2020 Second IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA) | 978-1-7281-8543-9/20/$31.00 ©2020 IEEE | DOI: 10.1109/TPS-ISA50397.2020.00033

Role-Based Access Control Models for Android

Samir Talegaon
Department of Electrical & Computer Engineering
University of Texas at San Antonio
San Antonio, USA
samir.talegaon@utsa.edu

Ram Krishnan
Department of Electrical & Computer Engineering
University of Texas at San Antonio
San Antonio, USA
ram.krishnan@utsa.edu

***Abstract*—Android uses runtime permissions to alert users of application resource usage. Only a limited portion of Android permissions are allowed to be managed by the users. This is made essential, because Android assigns permissions directly to applications, and the number of applications and permissions is high. However, due to this tradeoff, users are restricted from managing all the aspects of their own devices. Android itself groups permissions based on their functionality; however, these groups are immutable and non-overlapping, which confers a rigidity to the permission system. Prior work in adapting RBAC to Android exists but deviates from the standardized NIST RBAC and does not include sessions, a key component of RBAC, used to mitigate the exposure of system resources. So, to fully understand the benefits RBAC offers for Android, and to mitigate its permissions management problem, we propose three new models for RBAC in Android. Our models are aimed to address the issue of user permission management in conjunction with flexibility of being able to assign permissions to either users, applications, or app-components.**

***Index Terms*—role-based, access control, android**

I. Introduction

Android is one of the most popular mobile operating system, and Android users download apps such as WhatsApp, Facebook, YouTube, and Chrome to utilize the device's full potential. These apps require access to sensitive resources such as internet, contacts, camera, and microphone amongst others. The Android OS uses a permission-based mechanism to control access to sensitive resources that are available to apps[1], and despite advancements to access control in Android, this model has remained largely the same. According to this model, apps that require access to sensitive system resources, must request prior approval from the user (Figure 1). If the user approves such requests, permissions are permanently granted to apps and there is no mechanism by which permissions are automatically revoked.

Although the runtime permissions increase user awareness related to app-resource usage, the control users have over their own devices remains the same, because permissions categorized to be in the normal and signature protection level are restricted from being managed. Also, despite dangerous permissions being revocable, doing so is extremely tedious. Furthermore, to reduce the number of prompts to which users are expected to respond, these runtime permissions are grouped together based on their functionality, and the permissions within a group are collectively administered. Although this reduces the administrative burden on users, it also results in a reduced granularity since permissions within a group cannot be selectively granted or revoked. Also, the permission groups themselves are immutable and do not have any overlap, resulting in an inflexible permission management system.

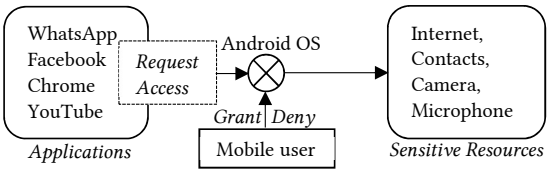


Figure 1. Permission-based access control in Android

Role based access control (RBAC) [8] is used in enterprise scenarios due to the administrative ease it provides, and we believe 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. Due to this, an RBAC based access control system is a promising approach towards developing an adaptive, user-administered access control mechanism, for Android.

There are a few works which have implemented RBAC in Android. Abdella et. al. [1] proposes a context-aware Android role-based access control (CA-ARBAC) in which they associate roles with users and contexts with permissions. However, this does not ease permission management for the users, nor does it enable the OS to differentiate between app components. Apart from this, roles are formed arbitrarily based on functional groups. DR-BACA [17] introduces static and dynamic RBAC in Android, and, uses 6-tuples called as rules to make access control decisions. While this removes the issue of permissions granted permanently to apps, it exacerbates the issue of user-management of permissions by increasing the complexity of the access control mechanism. Also, users in

This work is partially supported by NSF Grants CNS-1553696 and HRD-1736209.

[1]https://developer.android.com/training/permissions/requesting/

978-1-7281-8543-9/20/$31.00 ©2020 IEEE
DOI 10.1109/TPS-ISA50397.2020.00033

RBAC are substituted with apps in their work, and although this is similar to one of the models we propose, as will be discussed further, the goal of this work is to provide comprehensive models for RBAC in Android.

OUR CONTRIBUTION: We develop three new models for RBAC in Android, with distinctions in substitutions for the set of users in RBAC, with the set of users (RiA_u), apps (RiA_a) or app components (RiA_{ac}) in Android. For adapting the current Android to the Android with RBAC, we need to design roles. To this end, we demonstrate how roles can be engineered using a bottom-up approach via various algorithms from the literature. We also develop a prototype implementation of RBAC in Android with sessions, using the RiA_a model.

