SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2016; **9**:5977–5995
Published online 24 January 2017 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1750

RESEARCH ARTICLE

CA-ARBAC: privacy preserving using context-aware role-based access control on Android permission system

J. Abdella, M. Özuysal and E. Tomur*

Department of Computer Engineering, İzmir Institute of Technology, Urla, İzmir, Turkey

ABSTRACT

Existing mobile platforms are based on manual way of granting and revoking permissions to applications. Once the user grants a given permission to an application, the application can use it without limit, unless the user manually revokes the permission. This has become the reason for many privacy problems because of the fact that a permission that is harmless at some occasion may be very dangerous at another condition. One of the promising solutions for this problem is context-aware access control at permission level that allows dynamic granting and denying of permissions based on some predefined context. However, dealing with policy configuration at permission level becomes very complex for the user as the number of policies to configure will become very large. For instance, if there are A applications, P permissions, and C contexts, the user may have to deal with $A \times P \times C$ number of policy configurations. Therefore, we propose a context-aware role-based access control model that can provide dynamic permission granting and revoking while keeping the number of policies as small as possible. Although our model can be used for all mobile platforms, we use Android platform to demonstrate our system. In our model, Android applications are assigned roles where roles contain a set of permissions and contexts are associated with permissions. Permissions are activated and deactivated for the containing role based on the associated contexts. Our approach is unique in that our system associates contexts with permissions as opposed to existing similar works that associate contexts with roles. As a proof of concept, we have developed a prototype application called context-aware Android role-based access control. We have also performed various tests using our application, and the result shows that our model is working as desired. Copyright © 2017 John Wiley & Sons, Ltd.

KEYWORDS
privacy; usability; roles; permissions; context

***Correspondence**
E. Tomur, Department of Computer Engineering, İzmir Institute of Technology, Urla, İzmir, Turkey.
E-mail: emrah.tomur@gmail.com

1. INTRODUCTION

During the last decade, there has been a major change in the computing ecosystem, as more and more computing devices have been replaced by their mobile counterparts. This is due to a combination of advancement of technology that increased the computational capacity of mobile devices and a drop in prices. Mobile devices are now ubiquitous in both personal and enterprise environments. This brings in new challenges in terms of privacy and security because most mobile users have a constantly attached profile to their devices. A large part of this mobile computing environment, approximately 85% [1], is composed of devices running the Android platform. Because of this prevalence of Android devices, around 98% of the attacks target the Android platform [2].

One major issue with the current permission systems is that once a permission is granted, the application always has the privilege to access the related resources. They do not support the dynamic switching of application permissions based on the user context. Such a static approach to application permissions is not well suited in the case of mobile systems. The security risks depend a lot on the present situation and, while accessing a particular resource is safe in some contexts, it might violate privacy in another context. While the modern platforms allow editing of permission lists, it is impractical to manually change a long list of permissions when the context changes fast and often.

Going back to Android versions earlier than Android version 4.3, Android permission system (APS) was not only course-grained but also static. Hence, most of the earlier researches focused on making APS fine-grained and dynamic. Some of the most prominent works published on this topic are AppGuard [3], Apex [4], BlurSense [5,6], Flaskdroid [7], TISSA [8], MockDroid [9], and Dr. Android and Mr. Hide [10]. However, although the concept of context awareness is crucial in mobile environment, it has been ignored by most of these earlier solutions.

Google, on the other hand, modified the permission system to support fine-grained and dynamic permission system starting from Android version 4.3 and officially declared the change in Android 6. In earlier versions of Android, to install an application, the user had to review a long list of permissions and accept some of them that had significantly affected the usability of the system. Android 6 also improves usability by moving the permission decisions to resource access time instead of the time of the application installation similar to that of Apple's iOS permission system.

Once the user grants permission to a specific application, the permission will be added to the list of allowed permissions for that application, and it will permanently stay granted until the user manually revokes the permission. However, if the user denies the requested permission, the decision will not be permanent. The application has the chance to request the permission at a later time.

The negative side of this approach is that permission requests could become annoying if users have to be asked confirmation for each specific permission requested by applications. As a solution to this problem, Google grouped related permissions together. Therefore, when a user is asked to grant permission, he is actually being asked to grant many permissions at a time not just a single permission. For example, when the user grants `PHONE` permission, he is granting six permissions: directly call phone numbers, write call log, read call log, reroute outgoing calls, modify phone state, and make calls without user's intervention permissions. This may result in privacy problems because users are being made to grant all permissions inside the group even though they do not want to grant some of the permissions inside the group. The other important point in APS is that the new system grants Internet permission to all applications by default. Users will not be asked to grant access to the Internet, and it is not even possible to revoke it, even if they wanted to do so.

Despite significant improvements, APS does not take context awareness into consideration yet. Once the user grants permission to an application, the application can use the permission at all conditions without limit. Existing work that introduces context-aware access control (CAAC) to mobile devices include the works of [11], ConUcon [12], CRêPE [13], and ConXsense [14].

In all of these earlier access control models, privacy policies have to be configured for each individual entity separately. For example, in APS, we may have to deal with approximately 140 permissions. The problem with such a model is that the user has to deal with large number of policy configurations. Generally, if we have A applications and P permissions, in the worst case, we need to deal with $A \times P$ number of policies. In addition, in the case where context is considered, context policy configuration has to be performed for each permission per each application. Users usually need to associate more than one context with a single permission. If we have C number of contexts for each permission on average, we will end up with $A \times P \times C$ number of policies. For systems that have large number of permissions and installed applications, configuring this much number of policies leads to reduced usability especially for the ordinary users.

In fact, different studies show that most mobile users are not interested or not able to configure detailed policies, and instead, they prefer to accept every permission request without careful examination resulting in overprivileged applications. For example, [15] reports how much Android users understand APS and pay attention to privacy risks during application installation as discovered by an Internet survey and laboratory experiments. The result shows that only 3% of users correctly understand the permissions and only 17% of the users give their attention to permissions requested by applications.

To overcome these problems, we propose a permission system that combines role-based access control (RBAC) with CAAC. Our model, context-aware Android role-based access control (CA-ARBAC), works by assigning roles to applications where roles consist of a list of permissions that will be activated and deactivated for the containing role depending on a set of contexts. In CA-ARBAC, users are not required to deal with a large number of permissions; instead, they just need to assign roles to applications. In other words, the kind of permission grouping adopted by Android to promote usability is replaced by RBAC in our system without compromising the privacy. However, there is small amount of overhead at the beginning. The user has to configure policies initially. Once created, roles can be used for as many applications as needed. Furthermore, it is possible to have default roles such that ordinary users who have difficulty in creating their own roles can use them.

Therefore, by using our system, the number of policies can be reduced to $A \times R$ where R is the number of roles. Moreover, in our model, because role-permission and permission-context maps are independent of applications, we do not need to modify these configurations if applications have to be uninstalled and installed again. Without RBAC, every time we re-install a given application, all the rules related to that application have to be reconfigured because they are dependent on the application. Altogether, our system satisfies three requirements at the same time: least privilege, dynamic permission granting and revoking, and keeping the number of policies as small as possible.

To create roles, we followed a method of categorizing applications into logical groups. Examples of functional groups include messenger applications, photography applications, multimedia applications, and travel applications. Roles correspond to these functional groups. Different functional groups require different type and number of permissions and hence will be assigned different kinds of roles. This approach should not be taken as the best way of creating roles. It is rather a simple approach used to demonstrate our model. We believe that there can be better way of doing this. However, as the main goal of this paper is not providing an appropriate method of creating roles, we have chosen to postpone this work to the future.

To give more insight into the process of role creation and assignment in our system, we look at some examples. For instance, "PHOTOGRAPHY" role can be created for photography applications and a "MESSENGER" role can be created for messenger applications. These roles will be allocated different number and types of permissions. For example, permissions such as `CALL_PHONE`, `SEND_SMS`, `RECIEVE_SMS`, `RECORD_AUDIO`, `CAMERA`, `WRITE_CONTACTS`, or `READ_CONTACTS` are normal for messenger applications but most of them are suspicious if requested by a photography application. Thus, out of these mentioned permissions, PHOTOGRAPHY role may be assigned only `CAMERA` permission, whereas MESSENGER role could be granted all of them.

Moreover, using the context awareness functionality of the system, we can impose restrictions on permission usage for already granted permissions. For instance, we can set the precondition that `RECORD_AUDIO` permission is not allowed for MESSENGER role during the time that the user is talking on the phone or if the user is in a meeting room. We can also say that PHOTOGRAPHY role is forbidden from using `CAMERA` permission if the user is at home or in a meeting room. A common attack by hackers is calling and/or sending SMS messages to premium numbers when the phone is locked. In such cases, we may have a context policy that forbids the application from using permissions such as `CALL_PHONE`, `SEND_SMS`, and `RECIEVE_SMS` if the screen is locked. Another common privacy attack is tracking user's location. The user can have a policy that denies location permission to applications when the user is at secure locations such as home. As mentioned earlier, in Android 6, all applications have Internet permission by default. We believe that limiting the Internet permission depending on context allows a better privacy/functionality trade-off that is possible with the system we propose.

We also argue that CA-ARBAC can improve privacy and usability at the same time. First of all, RBAC is a known method of applying principle of least privilege. RBAC enables users to give a minimum number of permissions as they wish. Secondly, CAAC further strengthens privacy protection by allowing dynamic alteration of application privileges based on contexts. As explained earlier, the numbers of policy rules needed to be configured is less in CA-ARBAC as compared with the existing access control models.

Our contributions:

- New context-aware role-based access control (CA-RBAC) model for APS that assigns roles to applications and associates contexts with permissions allowing principle of least privilege and dynamic granting and revoking of application permissions with little effect on usability
- Dynamic and fine-grained permission system for Android versions earlier than Android 6.
- New CA-RBAC architecture for APS that can possibly be integrated to Android security modules (ASM) [16].

The rest of this paper is organized as follows: Section 2 discusses related work. We review existing Android security mechanisms in Section 3. CA-ARBAC access control policy model is explained in detail in Section 4. Section 5 presents our CA-ARBAC architecture designed based on ASM. Section 6 provides a description of the implementation of our system followed by Section 7 that demonstrates our implementation by examples and tests. The formal verification of our system is presented in Section 8. Section 9 addresses performance related issues, and Section 10 discusses our future work plan. Finally, Section 11 summarizes and concludes the paper.

