Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints

Mohammad Nauman
Institute of Management Sciences, Pakistan
nauman@imsciences.edu.pk

Sohail Khan
School of Electrical Engineering and Computer Science, NUST Pakistan
sohail.khan@seecs.edu.pk

Xinwen Zhang
Samsung Information Systems America, USA
xinwen.z@samsung.com

ABSTRACT

Android is the first mass-produced consumer-market open source mobile platform that allows developers to easily create applications and users to readily install them. However, giving users the ability to install third-party applications poses serious security concerns. While the existing security mechanism in Android allows a mobile phone user to see which resources an application requires, she has no choice but to allow access to all the requested permissions if she wishes to use the applications. There is no way of granting some permissions and denying others. Moreover, there is no way of restricting the usage of resources based on runtime constraints such as the location of the device or the number of times a resource has been previously used. In this paper, we present Apex – a policy enforcement framework for Android that allows a user to selectively grant permissions to applications as well as impose constraints on the usage of resources. We also describe an extended package installer that allows the user to set these constraints through an easy-to-use interface. Our enforcement framework is implemented through a minimal change to the existing Android code base and is backward compatible with the current security mechanism.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Access controls*

General Terms

Security

Keywords

Mobile platforms, Android, Policy Framework, Constraints

1. INTRODUCTION

In the current scenario of mobile platforms, Android [3] is among the most popular open source and fully customizable software stacks for mobile devices. Introduced by Google, it includes an operating system, system utilities, middleware in the form of a virtual machine, and a set of core applications including a web browser, dialer, calculator and a few others.

Third party developers creating applications for Android can submit their applications to Android Market from where users can download and install them. While this provides a high level of availability of unique, specialized or general purpose applications, it also gives rise to serious security concerns. When a user installs an application, she has to trust that the application will not misuse her phone's resources. At install-time, Android presents the list of permissions requested by the application, which have to be granted if the user wishes to continue with the installation. This is an all-or-nothing decision in which the user can either allow *all* permissions or give up the ability to install the application. Moreover, once the user grants the permissions, there is no way of revoking these permissions from an installed application, or imposing constraints on how, when and under what conditions these permissions can be used.

Consider a weather update application that reads a user's location from her phone and provides timely weather updates. It can receive location information in two ways. It may read it automatically from GPS or prompt the user to manually enter her location if GPS is unavailable. In Android, the application must request permission to read location information at install-time and if the user permits it, the application has access to her exact location even though such precision is not necessary for providing weather updates. If however, she denies the permission, the application cannot be installed. The user therefore does not have a choice to protect the privacy of her location if she wishes to use the application for which the exact location isn't even necessary and the application itself provides an alternative.

To address these problems, we have developed *Android Permission Extension (Apex)* framework, a comprehensive policy enforcement mechanism for the Android platform. Apex gives a user several options for restricting the usage of phone resources by different applications. The user may grant some permissions and deny others. This allows the user to use part of the functionality provided by the application while still restricting access to critical and/or costly resources. Apex also allows the user to impose runtime constraints on the usage of resources. Finally, the user may wish to restrict the usage of the resources depending on an application's use e.g., limiting the number of SMS messages sent each day. We define the semantics of Apex as well as the policy model used to describe these constraints. We also de-

ASIACCS'10 April 13–16, 2010, Beijing, China.

scribe an extended package installer which allows end-users to specify their constraints without having to learn a policy language. Apex and the extended installer are both implemented with a minimal and backward compatible change in the existing architecture and code base of Android for better acceptability in the community.

2. BACKGROUND

2.1 Android Architecture

Android architecture is composed in layers. These are the application layer, application framework layer, Android runtime and system libraries. *Applications* are composed of one or more different *components*. There are four types of components namely *activities*, *services*, *broadcast receivers* and *content providers*. Activities include a visible interface of the application. Service components are used for background processing which does not require a visible interface. The broadcast receiver component receives and responds to messages broadcast by application code. Finally, content providers enable the creation of a custom interface for storing and retrieving data in different types of data stores such as filesystems or SQLite databases. The *application framework layer* enables the use or reuse of different low-level components. Android also includes a set of system libraries, which are used by different components of Android.

Components of an application can interact with other components – both within the application and outside it – using a specialized *inter-component communication* mechanism based on *Intents*. An intent is "an abstract representation of an action to be performed" [4]. Intents can either be sent to a specific component – called *explicit intents* – or broadcast to the Android framework, which passes it on to the appropriate components. These intents are called *implicit intents* and are much more commonly used. Both of these types share the same permission mechanism and for the sake of clarity, we only consider implicit intents in this paper.

