Administrative Models for Role Based Access Control in Android

Samir Talegaon* and Ram Krishnan
The University of Texas at San Antonio, San Antonio, Texas 78249 USA
{samir.talegaon, ram.krishnan}@utsa.edu

Abstract

Prior works propose new models for role based access control (RBAC) in Android; this paper adds on to that body of research. Despite RBAC's inherent administrative ease, managing roles for Android applications is tedious for the device user, owing to their lack of knowledge in access control. To realize the full potential of RBAC and to equip the user with ability to effectively manage Android permissions, we introduce three new models for administration of RBAC in Android. These models are based on an in-depth analysis of applications in Android, and support the principle of least privilege to reduce unwanted permission exposure.

Keywords: Role based access control, Android, access control, administration

1 Introduction

Android is one of the most widely used mobile operating systems, and uses a permission based access control system. Owing to the issues such as tedious administration mechanism of dangerous permissions, absence of user control over normal permissions and the perpetually increasing number of permissions and applications that can be installed at any time on modern devices, previous work has explored a role based access control (RBAC) in Android [1, 11]. Given RBAC's administrative ease, these works have shown a promising approach towards managing application to resource access.

Analyzing a standardized RBAC system on Android, with users substituted for applications (see Fig. 1), gives us an insight into the issues that could be faced in the administration of such a system. Since a clear understanding of the constituencies of roles (i.e.: the permission that are assigned to the roles (PA)) does not yet exist, 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). On a side note, for this paper, it is assumed that the user of the Android device, the application developers and Google are responsible for administering the UA and PA. Also, roles are assumed to be defined by the application developers, that consist of role identities and their permission assignments (PA).

In this paper, we propose three new administrative models for RBAC in Android. 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 [5], 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 ensuring administrative ease, is with an automated access control system such as RAdAC [8]. 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 operations under consideration pose, and make access control decisions based on this risk.

Journal of Internet Services and Information Security (JISIS), volume: 10, number: 3 (August 2020), pp. 31-46
DOI: *10.22667/JISIS.2020.08.31.031*

*Corresponding author: Electrical and Computer Engineering Department, The University of Texas at San Antonio, One UTSA Circle, San Antonio, Texas, 78249, USA, Tel: +1-210-660-8859

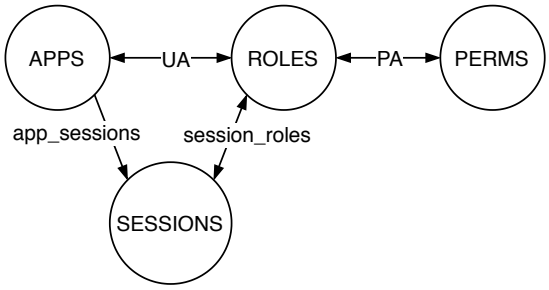


Figure 1: RBAC in Android

These models target the administration of UA and PA, and can mitigate user burden in the administration of RBAC in Android.

Our Contribution: We propose three new models of administration of RBAC in Android, the first model employs a user prompt based administration technique for UA and PA operations. The second model introduces a constraint based approach, wherein the user can set multiple constraint variables (called as `cvar`) that regulate the administrative RBAC operations. The third mode uses an automated approach known as RAdAC, to manage UA and PA operations based on the risk they pose to the user's privacy and security.

OUTLINE: In Sect. 2 we present some of the related works and in Sect. 3 we present the administrative models for RBAC in Android. In Sect. 4 we discuss the rationale for constraints and describe a few examples of operations with respect to Android. Sect. 5 describes the conclusion and the scope of future work for the paper.

2 Related Works

This section describes the prior works that target access control in Android and implement RBAC in Android, in order to improve it. 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. A few works that implement RBAC in Android, without a user controlled administrative model, are described below.

Abdella et.al. [1] implemented RBAC in Android, by assigning roles to permissions and granting these roles to apps. 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. We believe, 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, 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 [11] 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. [15] (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 [12, 13] introduced a decentralized administration for UA, PA and RR assignment relations. Our models obey this distinction, which can also be seen in [9, 14]. 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 [14] 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 [7] 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 paper, are reversed by the appropriate ”revoke” operations.

SARBAC 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 [8].

The models we present in this paper, 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.