OUTLINE: Sect. II placed our work in the context of others in the area of RBAC in Android while Sect. III introduces a few key concepts in Android and describes the three models for RBAC in Android along with a brief description of the implementation of the RiA_a model in Android. In Sect. IV, we evaluate our RBAC models by providing suitable use case scenarios, and, by creating the required assignment relations by using role-mining algorithms techniques. Sect. V, describes the conclusion and provides directions for the future work.

II. Related Works

In this section, we discuss the prior works that target role-based access control in Android. Abdella et.al. [1] implemented RBAC in Android, by assigning roles to permissions and granting these roles to apps. They use context 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, their work does not include the use of sessions, a critical component of RBAC. Also, roles are arbitrarily created in their model. We believe that the creation of roles is an important step to fully realize the benefits of RBAC in Android, since, arbitrarily created roles hamper the effective advantage gained by using RBAC in Android.

Rohrer [17] 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 and it is for this reason we propose a hitherto un-implemented RBAC model for improving security and privacy in the Android OS. MPDroid [12] presents a custom, user-definable access control system that can be used to control perms granted to untrusted apps with a high level of granularity. It allows device administrators to limit the damage apps can do (using MAC in the kernel layer), and users to control app-access to resources (using RBAC in the framework layer). However, they require users to create roles, and since users are not experts in the field of access control, they cannot be expected to create security aware roles. Apart from this, their RBAC model does not include sessions, a key concept that allows us to combat conflict of interest issues.

Several works have proposed access control systems for Android using context-aware policies [18]. ipShield [4] draws inferences based on sensor data to make access control suggestions to the users. ConXsense [15] uses context sensing and machine learning in their framework to probabilistically make an access control decision. Ren et. al. [16] propose a context-aware resource usage control system that includes four policy conflict resolution rules and implement it in Android (named EasyPrivacy).

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 [2] analyses the Android permission system in a bid to provide insight in to Android's API permissions, statistically. Barrera et. al. [3] 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 [7] 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 into the Android's permission system, however they do not propose any new access control methods, and instead suggest improvements on the Android's permission-based access control mechanism.

VetDroid [22] 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. [6] 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; however, they do not attempt to modify the permission-based access control mechanism.

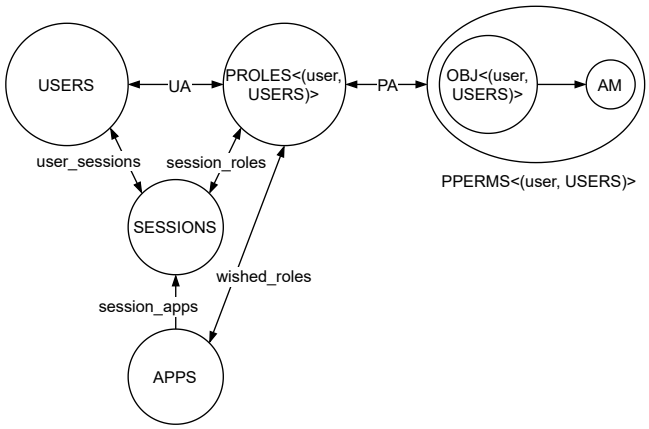
III. RBAC in Android

Before proceeding further, a few concepts in Android, its apps and RBAC require explanation, so that our work can be better understood.

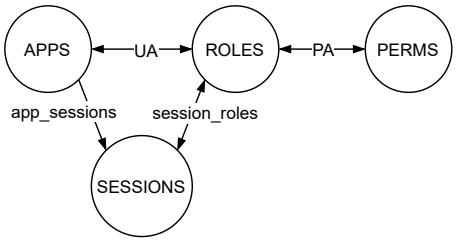
A. Background

Android app components. Android apps consist of four main components[2] - Activities, Services, Broadcast Receivers and Content Providers. Activities make up the GUI, which appears on the phone screen and interacts with the users; services are the invisible components which run in the background and are meant to perform tasks which take a significant amount of