2.2 Motivating Example

In order to demonstrate the existing Android security framework and its limitations, we have created a set of four example applications as a case study, which is representative of a large class of applications available in the Android Market. Ringlet is a sample application that performs several tasks using different low-level components like GPRS, MMS, GPS etc. It accesses three other applications, each gathering data from a different social network – facebook, twitter and flickr. On receiving user name/password pairs, Ringlet passes on the username and passwords of the social networks to their respective back-end services. The back-end services connect the user to the three networks at the same time and extract updates from the social network sites to their respective content provider datastores on the phone. The front-end GUI receives messages from the content providers, displays these messages to the user in one streamlined interface and allows her to reply back to the messages or forward these messages to a contact via SMS or MMS. It should be noted that several applications similar to Ringlet are available on the Android Market that use several permissions such as sending SMS and accessing the location of the user. If a user downloads several applications for different purposes and grants all requested permissions to all applications, there is no way of ensuring that none of the applications will misuse these permissions.

In essence, there are four issues: (1) The user has to grant all permissions in order to be able to install the application; (2) there is no way of restricting the extent to which an application may use the granted permissions; (3) since all permissions are based on singular, install-time checks, access to resources cannot be restricted based on dynamic constraints such as the location of the user or the time of the day; and (4) the only way of revoking permissions once they are granted to an application is to uninstall the application.

We address these issues by enhancing the existing security architecture of Android for enabling the user to restrict the usage limit of both newly installed applications as well as applications installed in the past. In the following section, we formally describe a policy model for this purpose and then detail how it has been incorporated in the existing security mechanism of the Android framework.

3. ANDROID USAGE POLICIES

In this section, we first present a logical model of the existing Android security mechanism that focuses on the semantics of *Inter-Component Communication (*ICC*)*. The model covers the semantics of intents, intent filters and the permission logic for granting or denying access to resources. Afterwards, we describe the policy model used for extending the permission mechanism of Android to incorporate user-defined dynamic constraints.

In traditional access control models, policies revolve around the abstractions of subjects, objects and rights. In system-level permission models, policies are based on processes, users, resources and rights. Android's security framework differs slightly from both of these approaches in that 1) the security model differentiates between the different modules of a single application and 2) there is usually only one user per device.

Each application consists of different modular portions termed as *components*. An application a_1 might be allowed access to one component of application a_2 but not another. This allows an application to make parts of its functionality publicly available to other applications while keeping the rest of the components protected. In this way, the smallest unit of an application, with respect to the Android security framework, is the component. We therefore define our security framework on the basis of components of applications.

DEFINITION 1 (APPLICATIONS AND COMPONENTS). *The set of applications and components in Android are denoted by A and C respectively and a component association function $\varsigma : C \rightarrow A$ associates each component with a unique application.*

Inter-Component Communication (ICC) in Android is accomplished through the concept of Intents. Intents encapsulate the information associated with the ICC call. The *action string* describes the action to be performed, *data* acts an argument for the action, *category* specifies the type of the component that should handle the intent and the *extras* field includes other arbitrary information associated with the raised intent. For example, in our motivating example (cf. Section 2.2), an intent of action string `edu.apex.android.ringlet.fkringlet.POST`, data `"New Image Caption"`, category `"edu.apex.android.category.CATEGORY_-`

PHOTO" and an image in the extras field can be used to post an image to the user's flickr profile. Formally,

DEFINITION 2 (INTENT). *An intent is a 4-tuple* $(\alpha, \sigma, \gamma, \epsilon)$*, where* α *is an action string describing the action to be performed,* σ *is a string representing the data,* γ *is the string representing the category and* $\epsilon : name \rightarrow val$ *is a function that maps names of extra information to their values. The set of intents is denoted as* I*.*

Any application willing to 'serve' an intent describes its willingness using the <IntentFilter> tag in the manifest file. An *intent filter* can be used to describe the fine grained details of the intent an application is willing to serve including the action string, data and the category of the intent. We formalize intent filters using action strings that they serve. This allows for a cleaner formalization without lack of generality. We define an intent filter as:

DEFINITION 3 (INTENT FILTER). *An intent filter is an application's willingness to serve an intent. Intent filters are associated with individual components of applications. An intent-filter association function* $A_f : C \rightarrow 2^F$ *maps each component of an application to a set of intent filters where* F *is the set of intent filters and* $I \subseteq F$*. If a component* $c \in C$ *has an intent filter* f*, we write* $f \in A_f(c)$*.*

EXAMPLE 1. The flickr service exposes the action string edu.apex.android.intents.FKS_INTENT in its intent filter. This intent filter catches all intents matching this action string, which can then be used to start the flickr service.