3 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_x which stands for administration of RBAC in Android - model x, where x is the num-

Table 1: Entity Sets

APPS
ROLES
PERMS
OWNER
DEV
ANDROID
AE

Table 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 : $\text{cvar}_{\text{protlvl}} \rightarrow 2^{\{\textbf{normal}, \textbf{dangerous}\}}$
pgrant_approved: PERMS $\times$ ROLES $\rightarrow \mathbb{B}$
rgrant_approved: APPS $\times$ ROLES $\rightarrow \mathbb{B}$

Table 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	

ber distinguishing different models. The entity sets and relations from all the three models are denoted in Tables 1, 2 and 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.

3.1 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 1). These entities are populated in accordance to the policies described further in the paper.

– 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 = $\{\text{owner}_1, \text{owner}_2, \ldots, \text{owner}_n\}$

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

– ANDROID, the set containing all the "Android" users. So, ANDROID = $\{\text{android}_1, \text{android}_2, \ldots, \text{android}_n\}$

– 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 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.

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 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, `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, `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, $\forall r \in \text{ROLES}, \forall ae_1 \neq ae_2 \in \text{AE}.\ (r, ae_1) \in \text{ROLE\_OWNER} \rightarrow (r, ae_2) \notin \text{ROLE\_OWNER}$

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 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

Table 4: RiA^0 Administrative Operations

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

– 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.

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

Table 5: Constraints for the PA Relation

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

Table 6: Constraints for the UA Relation

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

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 **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`.
- $C^{PA}_{i}(p,\ r)$, a constraint statement particular to the PA relation, that evaluates to either *true* or *false*. These constraints are defined in Table 5; and, need to be satisfied prior to a permission being assigned to a role.
- $C^{UA}_{i}(p,\ r)$, a constraint statement particular to the user assignment (UA), that evaluates to either *true* or *false*. These constraints are defined in Table 6; and, need to be satisfied prior to a permission being assigned to a role.

Table 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: $r \in \texttt{wished\_roles}(a) \wedge \Big(\Big(\Big(\texttt{protlvl(r)} \in \{\textbf{normal}, \textbf{signature}\} \wedge ae \in \text{ANDROID}\Big) \vee \Big(\texttt{protlvl(r)} = \textbf{dangerous} \wedge ae = \texttt{dev\_of}(a) \wedge \texttt{rgrant\_approved}(a,\ r)\Big)\Big) \vee \Big(\texttt{protlvl(r)} = \textbf{dangerous} \wedge ae \in \text{OWNER}\Big)\Big)$ 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: $\text{UA}' = \text{UA} \setminus \{a,\ r\}$

Helper Functions for ARiA_1: These functions extract data stored on the device, and facilitate 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 assigning 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. Note that, $\forall ae \in \text{AE},\ \forall r \in \text{ROLES},\ \forall p \in \text{PERMS}.\ \texttt{pgrant\_approved}(p,\ r) \rightarrow \Big(\bigwedge_{i=1}^{\texttt{n}} \text{C}_i^{\text{PA}}(p,\ r) \vee \texttt{usr\_approved}(ae,\ p,\ r)\Big)$. If all the constraints are satisfied, then user prompt

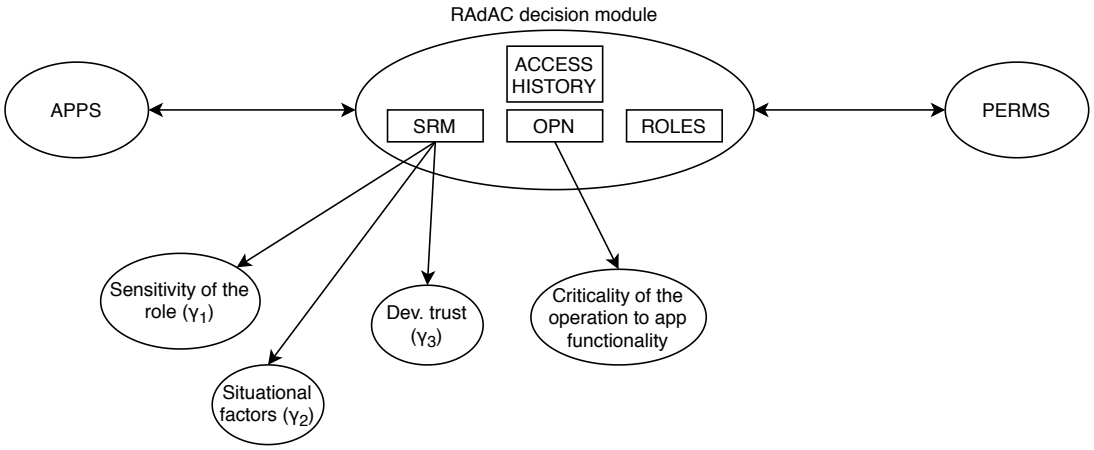


Figure 2: Administration of RBAC in Android using Risk-adaptive approach (RAdAC)

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 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, apps 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 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 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.