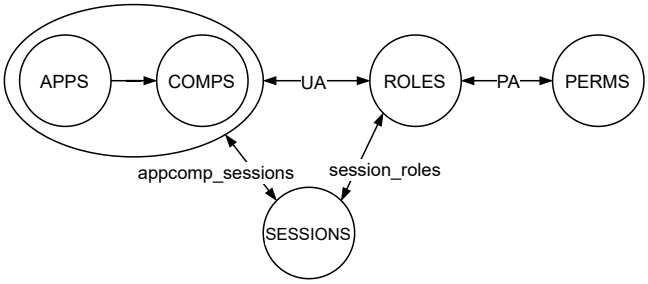
[2]https://developer.android.com/guide/components/fundamentals



(a) RBAC Users substituted with Android Users(RiA_u)



(b) RBAC Users substituted with Android Applications(RiA_a)



(c) RBAC Users substituted with Android App-components (RiA_{ac})

Figure 2. RBAC in Android (RiA) models with the set of Users in RBAC substituted as denoted

time to process. Broadcast receivers can receive system wide broadcasts indicating specific events such as, the completed boot-up of a device, and content providers are the mechanism which enable apps to share their data in a controlled manner, with other apps.

Android permissions. Permissions are categorized[3] as normal, dangerous and signature depending on the risk[4] they pose to the user's privacy and security. Normal permissions are automatically granted, and signature permissions are granted by the Android OS based on the signing certificate of the app, 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.

RBAC. There are 3 different models of RBAC: core RBAC, hierarchical RBAC and constrained RBAC. While core RBAC provides the basic entities for an RBAC system, hierarchical RBAC optimizes the PA and UA relationships by adding role hierarchies, which involve senior and junior roles. The third RBAC model adds separation of duty constraints that provides protection against conflict of interest.

[3]https://developer.android.com/guide/topics/permissions/overview

[4]We believe that Google evaluates risk with two parameters, namely, access to user-data, and control over the device that can negatively impact user experience.

B. RBAC Models for Android

In this subsection 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.

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 [9], [10], [13], [14]. As we can see from Fig.2a, 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 administrator for the device. Once users receive the required roles, they can launch new sessions when needed, and can

Table I
RiA$_u$ ENTITY SETS, RELATIONS AND FUNCTIONS

Entity Sets
USERS and APPS the sets of all users and apps 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)$\rangle$ = $2^{\text{OBJ}\langle\text{(user, USERS)}\rangle \times \text{AM}}$, 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) = {$pr\langle$(user, u)$\rangle$ $\in$ PROLES$\langle$(user, u)$\rangle$ $\mid$ (u, $pr\langle$(user, u)$\rangle$) $\in$ UA}
assigned_users: PROLES$\langle$(user, USERS)$\rangle$ $\rightarrow$ 2^{USERS}, the mapping of a parameterized role pr$\langle$(user, u)$\rangle$:PROLES$\langle$(user, USERS)$\rangle$ onto a set of users that it has been assigned to. Formally, assigned_users($pr\langle$(user, u)$\rangle$)$\mid_{u \in \text{USERS}}$ = {u $\mid$ (u, $pr\langle$(user, u)$\rangle$) $\in$ UA}
assigned_ppermissions: PROLES$\langle$(user, USERS)$\rangle$ $\rightarrow$ $2^{\text{PPERMS}\langle\text{(user, USERS)}\rangle}$, the mapping of pr$\langle$(user, u)$\rangle$:PROLES$\langle$(user, USERS)$\rangle$ onto a set of parameterized permissions for a particular user u:USERS. Formally, assigned_ppermissions($pr\langle$(user, u)$\rangle$) = {$pp\langle$(user, u)$\rangle$ $\in$ PPERMS$\langle$(user, USERS)$\rangle$ $\mid$ ($pp\langle$(user, u)$\rangle$, $pr\langle$(user, u)$\rangle$) $\in$ PA}
user_sessions: USERS $\rightarrow$ 2^{SESSIONS}, the mapping of user u onto a set of sessions. Note that, $\forall u_1 \neq u_2 \in$ USERS. user_sessions(u_1) $\cap$ user_sessions(u_2) = $\emptyset$
session_apps: SESSIONS $\rightarrow$ 2^{APPS}, the mapping of session s onto a set of apps.
session_proles: 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) = {$pr\langle$(user, u)$\rangle$ $\in$ PROLES$\langle$(user, u)$\rangle$ $\mid$ $u \in$ USERS $\wedge$ (session_users(s_i), $pr\langle$(user, u)$\rangle$) $\in$ UA}. Note that, $\forall s_1 \neq s_2 \in$ SESSIONS. session_proles(s_1) $\neq$ session_proles(s_2)
avail_session_pperms: SESSIONS $\rightarrow$ $2^{\text{PERMS}\langle\text{(user, USERS)}\rangle}$, the permissions available to a user in a session, $\bigcup_{pr\langle\text{(user, } u\text{)}\rangle \in \text{session\_proles(s)}}$ assigned_ppermissions($pr\langle$(user, u)$\rangle$).
wished_proles: 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.