2. RELATED WORK

In this section, we examine existing work that is closely related to ours by dividing them into three groups: those that focus on RBAC, those that deal with CAAC, and those that combine both of these (CA-RBAC models).

Role-based access control models. MPDROID [17] is good example of pure RBAC model that is close to our approach. It is a security framework that supports two kinds of access control models at two layers of Android system: RBAC at the application framework layer and mandatory access control (MAC) at the kernel layer. At the application framework layer, it enhances APS with RBAC to provide fine-grained access control. This enables users to define their own security policy and control malicious applications. At the kernel layer, it implements MAC to allow administrators enforce fine-grained access control. Administrators can limit activities of applications and their processes according to a centralized security policy. Similar to our system, users authorize Android applications by assigning roles instead of permissions. But MPDROID does not take context into account.

Context-aware access control models. Various kinds of CAAC models have been proposed in the past. Four of the most recent ones are [11], ConUcon [12], CRêPE [13], and ConXsense [14]. Our CAAC policy model is analogous to that of [11]. Similar to our proposed system, it associates Android permissions with context. However, it works only for two kinds of contexts: location and time. Our system is designed to support more types of context. In addition, [11] is a pure CAAC model unlike CA-ARBAC that is a hybrid of RBAC and CAAC.

ConUcon proposed a general CAAC framework that works with several mobile platforms. It uses context information to protect privacy and to control resource usage. It supports active context usage control, that is, context check is not only performed prior to resource access but also during the access.

CRêPE developed a system that enforces fine-grained context-related policies on Android. In CRêPE, policies can be configured both by phone users and authorized third parties locally or remotely, via SMS, MMS, Bluetooth, or QR-code. The policies can also be applied in a system-wide manner.

ConXsense focuses on usability of APS that is ignored by most other CAAC models. It does not require users to configure policies. Instead, it is based on a probabilistic approach that automatically classifies contexts according to their security and privacy risks using machine learning and context sensing.

Context-aware role-based access control models. There exist also works that combine aspects of RBAC model with context awareness. Some of the existing literature that fall under this category are dynamic role-based access control for Android [18], context-related role-based access control [19], CA-RBAC [20], [21], and role-based access control for Android (RBACA) [22].

Dynamic role-based access control for Android offers an RBAC system that is similar to traditional desktop computers. It allows the management of multiple users on Android mobile devices. In addition, it provides both application and permission level fine-grained access control by using RBAC.

Context-related role-based access control incorporates the contextual information of user and system environment with the traditional RBAC. Context-related role-based access control defines user roles based on their access privileges. Users are categorized according to their access rights of device's resources and services. Each user possesses one role at a time.

Context-aware role-based access control also proposes an access control model that combines RBAC with context awareness. CA-RBAC dynamically assigns roles to users and permissions to roles according to the current context. CA-RBAC is similar to our system in that it assigns permissions dynamically. However, CA-RBAC is for ubiquitous computing environments not for Android. In addition, our system does not change roles contextually to avoid unnecessary creation of roles.

Jung and Park [21] is a relationship-based CA-RBAC approach for mobile users in enterprise environment. It considers the relationship between users as context information. The access control architecture is designed by using near-field communication technology.

Role-based access control for Android is the closest existing work to our approach. It is an RBAC approach proposed for APS to mitigate the security risks caused by overprivileged applications. In RBACA, similar to our system, roles are assigned to applications, and roles contain a subset of Android permissions. The main difference between our system and RBACA is the way context is handled. In RBACA system, context is associated with roles. Application's roles are switched manually or dynamically depending on some contexts. In our system, roles stay constant. Context is applied on permissions, that is, permissions are turned on and off for the role they belong based on the status of associated context. We argue that our approach avoids the creation of extra roles for each context. Moreover, our method allows fine-grained context usage at permission level.

To support the claim that our system avoids the creation of unnecessary number of roles as compared with both CA-RBAC and RBACA, we present a comparison of our system with RBACA and CA-RBAC by example. Let us assume that a given user has installed application A that requires five permissions P_1, P_2, P_3, P_4, and P_5. Moreover, let us assume that the user wants to associate three contexts C_1, C_2, and C_3 with permissions P_1, P_3, and P_5, respectively, that is, P_1, P_3, and P_5 are allowed for application A only when contexts C_1, C_2, and C_3 are satisfied consequently. However, P_2 and P_4 are always allowed for application A as there is no context associated with them. To satisfy the previous requirement, in our model, only one role needs to be created for application A as shown in Table I. However, in RBACA and CA-RBAC, three roles need to be created for application A as shown in Table II. Application A will be assigned either role R_1, R_2, or R_3 based on the contexts C_1, C_2, and C_3.

Table I. CA-ARBAC way of creating role for application A.

Role	Permissions	Condition to use permission
	P_1	When C_1 is satisfied
	P_2	Always
R_1	P_3	When C_2 is satisfied
	P_4	Always
	P_5	When C_3 is satisfied

CA-ARBAC, context-aware Android role-based access control.

Table II. RBACA and CA-RBAC way of creating role for application A.

Context	Role	Permissions
		P_1
C_1	R_1	P_2
		P_4
		P_2
C_2	R_2	P_3
		P_4
		P_2
C_3	R_3	P_4
		P_5

RBACA, role-based access control for Android; CA-RBAC, context-aware role-based access control.

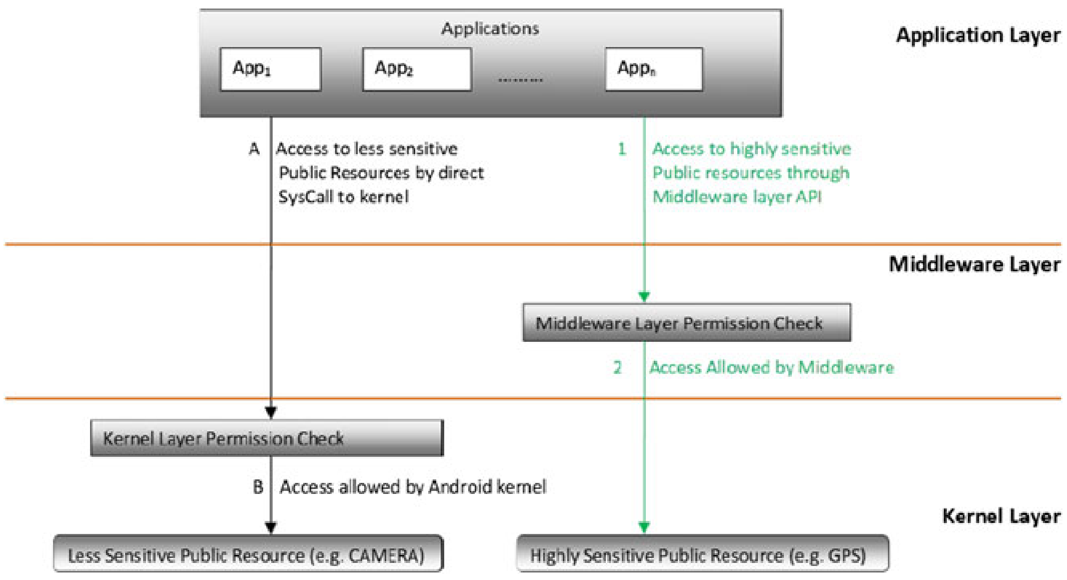

Applications
App₁
App₂
App_n
Application Layer
A Access to less sensitive Public Resources by direct SysCall to kernel
1 Access to highly sensitive Public resources through Middleware layer API
Middleware Layer
Middleware Layer Permission Check
2 Access Allowed by Middleware
Kernel Layer Permission Check
B Access allowed by Android kernel
Kernel Layer
Less Sensitive Public Resource (e.g. CAMERA)
Highly Sensitive Public Resource (e.g. GPS)

Figure 1. Android application resource access procedure.

3. ANDROID SECURITY OVERVIEW

In Android, every time the user installs an application, a unique user ID (UID) is generated for the application. The application runs under that UID for the whole of its life. In addition, all data stored by that application are assigned that same UID, whether a file, database, or other resource. Every application's sandbox accesses its own private resources by direct system calls to the kernel. The Linux kernel enforces private resource access by comparing the UID of the requesting application with the UID of the requested resources. An application needs permission from the user to access resources other than its private resources.

An application can access resources other than its private directory using two different ways. Firstly, when an application is granted less sensitive public resources such as SDCard and BLUETOOTH permissions, it is added to a Linux group that has access to the corresponding resources. Thus, the application is assigned a group ID (GID) in addition to the UID. Such kinds of public resources are also accessed by directly interacting with the underlying kernel through system calls in a similar fashion to private resource access. The Linux kernel enforces the access control policy, that is, the access control in the file system ensures that the application has the necessary permissions. For example, it checks whether the application is allowed to open a file on the BLUETOOTH by checking the GID of the application with the GID that is privileged to access the BLUETOOTH. The file system access control uses traditional Linux discretionary access control. The Linux kernel access control also supports a MAC scheme called security-enhanced Android starting from Android 4.3.

Secondly, applications are not allowed to access highly privileged resources, such as SMS, PHONE, and CONTACTS, by direct system calls to the kernel. Such kinds of resources are accessed through Middleware layer system services and applications that implement the target application program interface (API). For example, the location service provides the API used to communicate with the GPS or other location providers. Therefore, if an application wants to obtain user's location, it communicates with the location service instead of directly interacting with the GPS or other location providers. A permission check is also performed by system services/applications at the middleware layer. The system services/applications use Android permission validation mechanism to check whether the caller application with the given UID has the necessary permission or not. The system service/application obtains the UID of the caller application from the binder IPC. Figure 1 elaborates Android application resource access procedure.

The fact that in Android, applications are uniquely identified by their UIDs makes assigning roles to applications possible. In our CA-ARBAC system, applications are also identified by their UIDs. Our system assigns roles to applications based on their UIDs. Therefore, permission check is also performed by UID during resource access.

4. CONTEXT-AWARE ANDROID ROLE-BASED ACCESS CONTROL ACCESS CONTROL MODEL

Context-aware Android role-based access control is an adaptation of the traditional RBAC model. The three main components of the traditional RBAC are users, roles, and permissions. In CA-ARBAC, applications replace users, and applications are considered as users. In addition to this, our model contains a fourth component: context. In the following sections, we describe CA-ARBAC access control policy model. Here is the formal definition of the basic system components:

Definition (Applications)**.** *An application is any Android application in the system. Let* $\mathtt{A}$ *represent the set of all Android applications installed on the device.*

Definition (Roles)**.** *In CA-ARBAC, a role is a functional category of Android applications. Let* $\mathtt{R}$ *stand for the set of roles created in the system.*

Definition (Permissions)**.** *The permissions in our system are any one of the permissions defined in Android system. Let* $\mathtt{P}$ *represent the set of all permissions in Android system.*

4.1. Application assignment

Application assignment is a mapping that associates an application with an assigned role. A role can be assigned to multiple applications at the same time. Similarly, an application can also have more than one role simultaneously.