Components associated with intent filters may impose restrictions on which applications may call them. These restrictions are defined using the concept of *permissions*.

When an application provider writes an application, she provides a list of intent filters that are supported by different components of the application. Moreover, she can associate permissions with the individual components of the application. This would ensure that only an application that possesses the required permissions can call the component through the given intent. Permissions are first declared (as a unique string in <Permission> tag) and then associated with a component using android:permission attribute of <activity>, <service>, <receiver> or <provider> tag [6]. Note that these are the permissions *required* by the target component, not those *granted* to the calling component.

DEFINITION 4 (PERMISSIONS). *A permission declares the requirements posed by a component for accessing it. A permission association function* $A_p : C \rightarrow P$ *associates each component to a single*[1] *permission where* P *is the set of permissions. If a component* $c \in C$ *requires that a calling component have a permission* $p \in P$*, then we write* $p = A_p(c)$*.*

Applications are granted specific permissions by the Android framework at install-time. The manifest file includes one or more <uses-permission> tags that specify the permissions required by the application to function properly. During installation of the application, the user is presented with an interface listing the requested permissions. If the user chooses to grant these permissions, the application is installed and thus granted the requested permissions. Formally:

[1]"A feature can be protected by at most one permission". [5]

DEFINITION 5 (USES-PERMISSIONS). *A uses-permission construct declares the permissions granted to the application by the user at install-time. Permissions are associated with applications rather than their individual components. The basic permission function* $\mu : A \rightarrow 2^P$ *is a function that maps an application to the permissions it is granted where* A *is the set of applications and* P *is the set of permissions.*

Since we base our formalization on the concept of components, we define a second permission function ρ that defines the permission of one component to call another with a specific intent. The permission function is defined as follows:

DEFINITION 6 (PERMISSION FUNCTION). *The permission function* ρ *defines the complete set of conditions under which a component* c_1 *is allowed to call another component* c_2*.* c_1 *can communicate with* c_2 *if and only if either 1) there is no permission associated with the second component or 2) the application to which* c_1 *belongs has been granted the permission required by* c_2*. Formally:*

$$\begin{aligned} \rho(c_1, c_2, i) \iff & A_p(c_2) = null \quad \vee \\ & \exists p \in P, a \in A \cdot a = \varsigma(c_1) \\ & \wedge \quad p = A_p(c_2) \wedge p \in \mu(a_1) \\ & \wedge \quad i \in A_f(c_2) \end{aligned}$$

EXAMPLE 2. The facebook service component declares the permission edu.apex.android.permission.FBS_START in its android:permission attribute. In order to be able to raise the intent for starting the facebook service, the Ringlet activity needs to have this permission granted to it.

This example concludes our formalization of the existing security mechanism provided by Android. It is evident that in this mechanism, there is no way of specifying complex or fine-grained runtime constraints for permissions. Below, we describe how we have enhanced this mechanism to include dynamic constraints on permissions.

3.1 Dynamic Constraints

For associating dynamic constraints with permissions, we introduce the concept of *application attributes*. Each application in Android is associated with a finite set of attributes. The *application state* is a function that maps the attributes of an application to their values. The application state is a persistent structure that maintains its values between different system sessions.

DEFINITION 7 (APPLICATION STATE). *An application state is a function* $\tau : \eta(A) \rightarrow dom(\eta(A))$*, where* A *is the set of applications,* η *is the function that maps an application to a set of attribute names, and* $dom(x)$ *is the value domain of attribute set* x *of an application* $a \in A$*.*

EXAMPLE 3. The Ringlet application has several attributes associated with it such as *sentMms*, which captures the number of MMS that have been sent by the application. Each application can have a different set of attributes. Existing Android applications for which no attributes are defined can be considered as having an empty set of attribute names associated with them.

Constraints for permissions are defined in terms of *predicates* – functions that map the set of application attributes,

system attributes and constants to boolean values. A predicate returns true if and only if the attribute values in the current application state satisfy the conditions of the predicates. We denote the set of predicates as Q.

An application transitions from one state to another as a result of a change in the value of the application's attributes. This change is captured by an attribute update action.

DEFINITION 8 (ATTRIBUTE UPDATE ACTION). *An attribute update action $u(a.x, v') : \tau \rightarrow \tau'$ is a function that maps the value of an attribute $x \in \eta(a)$ of an application $a \in A$ to a new value $v' \in dom(\eta(a))$.*

Attribute updates play a key role in our policy framework. Predicates based on these attributes are used for two purposes. First, they are used to specify the conditions under which a permission may be granted. Second, they can cause an update action to be triggered, which may modify the values of attributes. Conditions and updates are both specified in a *policy*.