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$\langle(P_{name},\ P_{domain})\rangle$. 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 I, and the key operations for this model can be found in the Table II.

2) RiA$_a$ *(Users in RBAC Replaced with Apps in Android):* In this model, users in RBAC are substituted with apps in Android (see Fig.2b). This model dictates entrusting decision

Table II
RiA$_u$ OPERATIONS

Operation: **CreateSession**(u : USERS, a : APPS, $ars : 2^{\text{PROLES}\langle(\text{user},\ u)\rangle}$, s : NAME)
Authorization Requirement: $ars \subseteq \texttt{user\_proles}(u) \wedge ars \subseteq \texttt{wished\_proles}(a) \wedge s \notin \text{SESSIONS}$
Updates:
$\text{SESSIONS}' = \text{SESSIONS} \cup \{s\}$
$\texttt{user\_sessions}' = \texttt{user\_sessions} \cup \{u,\ s\}$
$\texttt{session\_proles}' = \texttt{session\_proles} \cup \{s\} \times ars$
Operation: **DeleteSession**(u : USERS, s : SESSIONS)
Authorization Requirement: $\texttt{sessUser}(s) = u$
Updates:
$\texttt{user\_sessions}' = \texttt{user\_sessions} \setminus \{u,\ s\}$
$\texttt{session\_proles}' = \texttt{session\_proles} \cup \{s\} \times ars$
$\text{SESSIONS}' = \text{SESSIONS} \setminus \{s\}$
Operation: **RequestAccess**$\big(a$: APPS, u : USERS, $pr\langle(\text{user},\ u)\rangle : \text{PROLES}\langle(\text{user},\ u)\rangle\big)$
Authorization Requirement: $\exists s \in \text{SESSIONS}.\ a \in \texttt{session\_apps}(s) \wedge \texttt{u} = \texttt{session\_users}(s) \wedge \big(u,\ pr\langle(\text{user},\ u)\rangle\big) \in \text{UA} \wedge pr\langle(\text{user},\ u)\rangle \notin \texttt{session\_proles(s)}$
Updates:
$\texttt{session\_proles}' = \texttt{session\_proles} \cup \{s,\ pr\langle(\text{user},\ u)\rangle\}$
Operation: **RevokeRole**$\big(u$: USERS, s : SESSIONS, $pr\langle(\text{user},\ u)\rangle : \text{PROLES}\langle(\text{user},\ u)\rangle\big)$
Authorization Requirement: $s \in \texttt{user\_sessions}(u)$
Updates:
$\texttt{session\_proles}' = \texttt{session\_proles} \setminus \{s,\ pr\langle(\text{user},\ u)\rangle\}$
Operation: **CheckAccess**(a : APPS, s : SESSIONS, $pp\langle(\text{user},\ u)\rangle : \text{PPERMS}\langle(\text{user},\ u)\rangle$, $outresult$: BOOLEAN)
Authorization Requirement: $\exists pr\langle(\text{user},\ u)\rangle \in \text{PROLES}\langle(\text{user},\ u)\rangle.\ u = \texttt{session\_users}(s) \wedge pr\langle(\text{user},\ u)\rangle \in \texttt{session\_proles}(s) \wedge \big(pr\langle(\text{user},\ u)\rangle,\ pp\langle(\text{user},\ u)\rangle\big) \in \text{PA}$
Updates:
-

Table III
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.

policies such as what roles to activate and when, with the apps 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 apps, the administrative burden of users is reduced without a disproportionate increase in the number of user prompts[5].

Design. Since apps 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 apps. These roles are generated by top-down (semantic meaning) and bottom-up (algorithms for mining roles) approaches [5] using apps from the Play store and the information on which permissions are requested by them. Although apps 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, apps 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.

RiA$_a$ *Model.* The model for RiA$_a$ is described below along with the entity sets, relations and functions in Table III, and, its key operations in Table IV.