Definition (Application role mapping)**.** *Let* $\mathtt{ARM}$ *be the list containing the mapping between applications and roles. The elements of* $\mathtt{ARM}$ *are tuples:* $\langle A_i, R_j \rangle$ *where* $A_i \in \mathtt{A}$ *and* $R_j \in \mathtt{R}$.

A many-to-many mapping (application-to-role assignment relation) exists between applications and roles: $\mathtt{ARM} \subseteq \mathtt{A} \times \mathtt{R}$. The user manually creates roles and assigns it to one or more applications. When the user assigns roles to applications, it is added to the application role mapping ($\mathtt{ARM}$). The ARM is static and do not change dynamically based on contextual data.

4.2. Static permission assignment

Permission assignment is a mapping that associates roles with an assigned permission. A role can be assigned multiple permissions and a single permission can also occur in many roles.

Definition (Role permission mapping)**.** *Let* $\mathtt{RPM}$ *be the list containing the mapping between roles and permissions. The elements of* $\mathtt{RPM}$ *are tuples:* $\langle R_m, P_k \rangle$ *where* $R_m \in \mathtt{R}$ *and* $P_k \in \mathtt{P}$. *A many-to-many mapping (role-to-permission assignment relation) exists between roles and permissions,* $\mathtt{RPM} \subseteq \mathtt{P} \times \mathtt{R}$.

In the absence of context associated with permissions, the permission set assigned to roles stays active for the role all the time, that is, all the permissions assigned to a role are allowed for the role all the time.

4.3. Dynamic permission assignment in the presence of context

In CA-ARBAC, the usage of permissions inside a given role can be restricted by specifying the conditions under which the permission should or should not be allowed.

In this paper, we use two kinds of context sources: environmental context and system context. Location of the user and surrounding temperature are types of environmental contexts. Some examples of system context include: time, battery status, whether there is an ongoing phone call or not, and whether the screen is locked or not.

Definition (Context)**.** *Many kinds of contexts can be applied in our model. Each context is identified by its name and one or more attributes:* $\mathtt{Context} = \langle \mathit{ContextName}, \mathit{ContextAttributes} \rangle$. *For example, LOCATION is a context that is identified by two attributes: latitude and longitude; TIME is a context identified by single attributes: time of day.*

Definition (Context policy)**.** *A context policy is a rule that specifies the condition under which a given permission should be allowed or not allowed. It consists of two parts: the context description and the action to take. Context description is expressed as follows:* $\mathtt{ContextDescription} = \langle \mathit{ContextName}, \mathit{Operator}, \mathit{AttributeValues} \rangle$. *The Operator represents different kinds of key words used for comparison. It includes: EqualTo, GreaterThan, LessThan, GreaterThanOREqualTo, LessThanOREqualTo, InBetween, and In. AttributeValues is the set of values for each of the context attribute of the given context. Let* $\mathtt{CD}$ *and* $\mathtt{CP}$ *be the set of all context descriptions and context policies configured in the system, then* $CP_i = \langle CD_i,\ \mathit{Action} \rangle$ *where* $CP_i \in \mathtt{CP}$ *and* $CD_i \in \mathtt{CD}$.

When we configure context policy, we may need to specify multiple values for the context based on the range we want to include. For example, we may specify that some permission should be denied access from some starting time to some end time. Another case is we may specify that a given permission can be allowed on weekdays. The action attribute indicates the action to be taken when the context is satisfied. It is either allow or deny.

Definition (Context combination)**.** *Often, context policies are made up of a combination of contexts not just with a single context. Let* $\mathtt{CCP}$ *(combined context policy) be the set of context policies containing combination of context descriptions, then* $CCP_i = \langle (CD_1 \wedge CD_2 \wedge \ldots \wedge CD_m),\ \mathit{Action} \rangle$, *where* $CCP_i \in \mathtt{CCP}$ *and* $CD_i \in \mathtt{CD}$.

The value of $\mathtt{CCP}$ is either true or false based on the value of the *Action* attribute and the return value of the combination of the context descriptions. When the *Action* attribute is set to allow, it returns true if all the contexts in the combination are satisfied. Otherwise, it returns false. When the *Action* attribute is set to deny, it returns false if all the contexts in the combination are satisfied. Otherwise, it returns true.

Definition (Context list)**.** *Sometimes, permissions are also associated with a list of combined contexts joined*

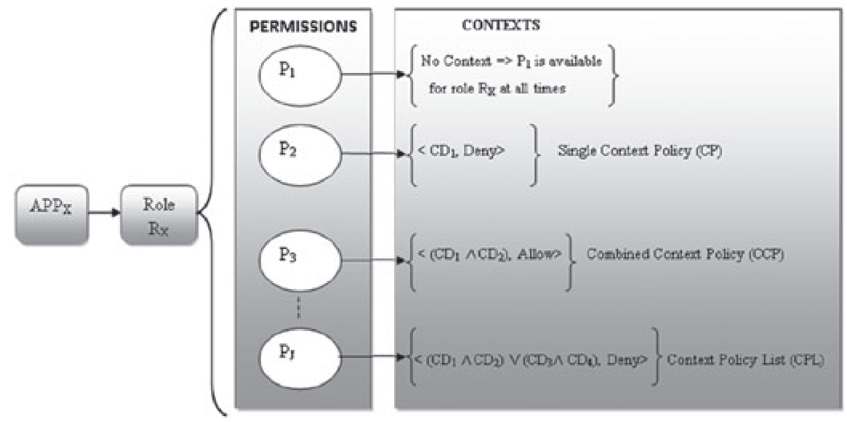

PERMISSIONS
CONTEXTS
APPx
Role Rx
P1
P2
P3
Pj
No Context => P1 is available for role Rx at all times
< CD1, Deny>
Single Context Policy (CP)
< (CD1 ∧CD2), Allow>
Combined Context Policy (CCP)
< (CD1 ∧CD2) V (CD3∧ CD4), Deny>
Context Policy List (CPL)

Figure 2. Illustration of context-aware Android role-based access control model.

together using logical disjunction operator. Let `CPL` *be the set of context policy lists,* $CPL_i \in CPL$*, where* CPL_i *is the context policy list associated with permission* P_i*.* $CPL_i = \langle (CCP_1 \ V \ CCP_2 \ V \dots \ V \ CCPn), Action >$*, where* $CCP_i \in CCP$*.*

In this case, when the *Action* is set to allow, `CPL` returns false if all of the `CCP`s in the list return false. Otherwise, it returns true. When the *Action* is set to deny, `CPL` returns true if all the `CCP`s in the list return false. Otherwise, it returns false.

Definition (Active permissions)**.** *Not all the permissions assigned to a role are active for the role all the time. An application can only use active permissions. Whether permission is active or not for a given role is determined by the list of contexts associated with the permission. Active permissions are permissions for which the associated context is satisfied.*

Figure 2 above illustrates our access control model graphically by example. In the figure, $\mathtt{APP}_x$ is assigned the role R_x. Role R_x is granted *j* permissions. Permission P_1 has no any context associated with it, which means that it will be active for role R_x at all times. Permissions P_2, P_3, and P_j of role R_x on the other hand have contexts associated with them. P_2 is associated with single context. P_3 is associated with combination of two contexts, and P_j is associated with a context list that contains two combined contexts. APP_x can access these permissions only if the contexts associated with them are satisfied.

5. CONTEXT-AWARE ANDROID ROLE-BASED ACCESS CONTROL ARCHITECTURE

Android system does not provide a comprehensive API for the development and integration of new security applications and enhancements. Because of this, all of the earlier Android security system enhancements required modification to the Android operating system. Consequently, these previous works are provided in one of two ways. Some of them are presented as separate model-specific patches to the Android operating system. Others are imbedded into Android's software stack and become part of the Android system.

As noted by Android security framework (ASF) [23], if we look back at the history of stable security frameworks like Linux security modules (LSM) and the BSD MAC Framework, following either of the earlier two approaches is not a practical way of providing security solutions.

Firstly, if security solutions are provided as updates to Android operating system, it causes maintenance problems. Every time the operating system is upgraded to another version, each of the security solutions should also be updated to make them compatible with the new version.

Secondly, integrating the security model into the operating system makes it much more difficult to work with multiple different security models that are suitable for different kinds of scenarios. Moreover, each patch to the security system requires rebuilding the operating system, which is very difficult for the average user.

Understanding this gap, ASF and ASM recently developed an extensible security framework for Android that provides a programmable interface that allows the development and integration of various kinds of security applications in the form of security modules. ASF and ASM are two independent but similar systems developed by independent researcher groups. Taking the aforementioned unsatisfactory situations into consideration, we decided to design CA-ARBAC as independent code-based security module which is a pure java application built on the application layer based on ASM. Before we present our design, we briefly discuss ASM.

5.1. Android security modules

The motivation behind the development of ASM is to provide a programmable interface that will enable security application developers to extend Android security without changing the operating system. ASM provides a reference monitor interface for building new security applications/security modules. This allows reference monitor developers to focus on their novel security models and not worry about enforcement hooks. The security modules are called ASM apps, and they are developed just like any other conventional Android applications. ASM apps implement the security logic. They use ASM hooks for policy enforcement.

Android security modules supports enforcement hooks at two layers of Android system: at the middleware layer and at the kernel layer. The ASM reference monitor interface that is placed at the middleware layer is called ASM Bridge. This is the part which is directly communicating with ASM apps. The ASM Bridge automatically invokes a callback in the ASM apps when a resource is going to be accessed. The component of ASM that provides enforcement hooks at the Linux kernel is ASM LSM. ASM LSM makes up calls to the ASM Bridge when an application makes a direct system call to kernel to access sensitive resource.