DEFINITION 9 (POLICY). *A policy defines the conditions under which an application is granted a permission. It consists of two input parameters – an application and a permission – on which it is applicable, an authorization rule composed of predicates that specify the conditions under which the permission is granted/denied and a set of attribute update actions, which are to be performed if the conditions in the authorization rule are satisfied. Specifically:*

l(a, p):
$q_1 \wedge q_2 \wedge q_3 \wedge \ldots \wedge q_n \rightarrow \{permit, deny\}$
$u_1; u_2; u_3; \ldots; u_n$

where $a \in A$, $p \in P$, $q_i \in Q$, u_i are attribute update actions, $l \in \Lambda$ and Λ is the set of policies in the system. The right-hand-side of the authorization rule defines the value returned by the policy.

Note that if the predicates in an authorization rule are satisfied, updates specified in the policy are performed regardless of the return value of the authorization rule.

A policy is applied to a specific application state. Attribute values in the particular state determine the truth value of the predicates. If the predicates are satisfied, the permission is either granted or denied (depending on the return value of the authorization rule) and the updates specified in the policy are executed resulting in a new state. This may render predicates in other policies true, thus allowing for the dynamic nature of the policy-based constraints on permissions.

We incorporate these policies in the existing security model of Android by redefining the permission function ρ.

DEFINITION 10 (DYNAMIC PERMISSION FUNCTION). *The dynamic permission function specifies the conditions under which a component c_1 is granted permission to call another component c_2 using intent i. It incorporates the static checks as well as the dynamic runtime constraints in its evaluation. For a permission to be granted, Android's permission checks must grant the permission and there must not be a policy that denies the permission. Formally:*

$$\begin{aligned}\rho(c_1, c_2, i) \iff & A_p(c_2) = null \quad \vee \\ & \exists p \in P, a \in A \cdot a = \varsigma(c_1) \\ \wedge \quad & p = A_p(c_2) \wedge p \in \mu(a_1) \\ \wedge \quad & i \in A_f(c_2) \\ \wedge \quad & \neg \exists l \in \Lambda \cdot l(a, p) = deny\end{aligned}$$

EXAMPLE 4. The Ringlet activity is able to include the location of the user in the messages posted. Similar to the weather update example given in Section 1, the user may wish to restrict access to GPS for protecting her privacy. Using dynamic constraints, she may define a policy that denies access to GPS at all times. The constraint in the policy is set to `true` and the authorization rule to `deny` with no updates. Using this policy, she may install Ringlet and use it for all other functionality while still protecting the privacy of her location. Similarly, she may define a policy that imposes a limit on Ringlet's ability to send updates through MMS messages, say 5 each day, thereby controlling the carrier costs at a fine-grained level.

For the implementation of Apex, we have defined a policy language that allows the user to define her policies including dynamic constraints. Figure 1 shows how both of the above example policies can be depicted in our language. We have incorporated the policy model in the existing android security mechanism and inter-component communication mechanism. This includes changes to the *activity manager*, the *package manager* and the permission checking mechanisms associated with these components. However, we are unable to include the details of this implementation here due to space limitations.

4. POLY ANDROID INSTALLER

Writing usage policies is a complex procedure, even for system administrators. Android is targeted at the consumer market and the end users are, in general, unable to write

```
mms_count_allow("edu.apex.ringlet.Ringlet" as Ringlet,
                "android.permission.SEND_SMS" as MMS):
  Ringlet.sentMms <= 5 ∧
  Ringlet.lastUsedDay = System.CurrentDay
                              → permit(Ringlet, MMS);
  Ringlet.sentMms' = Ringlet.sentMms + 1;

mms_count_deny("edu.apex.ringlet.Ringlet" as Ringlet,
                "android.permission.SEND_SMS" as MMS):
  Ringlet.sentMms > 5 ∧
  Ringlet.lastUsedDay = System.CurrentDay
                              → deny(Ringlet, MMS);

reset_mms_count("edu.apex.ringlet.Ringlet" as Ringlet,
                "android.permission.SEND_SMS" as MMS):
  Ringlet.lastUsedDay != System.CurrentDay
                              → permit(Ringlet, MMS);
  Ringlet.lastUsedDay' = System.CurrentDay;
  Ringlet.sentMms' = 1;
```

```
deny_gps("edu.apex.ringlet.Ringlet" as Ringlet,
         "android.permission.ACCESS_FINE_LOCATION" as GPS):
  System.CurrentTime > 1700 ∨ System.CurrentTime < 0900
                              → deny(Ringlet, GPS);
```

```
restrict_internet("edu.ringlet.Ringlet" as Ringlet,
                  "android.permission.INTERNET" as Net):
  true → deny(Ringlet, Net);
```