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

Table IV
RiA$_a$ OPERATIONS

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

Table V
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.
PA $\subseteq$ PERMS $\times$ ROLES, a many-to-many mapping permission-to-role assignment relation.
Functions
assigned_comps: ROLES $\nrightarrow 2^{\text{APP\_COMPS}}$, the mapping of role *r*:ROLES onto a set of app_components. Formally, assigned_comps(*r*) = {*ac* $\in$ APP_COMPS | (*ac, r*) $\in$ UA}.
appcomp_roles: APP_COMPS $\rightarrow 2^{\text{ROLES}}$, the mapping of app-comp *ac*:APP_COMPS onto a set of roles. Formally, appcomp_roles(*ac*) = { *r* $\in$ ROLES | (*ac, r*) $\in$ UA}.
assigned_permissions:ROLES $\rightarrow 2^{\text{PERMS}}$, the mapping of *r*:ROLES onto a set of permissions. Formally, assigned_permissions(*r*) = {*p* $\in$ PERMS | (*p, r*) $\in$ PA}.
APPCOMP_SESSIONS $\subseteq$ APP_COMPS $\times$ SESSIONS, a many-to-many mapping app_components - to - session assignment relation.
sess_appcomp: APP_COMPS $\nrightarrow 2^{\text{SESSIONS}}$, the mapping of *ac*:APP_COMPS onto a set of sessions. Formally, sess_appcomp(*ac*) = {*s* $\in$ SESSIONS | (*ac, s*) $\in$ APP-COMP_SESSIONS}.
appcomp_session: SESSIONS $\rightarrow$ APP_COMPS, the mapping of *s*:SESSIONS to an app_component *ac*. Note that, $\forall s \in$ SESSIONS. (appcomp_session(s), s) $\in$ COMP_SESSIONS.
session_roles $\subseteq$ SESSIONS $\times$ ROLES, a many-to-many mapping session-to-roles assignment relation.
session_roles: SESSIONS $\rightarrow 2^{\text{ROLES}}$, the mapping of *s*:SESSIONS onto a set of roles. Formally, session_roles(*s*) = {*r* $\in$ ROLES | (appcomp_session(*s*), *r*) $\in$ UA}.
avail_session_perms: SESSIONS $\nrightarrow 2^{\text{PERMS}}$, the permissions available to a component in a session, $\bigcup_{r \in \texttt{session\_roles}(s)}$ 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, $\forall s \in$ SESSIONS, $\forall p \in$ PERMS. isAuthorized(s, p) $\rightarrow \exists r \in$ ROLES. $((s, r) \in \texttt{session\_roles} \wedge (p, r) \in \texttt{PA})$
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.

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.2c). 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 apps. 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

Table VI
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* = appcomp_session(*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* = appcomp_session(*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* = appcomp_session(*s*) $\wedge$ $r \in$ session_roles(*s*) Updates: $\texttt{session\_roles}' = \texttt{session\_roles} \setminus \{(s,\ r)\}$
Operation: **CheckAccess**(ac : APP_COMPS, s : SESSIONS, p : PERMS, $outresult$: BOOLEAN) Authorization Requirement: $\exists r \in \mathrm{ROLES}.\ ac = \texttt{appcomp\_session}(s) \wedge r \in \texttt{session\_roles}(s) \wedge (r,\ p) \in \mathrm{PA}$ Updates: -

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.

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 V, and the key operations for this model can be found in Table VI.

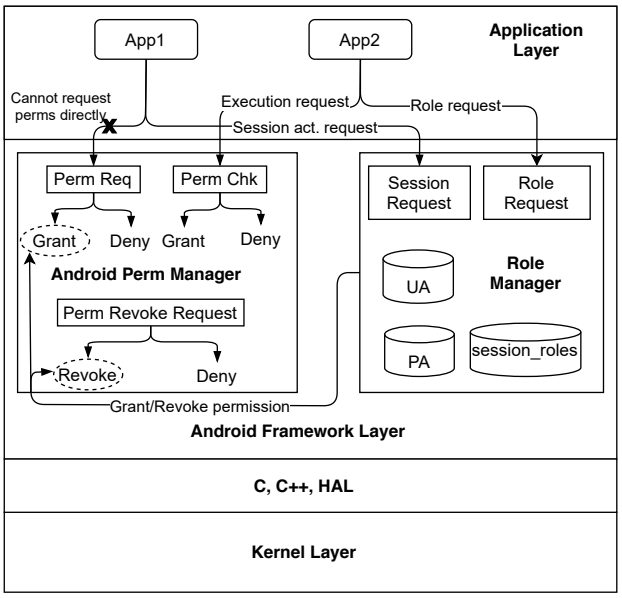


Figure 3. RiA_a implementation

C. Implementation

In this subsection, we briefly describe our implementation for RBAC in Android. We compiled the source code for API 29 (current at this time) and modified the source code for the same branch. The model RiA_a was implemented in Android and is described below (see Figure 3).