5.2. Architecture overview

The design of CA-ARBAC, which is based on ASM, is shown in Figure 3. The big grey colored part is our CA-ARBAC system. CA-ARBAC is an ASM app. Therefore,

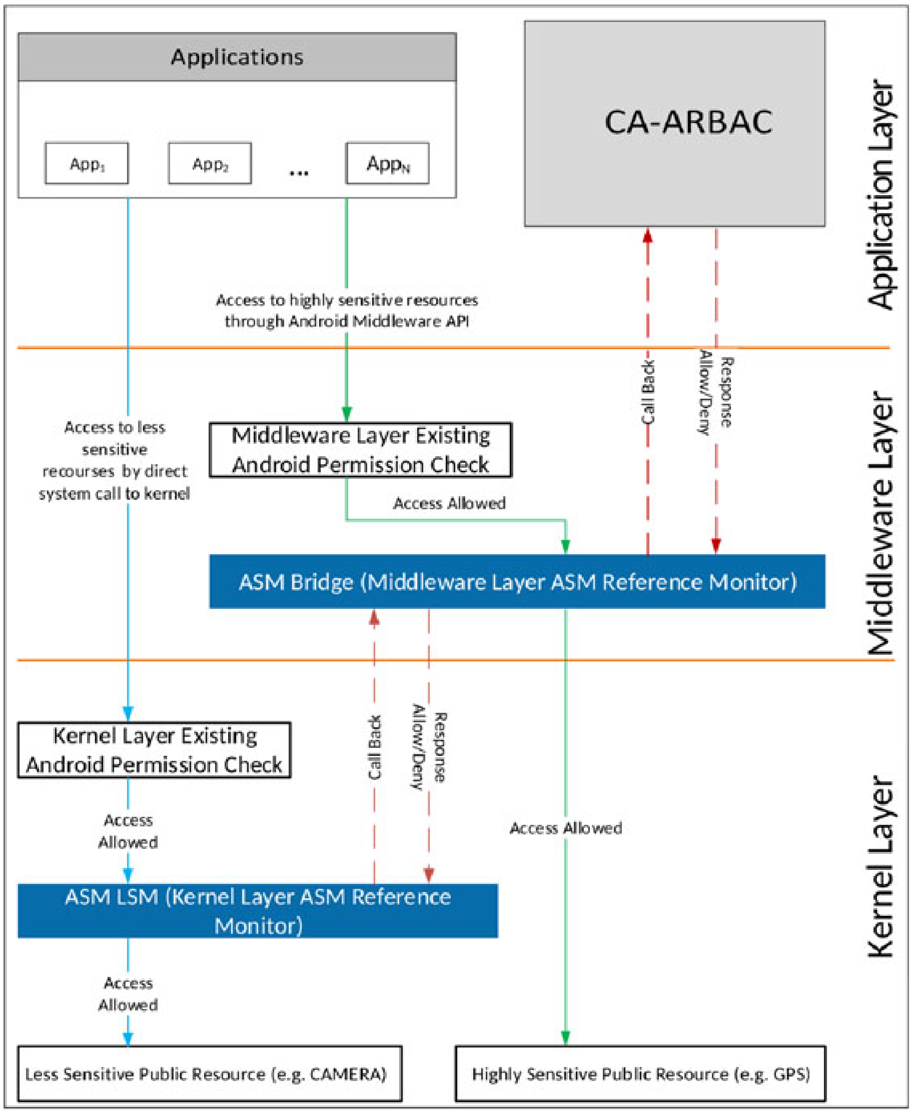

Applications
App1
App2
...
AppN
CA-ARBAC
Application Layer
Access to highly sensitive resources through Android Middleware API
Call Back
Response Allow/Deny
Middleware Layer
Access to less sensitive recourses by direct system call to kernel
Middleware Layer Existing Android Permission Check
Access Allowed
ASM Bridge (Middleware Layer ASM Reference Monitor)
Kernel Layer Existing Android Permission Check
Call Back
Response Allow/Deny
Access Allowed
Access Allowed
ASM LSM (Kernel Layer ASM Reference Monitor)
Access Allowed
Kernel Layer
Less Sensitive Public Resource (e.g. CAMERA)
Highly Sensitive Public Resource (e.g. GPS)

Figure 3. CA-ARBAC architecture based on ASM. On the figure, there are three kinds of components. The white colored parts belong to existing Android system components which participate in the resource access process. The two blue colored boxes represent ASM system extensions to Android operating system. The two (blue) and (green) arrows indicate the steps followed when an application needs to access a sensitive resource in Android system that is enhanced with ASM and CA-ARBAC. The big grey colored part on the top right corner is our CA-ARBAC system.

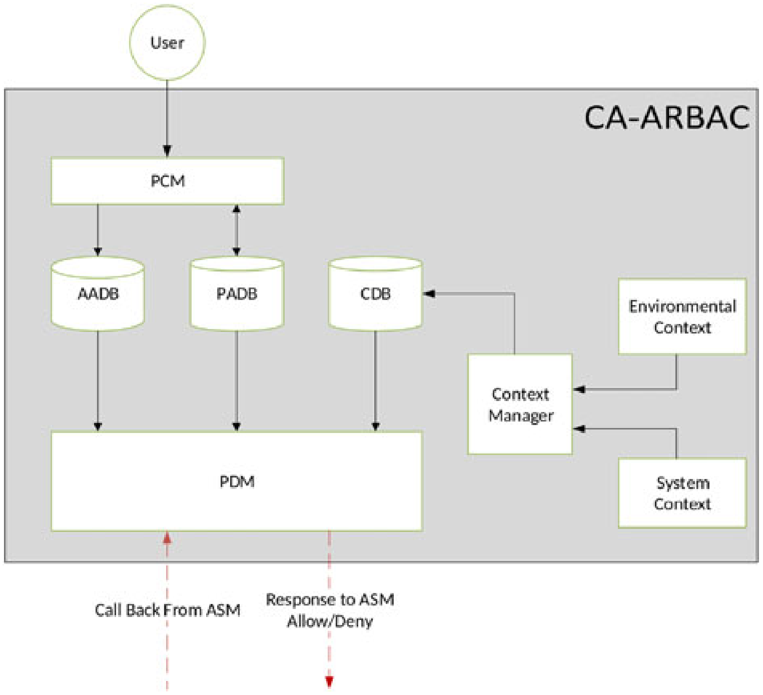


Figure 4. Context-aware Android role-based access control (CA-ARBAC) components.

in our case, CA-ARBAC is only responsible for implementing the policy logic. It is concerned about policy decision making, policy configuration, context detection, and policy storage. Policy enforcements are handled by ASM. CA-ARBAC receives a callback from ASM when a sensitive resource is going to be accessed. CA-ARBAC consists of four components. The components of CA-ARBAC, which are shown in Figure 4, are explained in the subsequent sections as follows:

Policy decision manager (PDM). The PDM is the core component of CA-ARBAC. It is the part that makes security decisions inside CA-ARBAC. CA-ARBAC is connected to ASM through PDM.

Policy configuration manager (PCM). Policies are configured by the user through the user interface component called PCM.

Context manager. The PDM needs to obtain the current contextual information to make access decision, that is, it needs to check whether the pre-specified context associated with permissions is fulfilled or not. The PDM obtains current context information from the context database (CDB). The CDB stores different kinds of context and their current values. The context manager is the part responsible for continuously receiving updates of contextual information from different context provider elements and updating the CDB with the new values.

Context-aware Android role-based access control policy databases. Access control policies are stored in two separate databases. Application assignment database (AADB) stores applications and their corresponding roles. Permission assignment database (PADB) is used to store roles and the permissions assigned to roles. PADB also contains context information for permissions that have associated contexts.

5.3. Working principle of context-aware Android role-based access control

Every time an application wants to access sensitive resource other than its private resource, it makes a call to either the middleware layer Android API or directly to the kernel as shown by blue and green lines on Figure 3. Permissions are also enforced at both of these points. In the existing Android system, when the two Permission Enforcement Points receive access request message, they will decide whether the application should be allowed access or not, and they either allow access to resources or send back exception message.

In Android system enhanced with ASM, the ASM intercepts access request messages and sends callback to registered ASM applications. CA-ARBAC is an ASM application. Thus, it receives callback from ASM through the PDM interface. The callback comes both from the kernel and the middleware layers. The red lines on Figure 3 show callback and response messages. The callback message contains tuples Application ID, Permission, that is, the application that requests access and the requested permission. The PDM analyzes the request and decides on whether the request should be allowed or denied. The PDM then responds with allow/deny message to ASM based on the decision made.

The PDM performs the following actions to make a decision. It first checks if there is a role assigned to the requesting application in the AADB. If there is no role given for the application in AADB, the PDM automatically sends deny message to ASM. If it finds a role associated with that application, it retrieves the list of permissions allowed for that role from PADB. If the requested permission is not found in the list, PDM will again send back a deny message to ASM. If the requested permission is found in the list, there are two cases. Either there is context data associated with the permission or not.

Therefore, the PDM goes on to check if there are any contexts associated with the permission. If the permission is not accompanied by context, it will be allowed for the application automatically. If however, there are some contexts associated with the permission, the PDM obtains the current value of the context from CDB and checks if the preconfigured context is satisfied by the current value of the context. If all the contexts are satisfied, an allow message will be sent to ASM. Otherwise, a deny message will be sent to ASM. Based on the response from PDM, the ASM either allows the application to access the requested resource or sends back an access denied exception to the application.

5.4. Integrating context-aware Android role-based access control to different kinds of Android devices and versions

Android operating system runs on a variety of different devices from different vendors. Because of this, device manufacturers customize the operating system for different reasons such as to be able to adapt it to their specific hardware design, to add new services, and to make it fit to different models such as smartphone and tablet. In addition to device manufacturers, Google also updates the operating system frequently. Given the vast variety of Android devices and versions, providing security applications that require modification as devices and Android versions change is very ineffective.

Context-aware Android role-based access control system is an independent code-based security application that does not require modification based on the type of the device or version of Android operating system installed on the device. This is because of the following two reasons: First of all, CA-ARBAC is not affected by the modification performed by device manufacturers or version upgrades as it is situated on the application layer of Android software stack. Secondly, CA-ARBAC system security policy configuration is based on the user-based security model of the Linux kernel. CA-ARBAC identifies each application using unique UID and keeps permission grant information with the UID. The user-based security model of the Linux kernel is the basic building block for Android security mechanism that does not change from device to device or from version to version.

Nonetheless, CA-ARBAC requires a programmable interface that sends callback (notification message) when a sensitive resource is going to be accessed. That is where systems such as ASM and ASF come to the scene. Therefore, to use CA-ARBAC on different Android devices and versions, we need to have systems such as ASM integrated to the operating system. One may argue that we should have developed our own reference monitor similar to ASM and ASF. For the two reasons mentioned earlier, this is not a good way. Every security applications developer should not modify the operating system but rather should rely on some extensible security framework integrated to Android code base and provide a programmable interface for security application developers. We believe that Google will soon integrate such kind of security framework into Android operating system.