3.3 ARiA$_2$ (RAdAC Based Model)

This model is based on the RAdAC model [8], and takes into consideration the operational risk before automatically making an access control decision. 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 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 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.

4 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 8: Operational need for apps 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 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: $\text{UA}' = \text{UA} \setminus \{a,\ r\}$

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

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

4.1 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 apps need; prior research heavily indicates over-privilege in Android [4, 6, 16–18]. 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 apps 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 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

Table 10: Permissions required by WhatsApp in Android

(a) Permissions used frequently

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

(b) Permissions used on occasion

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

(c) Permissions used rarely

AUTHENTICATE_ACCOUNTS
RECEIVE_SMS
SEND_SMS
INSTALL_SHORTCUT

applications.

Furthermore, legitimate applications can get tricked by malicious applications to reveal user data, resulting in the well known privilege escalation attacks [2, 3, 10]. 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 paper, 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.

4.2 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 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 paper is as follows.

AssignApp operation for $ARiA_0$: The third line in the authorization requirements (see Table 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

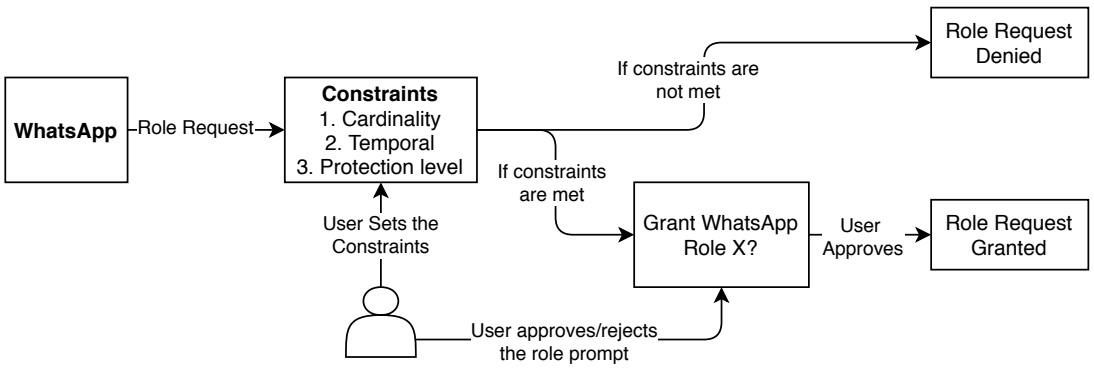


Figure 3: ARiA$_1$: UA Constraints Based Administration of RBAC - **AssignApp** Operation

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. 3 (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 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 paper. 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. 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).

Table 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**

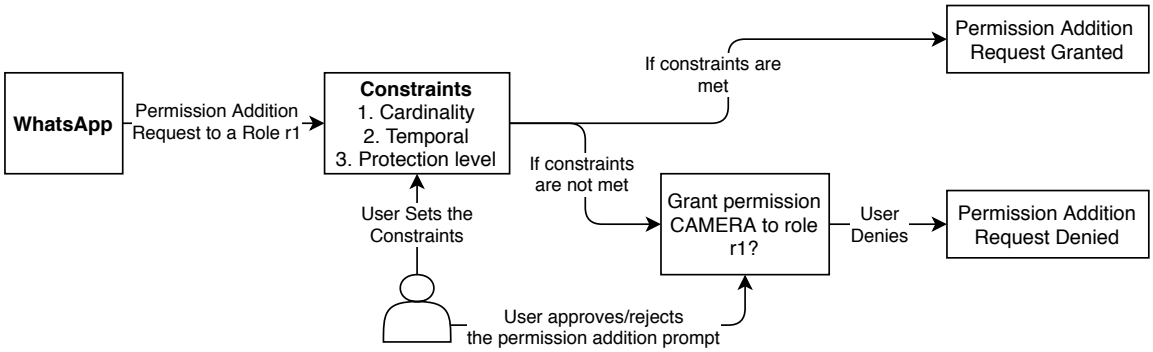


Figure 4: $ARiA_1$ Constraint Based Administration of RBAC - **AssignPerm** Operation

4.3 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 12.

The risk threshold and security risk values, for the **AssignPerm** operation (for $ARiA_2$) are 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.4 that this function states that either all the constraints set by the owner (see Table 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

Table 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**

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).