Implementation Methodology This implementation for RBAC in Android is built within the Android framework later, and utilizes Android's internal perm checking mechanism to allow or deny apps from accessing the resources. The role manager, a module built by us, processes incoming role requests, and maintains an updated UA, PA, and `session_roles`. A brief summary of the key modifications done to Android are described below.

Package Installer The package installer is responsible for streaming apps from the Google Play Store and installing them on the device. It is during the installation that all the normal and signature permissions are auto granted to apps. We modified the package installer to read external XML files to input roles defined by developers for their own apps. The installation only succeeds if such a role is not already defined on the device, or, if it was defined by the same developer (via signature match). This is achieved by modifying the InstallSucceed.java file and adding an exception to the final method for returning a successful installation.

The Android Manifest File for the Platform The AndroidManifest.xml file for the platform contained in /frameworks/base/core/res contains all the permissions definitions for

that device, and we removed the permission-group associations for each of the permissions defined in this file, since permission-groups are not used by our mechanism.

Permission Manager The requestPermission method from ActivityCompat.java and from the Activity.java files were overridden, and apps were prevented from being able request permissions, since apps are only be allowed to request roles and not individual permissions in RiA_a. We added the methods requestRole in the above-mentioned files to enable apps to request the appropriate roles, after they have been requested in the app's manifest file (AndroidManifest.xml).

Role Manager We developed a new module at the framework level to manage all role to permission assignments as well as role to app assignments (see Fig.3). This module hooks onto the permission manager to grant and revoke permissions according corresponding to role grants and revocations. When a role is granted, and activated by an app with the activateRole method, this module grants all the permissions for that role to the app. When the app shuts down, all those permissions are revoked. It should be noted that we modified the Android manifest file for the platform to remove all permission group assignments and all permissions are now individually granted or revoked.

IV. Evaluation

In this section, we evaluate the RBAC models for Android. We do this by providing plausible scenarios for use cases pertaining to each model. Also, we utilize role-mining, to algorithmically generate roles (i.e.: the permission assignment (PA) and the user-assignment (UA)). By providing example mapping of roles to permissions and applications, we show that it is not only feasible to utilize RBAC's true potential in easing administrative burden, but also that doing so does not over burden the user with role prompts instead of permission prompts.

A. Use Cases for RBAC Models for Android

Use Cases for RiA_u This model is useful in many scenarios where the Android devices are shared by the users. In a car, the control system can be designed with RiA_u, where the owner would get access to all the car's system functions from setting the tire pressure to modifying the alarm distance for the proximity sensors. Access for other drivers can be limited to adjusting the seat height and mirrors amongst other non-critical components that are required for safely driving the car. RiA_u can also be employed where Android devices are used in an attendance system. In an organization, the managers for the departments can receive a permission to read all attendance whereas the employees can only log and read their own attendance. Finally, at home, RiA_u can be used to share an Android device with the family, such as in a smart display system, which can control all smart devices within the household and initiate communication with the outside world (placing phone calls). Each member of the house can read their own contacts, photos, videos specific to them and can choose to share these with the rest of the household, while senior members can obtain access to modify critical settings such as thermostats, door alarms and security cameras.

Use Cases for RiA_a Since apps in Android are the subjects, this model is useful in varying special scenarios including general phone and tablet use. In the enterprise scenario, the devices used by employees can be automatically configured to obtain the required roles based on the employee's job title in the company. Devices used by managers can be programmed to receive roles to check on employee attendance, performance, and approve or decline leave requests. Whereas devices used by employees can be restricted from activating certain features such as microphone or camera, and apps can be limited that reduce productivity in the workplace.

Use Cases for RiA_{ac} Due to the inherently complex nature of this model, its use is limited to areas of high risk such as the military, banking sector and sensitive fields such as nuclear or cyber-security research. All the apps that can run on the Android built around this model need to be developed in-house, due to the major change in the subjects, and would be incompatible with other versions of the RiA. The roles would be managed by members of the administrative team, experienced in access control and users would have little to no control in the administration of roles. In order to realize the full potential of RBAC in Android, the formation of roles is explained in the next section.