5.5. Context-aware Android role-based access control on other mobile platforms

In this paper, we show the implementation of CA-ARBAC system on Android. However, CA-ARBAC can also be implemented on other major mobile platforms such as iOS and Blackberry.

iOS mobile platform. In Android, applications are uniquely represented using Linux UID. In CA-ARBAC system, these UIDs are mapped to roles and roles are mapped to Android permissions. Analogous to the UID in Android, iOS system also uses a unique ID called bundle ID to identify iOS applications. Moreover, access to resources in iOS is allowed through something similar to Android permission called entitlement. Entitlements are capabilities that are declared in the application code to request access to different system resources. Hence, CA-ARBAC can be implemented in iOS using bundle ID and entitlements. Bundle IDs can be mapped to roles and role can be mapped to entitlements. A reference monitor that sends a callback message should also be inserted into permissions enforcement points inside the iOS operating system.

Blackberry mobile platform. Blackberry's application security mechanism is quite similar to that of Android. Each application runs in its own virtual container called sandbox. Application sandboxes are isolated from each other. The sandbox encompasses the applications: own files and the application's memory area. The operating system assigns a unique group ID to each application. Like that of Android, by default, an application can access its own data in its own sandbox. If the application wants to access resources that are not associated with the application's group ID, it has to obtain permission from the user. The part of the operating system that enforces permissions is called authorization manager. Hence, CA-ARBAC could be implemented on Blackberry mobile platform by using group IDs and Blackberry application permissions.

Moreover, authorization manager should be modified to send callback message to CA-ARBAC whenever a sensitive resource is going to be accessed.

6. IMPLEMENTATION

Currently, ASM is not integrated to the Android operating system. Therefore, we could not use ASM directly, and instead, we simulated ASM with our own application called ASM simulator. Figure 5 shows the architecture of CA-ARBAC after ASM is replaced with the ASM simulator. In the future, we plan to rebuild Android operating system with ASM and integrate our system to it. For the time being, we will explain the implementation of our system using the ASM simulator application.

6.1. Android security modules simulator

Android security modules simulator is an Android library application that controls resource accesses operations of other applications. ASM simulator is different from ASM in that it lies in the application layer as opposed to ASM that is placed at two of the other layers: the middleware and kernel layers. In normal case in Android, applications directly communicate with either the middleware layer or the kernel layer Permission Enforcement Points to access sensitive resources. In this case, applications obtain access to sensitive resources through ASM simulator. When an application wants to access some resource, it calls the public method `getAsmSimulatorService()` of ASM simulator by specifying the resource it needs to access.

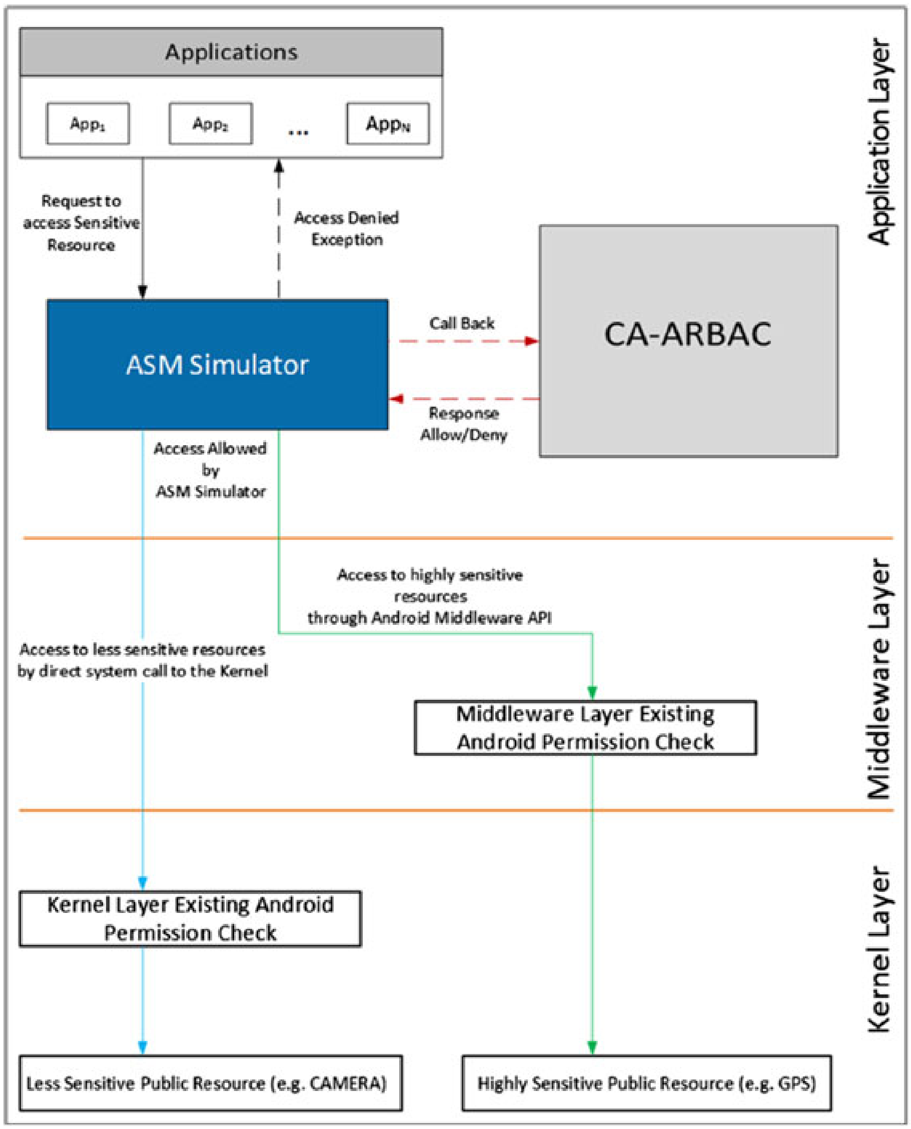

Applications
App₁
App₂
...
Appɴ
Application Layer
Request to access Sensitive Resource
Access Denied Exception
ASM Simulator
Call Back
Response Allow/Deny
CA-ARBAC
Access Allowed by ASM Simulator
Access to highly sensitive resources through Android Middleware API
Access to less sensitive resources by direct system call to the Kernel
Middleware Layer Existing Android Permission Check
Middleware Layer
Kernel Layer Existing Android Permission Check
Kernel Layer
Less Sensitive Public Resource (e.g. CAMERA)
Highly Sensitive Public Resource (e.g. GPS)

Figure 5. Architecture of context-aware Android role-based access control (CA-ARBAC) with Android security modules (ASM) simulator.

The ASM simulator checks whether the application has the necessary permission to access the requested resource by contacting CA-ARBAC. ASM simulator calls the public method `call()` inside the PDM. The `call()` method in turn invokes the private method `checkAppPermission()` inside PDM itself. The arguments to both methods are tuples Application ID, Permission. The response to ASM simulator is either `ALLOW` or `DENY`. If the response is `ALLOW`, ASM simulator obtains the resource from Android system on behalf of the application and passes the acquired resource to the requesting application. Otherwise, it sends back a security exception to the application.

Android applications have to be modified to make them use ASM simulator for resource access instead of existing Android system. The grey and white box on the left-top in Figure 5 indicates Android applications developed in such a way that they should ask resources from ASM simulator instead of existing Android system.

6.2. Context-aware Android role-based access control implementation

Context-aware Android role-based access control system is implemented using four of the Android application components: activities, services, content providers, and broadcast receivers. It consists of various components that altogether perform these four main operations: policy configuration, context detection, policy decision, and storage.

Policy configuration. PCM is the component of CA-ARBAC that provides user interfaces for policy configurations. As such, it is made up of many Android Activity classes that allow the user to perform various activities. It consists of classes used for role creation, role assignment, and role modification. Role creation involves giving appropriate name to the role, assigning one or more permissions to the role, and associating context with the permissions (in the case where the user is interested to associate context with permissions).

When a new role is created, it is stored in PADB. PADB and all other databases in our system are implemented using SQLite database. Role assignment is assigning roles to applications. When a role is assigned to an application, the data is saved in AADB. Role modification enables the user to modify existing roles.

Context detection. This part consists of the context manager and CDB. The list of contexts defined in the system and their current values is kept in CDB. The context manager is an Android service class that works continuously in the background. It constantly collects current context information from different context sources and updates the values in CDB. To be able to do so, it implements different kinds of listeners such as Android `LocationListener`. Context collection does not affect the performance of the other parts of our system because the service runs on a separate process independent of the other components. Moreover, not to harm the overall performance of the mobile device, it is possible to adjust the frequency at which the context manager collects context data.

The context manager service is started at boot time by the `ContextManagerStarter` class that extends Android `BroadcastReceiver` class. Broadcast receivers can register for `Intent.ACTION_BOOT_COMPLETED` system intent that tells the device has completed booting. Our `ContextManagerStarter` class is also registered for this intent. Hence, it starts the context manager service when it receives the `Intent.ACTION_BOOT_COMPLETED` intent.

Policy decision. Upon the arrival of request message from ASM simulator, the PDM performs policy decisions based on the entries in the AADB, PADB, and CDB. The PDM extends Android content provider class. It consists of various methods such as `checkAppPermission()`, `getAppRoles()`, `getRolePermissions()`, and `checkContext()`. `checkAppPermission()` is the main method that checks whether the application should be currently granted a given permission or not. It uses `getAppRoles()` to obtain the roles of the application from AADB and `getRolePermissions()` to retrieve the permissions assigned to the roles of the application from PADB. Finally, `checkContext()` is used to check if the preconfigured context is satisfied or not.

7. EXAMPLE USE CASES

In this section, we further demonstrate our system by using some real examples. We explain the working of our system using three applications, three roles, and four kinds of contexts. We developed three test applications for this purpose. The first one is a messenger application called `PhoneCaller` that allows phone call, SMS sending, and audio recording. The second one is called `PhotoEditor`. It is a photography application that allows taking photos and editing them. The last application is a simple application that obtains the users current location and displays it. We named it `LocationGetter`. The three roles we created for our test are `MESSENGER`, `PHOTOGRAPHY`, and `TRAVEL` roles. Four types of contexts, namely, `LOCATION`, `CALL_STATE`, `SCREEN_STATE`, and `TIME` are used for this test.

