proceedings of the second ACM workshop on Role-based access control*, pages 41–50, 1997.

[56] Ravi Sandhu, Venkata Bhamidipati, and Qamar Munawer. The ARBAC97 model for role-based administration of roles. *ACM Transactions on Information and System Security (TISSEC)*, 2(1):105–135, 1999.

[57] Ravi Sandhu and Qamar Munawer. The ARBAC99 model for administration of roles. In *Proceedings 15th Annual Computer Security Applications Conference (ACSAC'99)*, pages 229–238. IEEE, 1999.

[58] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, 29(2):38–47, 1996.

[59] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A comprehensive security assessment. *IEEE Security & Privacy*, 8(2):35–44, 2010.

[60] Wook Shin, Shinsaku Kiyomoto, Kazuhide Fukushima, and Toshiaki Tanaka. 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*, pages 944–951, 2010.

[61] StatCounter. Mobile operating system market share worldwide | statcounter global stats. https://gs.statcounter.com/os-market-share/mobile/worldwide, 2020.

[62] Samir Talegaon. Aosp RBAC. https://github.com/samtronxindia/RBAC-AOSP, 2020.

[63] Samir Talegaon and Ram Krishnan. Role based access control models for Android. In *2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*. IEEE, 2019.

[64] Samir Talegaon and Ram Krishnan. Administrative models for role based access control in Android. *Journal of Internet Services and Information Security (JISIS)*, 10(3):31–46, August 2020.

[65] Samir Talegaon and Ram Krishnan. A formal specification of access control in Android with URI permissions. *Information Systems Frontiers*, pages 0–0, 2020.

[66] Vincent F Taylor and Ivan Martinovic. Quantifying permission-creep in the Google Play Store. *arXiv preprint arXiv:1606.01708*, 2016.

[67] Constantin-Alexandru Tudorică and Laura Gheorghe. Context-aware security framework for Android. In *2016 International Workshop on Secure Internet of Things (SIoT)*, pages 11–19. IEEE, 2016.

[68] Guliz Seray Tuncay, Soteris Demetriou, Karan Ganju, and Carl A. Gunter. Resolving the Predicament of Android Custom Permissions. In *Proceedings 2018 Network and Distributed System Security Symposium*, Reston, VA, 2018. Internet Society.

[69] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: Finding a minimal descriptive set of roles. In *Proceedings of the 12th ACM symposium on Access control models and technologies*, pages 175–184. ACM, 2007.

[70] Jaideep Vaidya, Vijayalakshmi Atluri, and Qi Guo. The role mining problem: A formal perspective. *ACM Transactions on Information and System Security (TISSEC)*, 13(3):1–31, 2010.

[71] Jaideep Vaidya, Vijayalakshmi Atluri, Qi Guo, and Haibing Lu. Role mining in the presence of noise. In *IFIP Annual Conference on Data and Applications Security and Privacy*, pages 97–112. Springer, 2010.

[72] Jaideep Vaidya, Vijayalakshmi Atluri, and Janice Warner. Roleminer: Mining roles using subset enumeration. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 144–153. ACM, 2006.

[73] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing Android permission creep. In *Proceedings of the Web*, volume 2, pages 91–96, 2011.

[74] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in Android. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, pages 25–36, 2014.

[75] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the Android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2012.

[76] Sha Wu and Jiajia Liu. Overprivileged permission detection for Android applications. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE, 2019.

[77] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in Android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

VITA

Samir Talegaon was born in Mumbai, India in 1987. He received the bachelors degree (BE) in electronics from KIT's College of Engineering, Kolhapur. He received his 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, Android platform analysis and modification, and computer security.