Figure 1: Example Apex Policies

complex usage policies. One of the most important aspects of our new policy enforcement framework is the usability of the architecture. To this end, we have created *Poly* – an advanced Android application installer. Poly augments the existing package installer by allowing users to specify their constraints for each permission at install time using a simple and usable interface. In the existing Android framework, the user is presented with an interface that lists the permissions required by an application. We have extended the installer to allow the user to click on individual permissions and specify their constraints. When a user clicks on a permission she is presented with an interface that allows her to pick one of a few options.

1. For the novice user, the default setting is to *allow*. The default behavior of Android installer is also to allow all permissions, if the user agrees to install an application. This is a major usability feature that makes the behavior of the existing Android installer a subset of Poly and will hopefully allow for easier adoption of our constrained policy enforcement framework.

2. The *deny* option allows a user to selectively deny a permission as opposed to the all-or-nothing approach of the existing security mechanism. For example, Alice downloads an application that asks for several permissions including the one associated with sending SMS. Alice may wish to stop the application from sending SMS while still being able to install the application and use all other features. In Poly, Alice can simply tap on the 'send SMS' permission and set it to 'deny'.

3. The third option is the constrained permission. This is the main concern of this contribution and has been discussed at length in the previous sections. An important point to note here is that currently, we have incorporated only simple constraints such as restricting the number of times used and the time of the day in which to grant a permission. This simplification is for the sake of usability. We aim to develop a fully functional desktop application, which will allow expert users to write very fine-grained policies.

For the implementation of Poly we have extended the `PackageInstallerActivity`. Figure 2 shows the screens as presented to the user during installation. Moreover, we also allow users to modify their constraints even after install-time. For this purpose, we have provided the same interface in the settings application of Android (`com.android.settings.ManageApplications` class) so that the constraints may be modified as the user's trust on an application increases or decreases over time.

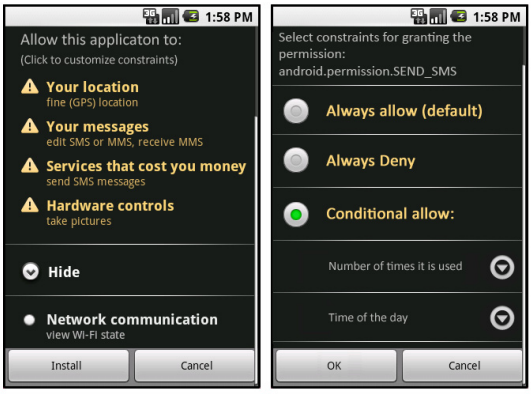

Figure 2: Poly Installation Interface: By clicking on a permission, the user can deny or impose constraints on that permission while still granting all others.

We believe that our comprehensive constrained policy mechanism coupled with the usable and flexible user interface of Poly provides a secure, yet user-friendly security mechanism for the Android platform.

5. CONCLUSION AND FUTURE WORK

In this paper, we have described Apex – an extension to the Android permission framework. Apex allows users to specify detailed runtime constraints to restrict the use of sensitive resources by applications. The framework achieves this with a minimal trade-off between security and performance. The user can specify her constraints through a simple interface of the extended Android installer called Poly. The extensions are incorporated in the Android framework with a minimal change in the codebase and the user interface of existing security architecture.

Our model is significantly different from related efforts [7, 1, 2] in that not only does it define an easy-to-use policy language, it is also *user-centric*. It allows users to make decisions about permissions on their device rather than automating the decisions based on the policies of *remote owners*. Secondly, it allows finer-granular control over usage through constructs such as attribute updates.

6. REFERENCES

[1] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *CCS '09: Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245, New York, NY, USA, 2009. ACM.

[2] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. In *Submitted to IEEE S&P'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[3] Google. Android Home Page, 2009. Available at: `http://www.android.com`.

[4] Google. Android Reference: Intent, 2009. Available at: `http://developer.android.com/reference/android/content/Intent.html`.

[5] Google. Android Reference: Manifest File - Permissions, 2009. Available at: `http://developer.android.com/guide/topics/manifest/manifest-intro.html\#perms`.

[6] Google. Android Reference: Security and Permissions, 2009. Available at: `http://developer.android.com/guide/topics/security/security.html`.

[7] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the Annual Computer Security Applications Conference*, 2009.