7.1. Creating roles

As mentioned earlier, we assign roles to applications based on their functional group. We categorize applications into functional groups and create different roles that are appropriate for each functional category. Being able to create roles, which contain optimum number of permissions, is one of the challenges of our system. For skilled users, we believe that deciding which permissions to assign to which roles is completely up to the user. However, as most users of mobile devices are ordinary users who have difficulty in creating roles, there is a need to create default system roles that can be used as needed. Currently, there is no any reference standard that states which kinds of applications should use which kind of permissions. There is also no satisfactory system that can identify the permissions appropriate for the different categories of applications. This topic by itself is a new research area that needs further study.

However, there are few works such as [24–28] that have performed limited researches on this topic. Most of these studies used this methodology as a way of detecting malicious Android applications. Among them, we found [24] to be more convenient for our work. We used their open source application called `SuspiciousAppsChecker` to find sample application roles and corresponding permissions. `SuspiciousAppsChecker` is an application that analyzes Android applications for overprivilege. It checks Android applications for overprivilege by comparing the permissions used by the applications with a predefined permission list allowed for the category that the application belongs. We recognize that this is not sufficient way of generating roles for applications. First of all, the methodology by itself may not be taken as a good means of dealing with this problem. Secondly, `SuspiciousAppsChecker` is not yet mature and has limitations. To mention one, the categorization is too general and not fine-grained. For instance, the system assumes that all messenger applications belong to the same category and believes that all messenger applications should be given the same set of permissions. In reality, there are various kinds of messenger applications such as text messaging applications and voice messaging applications. For example, `RECORD_AUDIO` permission is not necessary for text messaging applications but is must for voice messaging applications. So it is wrong to group all messenger applications into one category and assign them the same set of permissions.

In the future, we have a plan to develop a system that can automatically classify permissions based on application functionality. We also hope that a better automated technique may be discovered by other researchers. Nonetheless, for the purpose of explaining our model, we believe that it is adequate to use simple samples developed with the help of `SuspiciousAppsChecker` because our main goal in this paper is not identifying roles and equivalent permissions but rather showing that CA-RBAC can be used to provide usable privacy-preserving permission system. The three roles, the permissions they contain, and the contexts associated with them are shown in Table III earlier.

7.2. Associating contexts

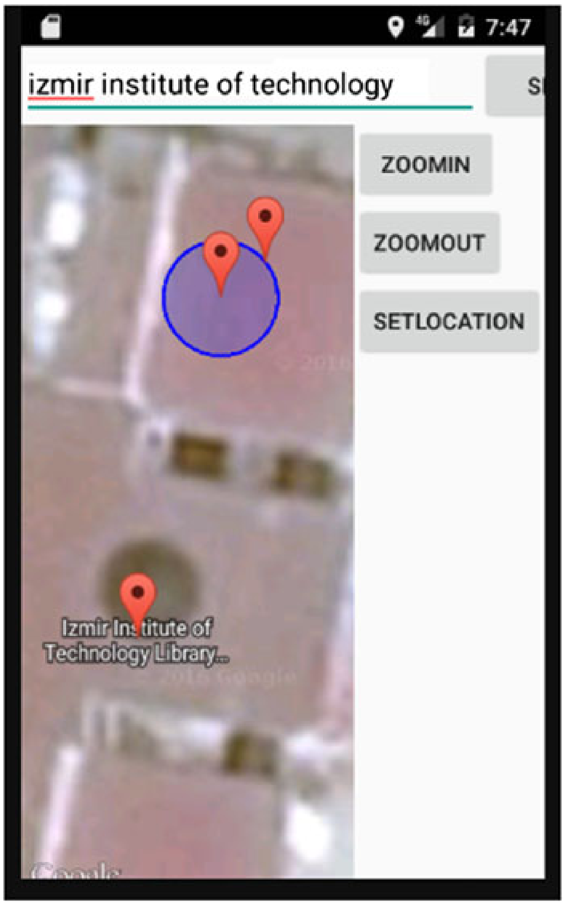
We associated contexts with three of the permissions assigned to our roles as shown in Table III. For example, `RECORD_AUDIO` permission is a desirable permission for multimedia and messenger applications. However, it may be very dangerous at some conditions such as when the user is in a meeting, or if the user is talking on phone and if the phone is locked. Therefore, we set a policy saying that `RECORD_AUDIO` permission is not allowed if the user is in one of these situations. To know that the user is in a meeting, we may need to know the meeting place and time. Hence, it is expressed using a combination of LOCATION and TIME contexts. For instance, let us say that the user John has meeting at İzmir Institute of Technology (IYTE)

Table III. Example application roles, permissions, and contexts.

Role	Permissions	Context policy	Context policy type
Messenger	RECORD_AUDIO	USER IS IN A MEETING(LOCATION+TIME)	DENY
		USER IS TALKING ON THE PHONE	DENY
		PHONE IS LOCKED	DENY
	READ_CONTACTS		
	WRITE_CONTACTS		
	CALL_PHONE	PHONE IS LOCKED	DENY
	SEND_SMS	PHONE IS LOCKED	DENY
	RECEIVE_SMS	PHONE IS LOCKED	DENY
	READ_SMS	PHONE IS LOCKED	DENY
	⋮	⋮	⋮
Travel	INTERNET		
	ACCESS COURSE LOCATION	USER IS NOT AT HOME	ALLOW
	ACCESS_FINE_LOCATION	USER IS NOT AT HOME	ALLOW
	⋮	⋮	⋮
Photography	CAMERA	USER IS NOT AT HOME	ALLOW
	WRITE_EXTERNAL_STORAGE		
	READ_EXTERNAL_STORAGE		
	⋮	⋮	⋮

Computer Engineering Department meeting room every Monday and Friday from 2:30 PM to 4:30 PM and he does not want applications to record audio while he is in a meeting. The snapshot in Figure 6 shows how location context policy can be configured in our system. The user sets a circular area by selecting two points on the map. Configuring time context involves selecting time range and days. The context policy for the previous context is represented in our system as $\langle([LOCATION = 38.32; 26.64; 38.32; 26.64] \wedge [TIME = 1430; 1630; MONDAY, FRIDAY]), \mathrm{DENY}\rangle$.

Moreover, John also does not want applications to record audio if he is talking on phone or if his phone is locked. The context policy for these two situations looks like this in our system $\langle[CALL\_STATE = CALL\_STATE\_OFFHOOK], DENY\rangle$ and $\langle[SCREEN\_STATE = SCREEN\_STATE\_OFF], DENY\rangle$, respectively. Similarly, the user John specified a context rule that says that `CALL_PHONE`, `SEND_SMS`, and `RECEIVE_SMS` permissions are not allowed if his phone is locked. Finally, John stated that `ACCESS_FINE_LOCATION` and `CAMERA` permissions are allowed only if he is out of his home such as in cafeteria, markets, and workplace. Let us assume that John's workplace is at IYTE Computer Engineering Department and his home is at İnciraltı Atatürk Student Dormitory.


7:47
izmir institute of technology
ZOOMIN
ZOOMOUT
SETLOCATION
Izmir Institute of Technology Library...
Figure 6. Location context configuration.

7.3. Assigning roles to applications and making experimental tests

In this section, we first see role assignment to our test applications, and then, we conduct an experiment by running our test applications in different contexts.

Let say John who has configured the prior contexts has installed our three applications. John then has to assign roles to the applications based on the permissions they require. Our test applications require different number of permissions. The `PhoneCaller` application, which represents a messenger application, requires the highest number of risky permissions. It usually requires most of the permissions in `MESSENGER` role. Moreover, messengers may require `CAMERA` permission that is in the `PHOTOGRAPHY` role. Messengers also usually need access to location. Location permission is contained in `TRAVEL` role. Hence, John assigned `PhoneCaller` all the three roles in the system. The other two applications each is assigned one role; `PhotoEditor` is assigned `PHOTOGRAPHY` role and `LocationGetter` is assigned `TRAVEL` role. We did various tests based on John's policy configurations as follows:

Firstly, we performed the following four tests using `PhoneCaller` application:

TEST 1. We tried to record audio using PhoneCaller at IYTE Computer Engineering Department meeting room on Monday and Friday between 2:30 PM and 4:30 PM. We also checked if we can record audio at some other contexts.

RESULT 1. The result shows that we are not able to record audio on the given days and time. PhoneCaller displays a "MediaRecorder not found" message. Moreover, a security exception is thrown by ASM simulator as shown in Figure 7. However, we can record audio at other contexts.

TEST 2. We tried to record audio while there is an ongoing phone call. We also tried to record audio while the phone is idle.

RESULT 2. We cannot record audio on the first case but we are able to record audio for the second scenario.

TEST 3. We again tried to record audio while the phone's screen is on. We also tried to record audio while the phone is locked.

RESULT 3. It is possible to record audio for the former case but not for the latter case.

TEST 4. We checked if it is possible to call a phone, send and receive SMS while the phone is locked. We also tested if the same thing may happen when the phone is unlocked.

RESULT 4. The result shows that it is possible to call phone, send and receive SMS when the phone is unlocked but not possible for the opposite case.

Secondly, we made a single test using `PhotoEditor` application.

TEST 5. We tried to take photos at İnciraltı Atatürk Student Dormitory and also at IYTE Computer Engineering Department.

RESULT 5. `PhotoEditor` can take photos when we are at IYTE Computer Engineering Department but not at İnciraltı Atatürk Student Dormitory.

Finally, we made a test using `LocationGetter` application.

TEST 6. We tried to obtain the current location of the user at IYTE Computer Engineering Department and outside of it.

RESULT 6. We can obtain the location of the user outside of IYTE Computer Engineering Department but not inside it.