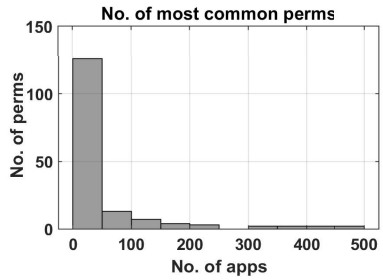
B. Role Mining for Generating UA and PA

Role mining is a bottom up approach [21] of role engineering [5], 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 analyzed and implemented five such algorithms i.e.: Fast Miner/ Complete Miner [21], Basic RMP [19], Delta RMP [20] and the Min Noise RMP algorithm [11]. It should be noted that, the issue of engineering good roles has been studied extensively, and is outside the scope of this paper; we present an analysis of these algorithms, as an illustration to the suitability of the generated roles for Android. This does not confer a finality to this work, and roles need to be engineered, according to the requirements of each system. For Android, these algorithms provide a brief outlook of the nature of generated roles, and we utilize these roles in our implementation of RBAC in Android.

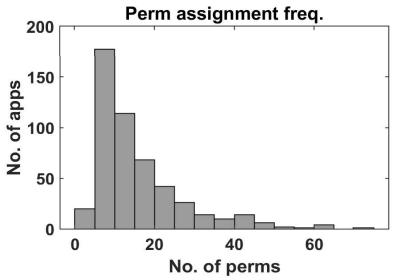
The purpose of mining roles, is to provide a few basic roles which can be pre-included with the RBAC in Android system, and it is not our intention to assign all of Android's perms (582 perms in API 29) to roles. It is shown below, that even when we consider as low as 10 roles, a significant number of 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. After prefacing this, the results from our role mining for Android are described in brief.

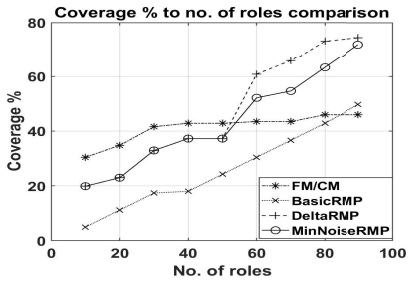
The above-mentioned algorithms are run on our data set consisting of top 500 free apps from the Google Play store (obtained from APK Pure[6]). While the total number of per-

[6]https://apkpure.com/

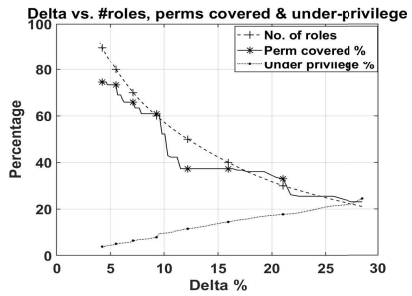


(a) No. of perm assignments reqd. in stock Android

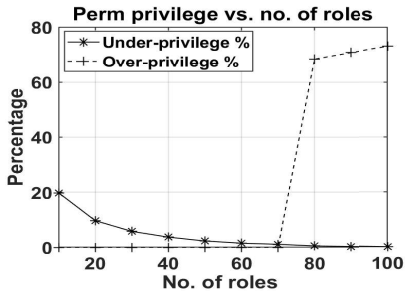




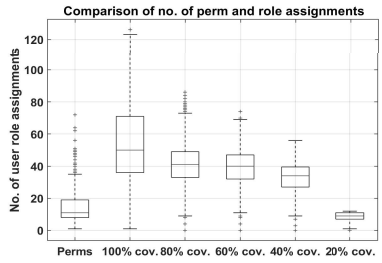
(b) Cvrg. of role mining alg.



(c) Delta RMP perm privilege



(d) MinNoise RMP perm privilege



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

Figure 4. Results from role mining for Android

missions 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(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(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(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 [20]. 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(d) 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 percentage above 120 mined roles which is noteworthy.

Finally, the Fig.4(e) 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. From Fig. 4(b),(d) and (e), 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 Fig.4(b)), and a corresponding under-privilege of 20% (4(d)). 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. A few sample roles generated by the MinNoise RMP algorithm are shown in Table VII. It should be noted that these algorithms also generate the UA, however it is not shown here for brevity.

Table VII
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

V. CONCLUSION AND FUTURE WORK

In this paper, we propose three new models for RBAC in Android, aimed at enhancing users' capabilities to manage Android permissions. Our models grant flexibility to users in regulating app-resource access, by enabling management of all Android permissions. It also improves the accessibility of Android permissions for app developers, by enabling them to define new roles that support overlap. To show the practicality of RBAC in Android, we analyzed and implemented several role mining algorithms, to generate the UA and the PA. We also run these algorithms on the UPA matrix generated from the top 500 apps in the Play store, and the generated assignment matrices show positive results with a mere 10 roles required to keep the under-privilege percentage at 20%. We also implemented one of the proposed models in the Android API-29, in the Framework layer by leveraging Android's internet permission checking mechanism. Future work includes analysis of the implementation for RiA_a with three distinct directives of security, usability, and performance. Finally, our models can be extended for application in hierarchical RBAC and constrained RBAC for Android, granting even more administrative power in terms of role hierarchies and separation of duty constraints.