5 Conclusion

In this work, we have presented several administrative models for RBAC in Android, to facilitate the user management of user-assignment and permission-assignment operations. Without knowing the PA, it is difficult and non-trivial to design an administrative model. We have utilized several techniques such as a constraints on UA and PA operations, and RAdAC based administration of RBAC. The models presented in this paper aim to mitigate the user burden in administration of Android by either automatically processing role-assignment and permission-assignment requests, or by cardinality, temporal and protection level based constraints. A few example operations are also explained in the paper, describing the operation to assign a role to an application, and assigning a permission to a role. Implementation of the models in Android, and, an analysis of the security, privacy and usability with respect to these models to determine the feasibility and advantage offered by them, can be done in the future.

Acknowledgments

This work is partially supported by NSF CREST Grant HRD-1736209 and NSF CAREER Grant CNS-1553696.

References

[1] J. Abdella, M. Özuysal, and E. 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.

[2] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *PRoc. of the 19th Annual Network & Distributed System Security Symposium (NDSS'12), San Diego, California, USA*, volume 17, page 19, February 2012.

[3] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. In *Proc. of the 13th International conference on Information security (ISC'10), Boca Raton, Florida, USA*, volume 6531 of *Lecture Notes in Computer Science*, pages 346–360. Springer, October 2010.

[4] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of the 18th ACM conference on Computer and communications security (CCS'11), Chicago, Illinois, USA*, pages 627–638. ACM, October 2011.

[5] D. F. Ferraiolo, R. Sandhu, S. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed nist standard for role-based access control. *ACM Transactions on Information and System Security*, 4(3):224–274, August 2001.

[6] D. Geneiatakis, I. N. Fovino, I. Kounelis, and P. Stirparo. A permission verification approach for android mobile applications. *Computers & Security*, 49:192–205, March 2015.

[7] N. Li and Z. Mao. Administration in role-based access control. In *Proc. of the second ACM symposium on Information, computer and communications security (ASIACCS'07), Singapore*, pages 127–138. ACM, March 2007.

[8] R. McGraw. Risk-adaptable access control (radac). In *Proc. of the 2009 NIST Privilege (Access) Management Workshop*, volume 25, pages 55–58, 2009.

[9] S. Oh and R. Sandhu. A model for role administration using organization structure. In *Proc. of the 7th ACM symposium on Access control models and technologies (SACMAT'02), Monterey, California, USA*, pages 155–162. ACM, June 2002.

[10] M. Rangwala, P. Zhang, X. Zou, and F. Li. A taxonomy of privilege escalation attacks in android applications. *International Journal of Security and Networks*, 9(1):40–55, February 2014.

[11] F. Rohrer, Y. Zhang, L. Chitkushev, and T. Zlateva. Dr baca: dynamic role based access control for android. In *Proc. of the 29th Annual Computer Security Applications Conference (ACSAC'13), New Orleans, Louisiana, USA*, pages 299–308. ACM, December 2013.

[12] R. Sandhu, V. Bhamidipati, E. Coyne, S. Ganta, and C. Youman. The arbac97 model for role-based administration of roles: preliminary description and outline. In *Proc. of the second ACM workshop on Role-based access control (RBAC'97), Fairfox, Virginia, USA*, pages 41–50. ACM, November 1997.

[13] R. Sandhu, V. Bhamidipati, and Q. Munawer. The arbac97 model for role-based administration of roles. *ACM Transactions on Information and System Security*, 2(1):105–135, February 1999.

[14] R. Sandhu and Q. Munawer. The arbac99 model for administration of roles. In *Proc. of the 15th Annual Computer Security Applications Conference (ACSAC'99), Phoenix, Arizona, USA*, pages 229–238. IEEE, December 1999.

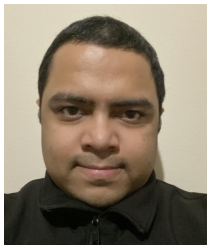
[15] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.

[16] T. Vidas, N. Christin, and L. Cranor. Curbing android permission creep. In *Proc. of the 5th workshop on Web 2.0 Security & Privacy (W2SP'11), Oakland, California, USA*, volume 2, pages 91–96, May 2011.

[17] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the android ecosystem. In *Proc. of the 28th Annual Computer Security Applications Conference (ACSAC'12), Orlando, Florida, USA*, pages 31–40. ACM, December 2012.

[18] S. Wu and J. Liu. Overprivileged permission detection for android applications. In *Proc. of the 2019 IEEE International Conference on Communications (ICC'19), Shanghai, China*, pages 1–6, May 2019.

Author Biography



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

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