8. FORMAL VERIFICATION

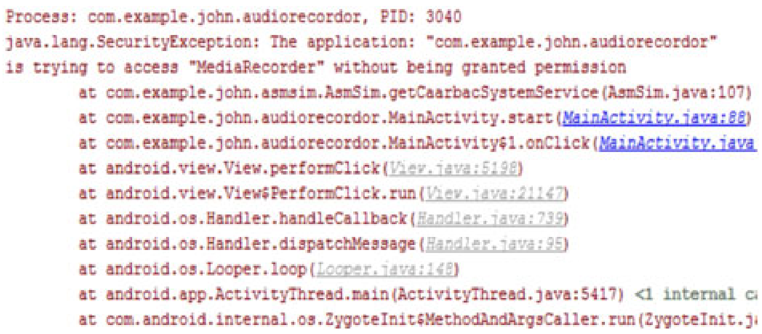
In the previous sections, we showed how our system enhances user privacy in APS by enabling dynamic permission granting and revoking. However, this is not enough to show that the system is valid. One of the important criteria for access control policies is the ability to prevent unauthorized access to resources. An access control policy should not also deny access to resources for actors that possess the right privilege. In this section, we present the formal expression of our access control policy to prove that our system allows only authorized applications to obtain access to permissions. In our model, an application is authorized to use a given permission if all of the four properties in the following are fulfilled:

- If there exists a role that the application is assigned.
- If that role contains the requested permission.
- If the context (if any) associated with the given permission is satisfied.
- If there does not exist any other role containing the same permission and also assigned to the same application but the context associated with the permission is not satisfied.

The last rule is required because a single permission can appear in different roles and hence can have different context policies associated with it for different roles. For example, permission P can have context policy $\mathtt{CP}_1$ for role R_1. The same permission may be associated with another context policy $\mathtt{CP}_2$ when assigned to role R_2. Hence, for instance, if application A is assigned the two roles R_1 and R_2, allowing permission P for application A requires that both of the two contexts $\mathtt{CP}_1$ and $\mathtt{CP}_2$ be satisfied. Otherwise, if application A is allowed to use permission P based on the satisfaction of only one of the contexts, it leads to contradiction.

We demonstrate the formal expression of our model based on the previous definitions covered in Section 4: `A`: set of applications, `R`: set of roles, `P`: set of permissions, `CP`: set of context policies, `ARM`: application role mapping, and `RPM`: role permission mapping. We also include more definitions in this section as follows:

- $\mathtt{AssignedApps}(r : \text{Role}) \rightarrow 2^{\mathtt{A}}$ is the mapping of a set of applications to role r, that is, $\mathtt{AssignedApps}(r) = \{a \in A | (a, r) \in \mathtt{ARM}\}$
- $\mathtt{AssignedPerms}(r : \text{Role}) \rightarrow 2^{\mathtt{P}}$ is the mapping of a set of permissions to role r, that is, $\mathtt{AssignedPerms}(r) = \{p \in P | (r, p) \in \mathtt{RPM}\}$
- Permission context mapping: Let `PCM` be the list consisting of the mapping between permissions and associated context policies. It contains triplets $\langle \mathtt{P}_i, \mathtt{R}_i, \mathtt{CP}_i \rangle$, where $\mathtt{P}_i \in \mathtt{P}$, $\mathtt{R}_i \in \mathtt{R}$ and $\mathtt{CP}_i \in \mathtt{CP}$.
- $\mathtt{AssociatedContext}(p : \text{Permission}, r : \text{Role}) \rightarrow \mathtt{CP}$ is the mapping of permission p to context policy `CP` for role r, that is, $\mathtt{AssociatedContext}(p, r) = \{\mathtt{cp} \in \mathtt{CP} | (p, r, cp) \in \mathtt{PCM}\}$.
- $\mathtt{ContextState} = \{1, 0\}$ is the set containing the possible outcome of a context policy rule. At any given time, the context policy rule evaluates to either true or false. If it evaluates to true, the `ContextState` is set to 1; otherwise, it is set to 0.
- `ContextStateMapping`: Let `CSM` be the list consisting of the mapping between context policy rules and their states. It contains tuples $\langle \mathtt{CP}_i, \mathtt{CS} \rangle$ where $\mathtt{CP}_i \in \mathtt{CP}$ and $\mathtt{CS} \in \mathtt{ContextState}$.
- $\mathtt{ActiveContextPolicies} = \{cp \in \mathtt{CP} | (cp, 1) \in \mathtt{CSM}\}$.



```
Process: com.example.john.audiorecordor, PID: 3040
java.lang.SecurityException: The application: "com.example.john.audiorecordor"
is trying to access "MediaRecorder" without being granted permission
        at com.example.john.asmsim.AsmSim.getCaarbacSystemService(AsmSim.java:107)
        at com.example.john.audiorecordor.MainActivity.start(MainActivity.java:88)
        at com.example.john.audiorecordor.MainActivity$1.onClick(MainActivity.java
        at android.view.View.performClick(View.java:5198)
        at android.view.View$PerformClick.run(View.java:21147)
        at android.os.Handler.handleCallback(Handler.java:739)
        at android.os.Handler.dispatchMessage(Handler.java:95)
        at android.os.Looper.loop(Looper.java:148)
        at android.app.ActivityThread.main(ActivityThread.java:5417) <1 internal c
        at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.j
```

Figure 7. Security exception thrown by Android security modules simulator.

$$
\begin{aligned}
&(\forall a : Application)(\forall p : Permission):\\
&allow(a, p) \Rightarrow\\
&(\exists r : Role)(\exists p : Permission):\\
&\left[a \in AssignedApps(r) \wedge p \in AssignedPerms(r)\right] \wedge\\
&\begin{bmatrix}(Associated\ Context\ (p, r) = \emptyset) \vee\\ (Associated\ Context\ (p, r) \in ActiveContextPolicies)\end{bmatrix} \wedge\\
&\neg\begin{bmatrix}(\exists r' : Role) : (r' \neq r) \wedge a \in AssignedApps(r') \wedge p \in AssignedPerms(r') \wedge\\ Associated\ Context\ (p, r') \in InactiveContextPolicies()\end{bmatrix}
\end{aligned}
$$

Figure 8. Formal expression for run time authorization in context-aware Android role-based access control.

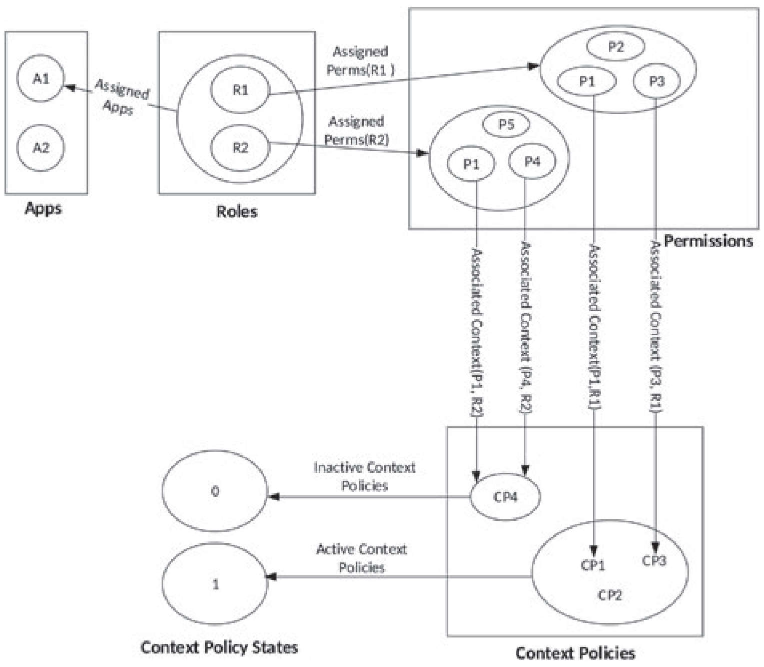


Figure 9. A_1 cannot use P_1 because even if $A_1 \in \texttt{AssignedApps}(R_1) \wedge P_1 \in \texttt{AssignedPerms}(R_1) \wedge \texttt{AssociatedContext}(P_1, R_1) \in \texttt{ActiveContextPolicies}$, there is another contradicting rule, that is, $A_1 \in \texttt{AssignedApps}(R_2) \wedge P_1 \in \texttt{AssignedPerms}(R_2) \wedge \texttt{AssociatedContext}(P_1, R_2) \in \texttt{InactiveContextPolicies}$. Similarly, A_1 cannot use P_4 because $A_1 \in \texttt{AssignedApps}(R_2) \wedge P_4 \in \texttt{AssignedPerms}(R_2) \wedge \texttt{AssociatedContext}(P_4, R_2) \in \texttt{InactiveContextPolicies}$. However, A_1 can use P_2 and P_5 because $A_1 \in \texttt{AssignedApps}(R_1) \wedge P_2 \in \texttt{AssignedPerms}(R_1) \wedge \texttt{AssociatedContext}(P_2, R_1) = \emptyset$ and also $A_1 \in \texttt{AssignedApps}(R_2) \wedge P_5 \in \texttt{AssignedPerms}(R_2) \wedge \texttt{AssociatedContext}(P_5, R_2) = \emptyset$. Moreover, A_1 can use P_3 because $A_1 \in \texttt{AssignedApps}(R_1) \wedge P_3 \in \texttt{AssignedPerms}(R_1) \wedge \texttt{AssociatedContext}(P_3, R_1) \notin \texttt{ActiveContextPolicies}$.

- $\texttt{InactiveContextPolicies} = \{cp \in \texttt{CP}|(cp, 0) \in \texttt{CSM}\}$.

Hence, the run-time authorization decision in our system is governed by the formal expression in Figure 8, and an example is provided in Figure 9.

9. PERFORMANCE TESTS

In this section, we analyze the additional performance cost incurred by CA-ARBAC on the mobile device and on applications' resource access time. Concerning the cost of device performance, we performed two kinds of performance tests. We took a measurement of CA-ARBAC memory and CPU consumption by running it for high resource-consuming operations (worst case) such as contextual location setting by using Google maps. After monitoring its execution over long time frame, we have observed that the average memory space and CPU utilized by CA-ARBAC for the heaviest operations is approximately 6 MB and 5%, respectively, as shown in Figures 10 and 11.

Context-aware Android role-based access control also introduces some time delay on applications' run time resource access. This additional time delay is caused by callback to CA-ARBAC during applications "access to

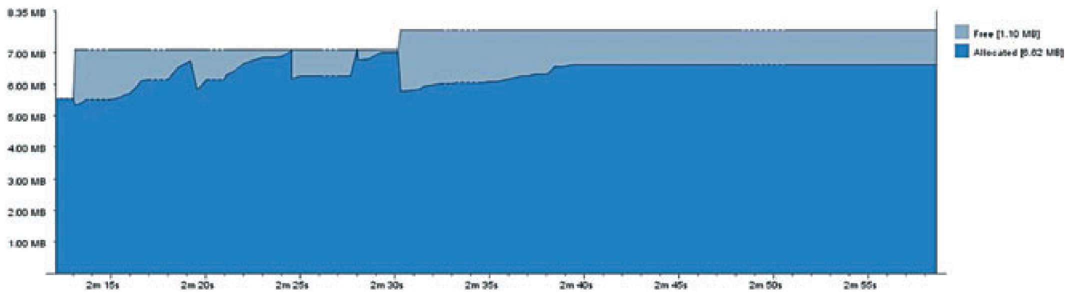


Figure 10. Context-aware Android role-based access control memory usage statistics.