ACKNOWLEDGEMENTS

This work is partially supported by NSF Grants CNS-1553696 and HRD-1736209.

REFERENCES

[1] Abdella, J., Özuysal, M., Tomur, E.: Ca-arbac: privacy preserving using context-aware role-based access control on android permission system. Security and Communication Networks **9**(18), 5977–5995 (2016)
[2] Au, K.W.Y., Zhou, Y.F., Huang, Z., Lie, D.: Pscout: analyzing the android permission specification. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 217–228. ACM (2012)
[3] Barrera, D., Kayacik, H.G., Van Oorschot, P.C., Somayaji, A.: 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. pp. 73–84. ACM (2010)
[4] Chakraborty, S., Shen, C., Raghavan, K.R., Shoukry, Y., Millar, M., Srivastava, M.: ipShield: A framework for enforcing context-aware privacy. In: 11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14). pp. 143–156 (2014)
[5] Coyne, E.J.: Role engineering. In: Proceedings of the first ACM Workshop on Role-based access control. pp. 4–es (1996)
[6] Fang, Z., Han, W., Li, Y.: Permission based android security: Issues and countermeasures. computers & security **43**, 205–218 (2014)
[7] Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 627–638. ACM (2011)
[8] Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed nist standard for role-based access control. ACM Transactions on Information and System Security (TISSEC) **4**(3), 224–274 (2001)
[9] Ge, M., Osborn, S.L.: A design for parameterized roles. In: Research Directions in Data and Applications Security XVIII, pp. 251–264. Springer (2004)
[10] Giuri, L., Iglio, P.: Role templates for content-based access control. In: Proceedings of the second ACM workshop on Role-based access control. pp. 153–159 (1997)
[11] Guo, Q.: A formal approach to the role mining problem. Ph.D. thesis, Rutgers University-Graduate School-Newark (2010)
[12] Guo, T., Zhang, P., Liang, H., Shao, S.: Enforcing multiple security policies for android system. In: 2nd International Symposium on Computer, Communication, Control and Automation. Atlantis Press (2013)
[13] Li, N., Mao, Z.: Administration in role-based access control. In: Proceedings of the 2nd ACM symposium on Information, computer and communications security. pp. 127–138 (2007)
[14] Lupu, E., Sloman, M.: Reconciling role based management and role based access control. In: Proceedings of the second ACM workshop on Role-based access control. pp. 135–141 (1997)
[15] Miettinen, M., Heuser, S., Kronz, W., Sadeghi, A.R., Asokan, N.: Conxsense: automated context classification for context-aware access control. In: Proceedings of the 9th ACM symposium on Information, computer and communications security. pp. 293–304 (2014)
[16] Ren, B., Liu, C., Cheng, B., Hong, S., Guo, J., Chen, J.: Easyprivacy: Context-aware resource usage control system for android platform. IEEE Access **6**, 44506–44518 (2018)
[17] Rohrer, F., Zhang, Y., Chitkushev, L., Zlateva, T.: Dr baca: dynamic role based access control for android. In: Proceedings of the 29th Annual Computer Security Applications Conference. pp. 299–308. ACM (2013)
[18] Tudorică, C.A., Gheorghe, L.: Context-aware security framework for android. In: 2016 International Workshop on Secure Internet of Things (SIoT). pp. 11–19. IEEE (2016)
[19] Vaidya, J., Atluri, V., Guo, Q.: The role mining problem: finding a minimal descriptive set of roles. In: Proceedings of the 12th ACM symposium on Access control models and technologies. pp. 175–184 (2007)
[20] Vaidya, J., Atluri, V., Guo, Q.: The role mining problem: A formal perspective. ACM Transactions on Information and System Security (TISSEC) **13**(3), 1–31 (2010)
[21] Vaidya, J., Atluri, V., Warner, J.: Roleminer: mining roles using subset enumeration. In: Proceedings of the 13th ACM conference on Computer and communications security. pp. 144–153 (2006)
[22] Zhang, Y., Yang, M., Xu, B., Yang, Z., Gu, G., Ning, P., Wang, X.S., Zang, B.: Vetting undesirable behaviors in android apps with permission use analysis. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 611–622. ACM (2013)