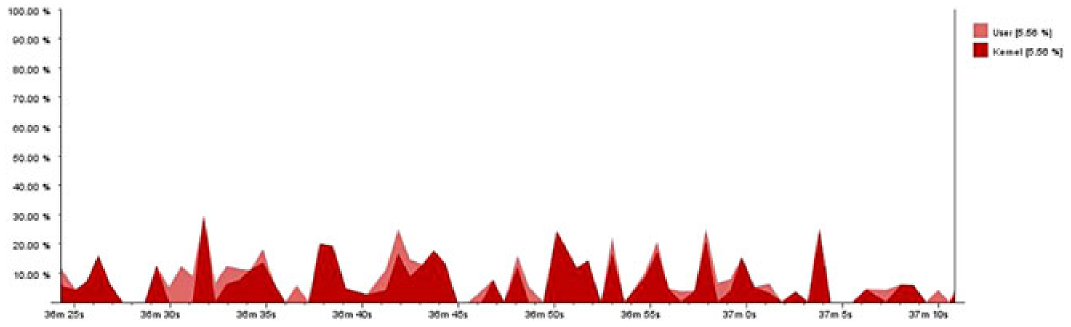


Figure 11. Context-aware Android role-based access control CPU consumption statistics.

Table IV. Record audio permissions run time access delay.

Measurement	1	2	3	4	5	6	7	8	9	10	Average
Callback time in ms	7	6	13	4	25	7	8	10	2	10	9.2

permissions." This extra time delay may vary from permission to permission and also from time to time. To show an example of the time delay, we took a record of the time delay introduced when our `PhoneCaller` application tries to access `RECORD_AUDIO` permission as shown in Table IV. There is an average delay of approximately 9.2 ms based on 10 measurements taken at different times. Finally, we would like to mention that the disk space used by CA-ARBAC is 3.94 MB.

10. FUTURE WORKS

Because of the various restrictions we have, there are some issues that we are forced to postpone for the future. The following are some of the future works that we plan to accomplish in our next work.

Active context management. In our current system, we use passive context management. In passive context management, once an application is granted permission, it can use it irrespective of changes in context information. However, for realistic situations, the application should be revoked access if the context changes during the time that the application is using the permission. We could not use active context management currently in CA-ARBAC as the underlying Android security policy enforcement framework does not support context management. ASM framework also does not support context management. For the future, we suggest the integration of security APIs similar to ASM that support active context management.

Usability test. We argue that our system can be better in usability than the existing APS. But, this claim needs to be confirmed with real user studies that measure usability.

Default system roles. Skilled users can easily create roles on their own. This might not be an easy task when it comes to naive users. The situation becomes more difficult if the user has to configure context for permissions. One of the solutions for this problem can be having system default roles. Creating default roles requires that the roles should contain optimum number of permissions and context configuration. As explained earlier, we used a rough method of grouping applications into functional groups to create roles. We believe that this is not the only way to so do. For example, we can think of an automated system that can analyze applications and suggest roles to the user.

11. CONCLUSION

In this paper, we proposed a new access control model for APS to protect user's privacy. Our model allows dynamic permission granting and revoking based on predefined contexts. Our system is a variety of CA-RBAC designed in such a way that roles, which will consist of a set of Android permissions, are assigned to applications and contexts are associated with permissions. This is different from existing CA-RBAC models that associate contexts with roles in contrary to our system that associates contexts with permissions. We believe that our proposed system can provide better privacy without significant effect on the usability of the permission system. Our system can also provide dynamic and fine-grained permission system for Android versions earlier than Android version 6. We designed a novel architecture for our proposed system. We also developed and implemented a prototype application called CA-ARBAC. We made various tests using our prototype application and obtained expected results.

Acknowledgement

Juhar Abdella has been supported by a YTB scholarship during his thesis studies.

REFERENCES

1. IDC. Worldwide smartphone shipments edge past 300 million units in the second quarter; Android and iOS devices account for 96% of the global market August 2014. http://www.idc.com/getdoc.jsp?containerid=prUS25037214 last accessed on December 2015.
2. Mobile cyber threats. *Technical Report*, Kaspersky Lab and INTERPOL Joint Report October 2014. http://media.kaspersky.com/pdf/Kaspersky-Lab-KSN-Report-mobile%2Dcyberthreats-web.pdf, last accessed on December 2015.
3. Backes M, Gerling S, Hammer C, Maffei M, von Styp-Rekowsky P. AppGuard—fine-grained policy enforcement for untrusted Android applications. In *Data Privacy Management and Autonomous Spontaneous Security*, vol. 8247, Garcia-Alfaro J, Lioudakis G, Cuppens-Boulahia N, Foley S, Fitzgerald WM (eds), Lecture Notes in Computer Science. Springer Berlin Heidelberg: Woburn, MA, 2014; 213–231.
4. Nauman M, Khan S, Zhang X. Apex: extending Android permission model and enforcement with user-defined runtime constraints. *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10,ACM, New York, NY, USA, 2010; 328–332.
5. Cappos J, Wang L, Weiss R, Yang Y, Zhuang Y. BlurSense: dynamic fine-grained access control for smartphone privacy. *Sensors Applications Symposium (SAS), 2014 IEEE*, Queenstown, 2014; 329–332.
6. Stelly CD. Dynamic user defined permissions for Android devices. *Master's Thesis*, Dept. of Computer Science, University of New Orleans, LA, USA, 2013.
7. Bugiel S, Heuser S, Sadeghi AR. Flexible and fine-grained mandatory access control on Android for diverse security and privacy policies. *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, USENIX, Washington, D.C., 2013; 131–146.
8. Zhou Y, Zhang X, Jiang X, Freeh VW. Taming information-stealing smartphone applications (on Android). *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11, Springer-Verlag, Berlin, Heidelberg, 2011; 93–107.
9. Beresford AR, Rice A, Skehin N, Sohan R. MockDroid: trading privacy for application functionality on smartphones. *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, ACM, New York, NY, USA, 2011; 49–54.
10. Jeon J, Micinski KK, Vaughan JA, Fogel A, Reddy N, Foster JS, Millstein T. Dr. Android and Mr. Hide: fine-grained permissions in Android applications. *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, ACM, New York, NY, USA, 2012; 3–14.
11. Shebaro B, Oluwatimi O, Bertino E. Context-based access control systems for mobile devices. *IEEE Transactions on Dependable and Secure Computing* 2015; **12**(2): 150–163.
12. Bai G, Gu L, Feng T, Guo Y, Chen X. Context-aware usage control for Android. In *Security and Privacy in Communication Networks, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, Vol. 50, Jajodia S, Zhou J (eds). Springer Berlin Heidelberg: Berlin, Heidelberg, 2010; 326–343.
13. Conti M, Crispo B, Fernandes E, Zhauniarovich Y. CRêPE: a system for enforcing fine-grained context-related policies on Android. *IEEE Transactions on Information Forensics and Security* 2012; **7** (5): 1426–1438.
14. Miettinen M, Heuser S, Kronz W, Sadeghi AR, Asokan N. ConXsense: automated context classification for context-aware access control. *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, ACM, New York, NY, USA, 2014; 293–304.

15. Felt AP, Ha E, Egelman S, Haney A, Chin E, Wagner D. Android permissions: user attention, comprehension, and behavior. *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, ACM, New York, NY, USA, 2012; 3:1–3:14.
16. Heuser S, Nadkarni A, Enck W, Sadeghi A R. ASM: a programmable interface for extending Android security. *23rd USENIX Security Symposium (USENIX Security 14)*, USENIX Association, San Diego, CA, 2014; 1005–1019.
17. Guo T, Zhang P, Liang H, Shao S. Enforcing multiple security policies for Android system. *2Nd International Symposium on Computer, Communication, Control and Automation*, Atlantis Press, Singapore, 2013; 165–169.
18. Rohrer F, Zhang Y, Chitkushev L, Zlateva T. DR BACA: dynamic role based access control for Android. *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, ACM, New York, NY, USA, 2013; 299–308.
19. Yee TTW, Thein N. Leveraging access control mechanism of Android smartphone using context-related role-based access control model. *2011 7th International Conference on Networked Computing and Advanced Information Management (NCM)*, Gyeongju, 2011; 54–61.
20. Choi J H, Jang H, Eom YI. CA-RBAC: context aware RBAC scheme in ubiquitous computing environments. *Journal of information science and engineering* 2010; **26**(5): 1801–1816.
21. Jung K, Park S. Context-aware role based access control using user relationship. *International Journal of Computer Theory and Engineering* 2013; **5** (3): 533–537.
22. Rohrer F, Zhang Y, Chitkushev L, Zlateva T. Role based access control for Android (RBACA). *Technical Report*, Boston University, MA, USA, 2012. http://www.acsac.org/2012/program/posters/poster09.pdf Access Date: 11.12.2015.
23. Backes M, Bugiel S, Gerling S, von Styp-Rekowsky P. *Android security framework: enabling generic and extensible access control on Android*, 2014. arXiv preprint arXiv:1404.1395.
24. Varga J, Muska P. Presenting risks introduced by Android application permissions in a user-friendly way. *Tatra Mountains Mathematical Publications* 2014; **60**(1): 85–100.
25. Mylonas A, Theoharidou M, Gritzalis D. Assessing privacy risks in Android: a user-centric approach. In *Risk Assessment and Risk-Driven Testing, Lecture Notes in Computer Science*, Vol. 8418, Bauer T, Großmann J, Seehusen F, Stølen K (eds). Springer International Publishing: Berlin, Heidelberg, 2014; 21–37.
26. Qadir MZ, Jilani AN, Sheikh HU. Automatic feature extraction, categorization and detection of malicious code in Android applications. *International Journal of Information and Network Security (IJINS)* 2014; **3**(1): 12–17.
27. Zhou Y, Jiang X. Dissecting Android malware: characterization and evolution. *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, IEEE Computer Society, Washington, DC, USA, 2012; 95–109.
28. Barrera D, Kayacik HG, van Oorschot PC, Somayaji A. A Methodology for empirical analysis of permission-based security models and its application to Android. *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, ACM, New York, NY, USA, 2010; 73–84.