Design and Implementation of a Fine-grained Resource Usage Model for the Android Platform

Mohammad Nauman[1], Sohail Khan[2]
[1] Department of Computer Science, University of Peshawar, Pakistan
recluze@gmail.com
[2] School of Electrical Engineering and Computer Science, NUST, Pakistan
sohail.khan@seecs.edu.pk

Abstract: *Android is among the new breed of smartphone software stacks. It is powerful yet friendly enough to be widely adopted by both the end users and the developer community. This adoption has led to the creation of a large number of third-party applications that run on top of the software stack accessing device resources and data. Users installing third party applications are provided information about which resources an application might use but have no way of restricting access to these resources if they wish to use the application. All permissions have to be granted or the application fails to install. In this paper, we present a fine-grained usage control model for Android that allows users to specify exactly what resources an application should be allowed access to. These decisions might be based on runtime constraints such as time of day or location of the device or on application attributes such as the number of SMSs already sent by the application. We give details of our implementation and describe an extended installer that provides an easy-to-use interface to the users for setting their policies. Our architecture only requires a minimal change to the existing code base and is thus compatible with the existing security mechanism. As a result, it has a high potential for adoption by the Android community at large.*

Keywords: *Security, Mobile platforms, Android, Policy Framework, Constraints.*

Received: February 2, 2010; accepted: March 1, 2010.

1. Introduction

In the current scenario of mobile platforms, Android [1] 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 [2] 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 describe 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.

Contributions: Our contributions in this paper are as follows: (1) We describe the extensions to the existing security mechanism of Android for incorporating usage constraints; (2) we create a policy enforcement framework that incorporates usage policies while granting permissions to applications for accessing resources; and (3) we describe and implement an extended package installer that utilizes an easy-to-use and intuitive interface for allowing users to specify their constraints and modify them even after the installation of an application.

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* [16]. Applications are composed of one or more different components. There are four types of components namely *activities*, *services*, *broadcast receivers* and *content providers* [11]. 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. The Android runtime includes Apache Harmony [1] class libraries that provide the functionality of core libraries for Java language.

Android enforces a sandboxing mechanism by running each application in a separate process of the *Dalvik virtual machine* [3]. Different instances of the virtual machine communicate with each other through a specialized inter-process communication mechanism provided by the application framework layer. This allows for loose coupling of code written by different developers.

Each application in Android is assigned a unique user ID (UID) upon installation. An application may request a specific UID through `sharedUserId` attribute of an application's manifest. However, packages requesting the same UID have to be signed using the same signature and are then considered to belong to the same application. The UID is therefore associated uniquely with an application and can be used to refer to a specific application [14].

Different applications are executed in their own instance of the Dalvik VM. 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"* [12]. Intents encapsulate the action to be performed in an action string as well as any data that is associated with the action to be performed, and the category which describes the type of component that may handle the Intent. Moreover, an intent can also include extra information associated with the call.

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 [10]. 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. Using Ringlet as an example application, we will describe the limitations of Android security mechanism for restricting access by the different applications to the phone's resources based on user's policies. This brief problem statement is elaborated in the following section.

2.3. Problem Description

Android comes with a suite of built-in applications like dialer, browser, address book, etc. Developers can write their own application using the Android SDK. Each application requires permissions to perform sensitive tasks like sending messages, accessing the contacts database or using the camera. The permissions required by an application are expressed in its `AndroidManifest.xml` file – referred to as the manifest file – and the user agrees or disagrees to them at install-time. When installing new software, Android framework prompts the user to allow the specified permissions required by the application. This way, the user has a chance to choose whether to trust the application or not. Unless the user grants all the required permissions to the application, it cannot be installed. Once the permissions are granted and the application is installed the user can not change these permissions [5], except by uninstalling the application from the device.

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.

2.4. Challenges

There are several challenges that need to be addressed while resolving these issues:

1. The new framework has to be compatible with the current architecture so that the existing developer community can readily accept the changes;
2. A minimum of changes must be made to the existing code base and user interface;
3. The framework must be easy to configure for mobile phone users keeping in mind the limitations of display and input methods; and
4. Performance overhead must be small.

We address these challenges 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.

3. Android Permission Extension Framework

Based on the problems and challenges described in the previous sections, we have developed an *Android Permission Extension (Apex)* framework. Figure 1 describes the architecture in brief. The existing Android application framework does not define a single entry point for execution of applications [11] i.e., applications have no `main()` function. Applications are composed of components, which can be instantiated and executed on their own. This means that any application can make use of components belonging to other applications, provided those applications permit it.

The instantiation of these components is handled by different methods of the `ApplicationContext` class in the Android application framework layer. The `ApplicationContext` acts as an interface for handling Intents. Whenever an intent is raised, the `ApplicationContext` performs two checks: first, it checks whether there are permissions associated with the Intent; secondly, it checks whether the calling component has been granted the permission associated with the Intent.

The `ApplicationContext` implements the `IActivityManager` interface that uses the concept of *Binders* and *Parcels*, the specialized Inter Process Communication mechanism for Android. `Binder` is the base class for remotable objects that implements the `IBinder` interface. This interface provides the core part of a lightweight remote procedure call mechanism, which is designed specifically for improving performance of in-process and cross-process calls. `Parcel` acts as a generic buffer for inter-process messages and is passed through `IBinder`.

The `ApplicationContext` creates a parcel aimed at deciding whether the calling application has a specific permission. The `ActivityManagerNative` class receives this parcel and extracts the PID, UID and the permission associated with the call and sends these arguments to the `checkPermission()` method of the `ActivityManagerService` class. This method is *the only public entry point for permissions checking* (Source: Comments in Android source code for class: `com.android.server.am.ActivityManagerService`). These arguments are passed to `checkComponentPermission()`, which performs multiple checks: if the UID is a system or root UID, it always grants the requested permission. For all other UIDs, it calls the `PackageManagerService`, which extracts the package names for the passed UID and validates the received permissions against the `grantedPermission` hashset of the application. If the received permission does not match any of those contained in the hashset, the Android framework throws a security exception signifying a denial of the permission.

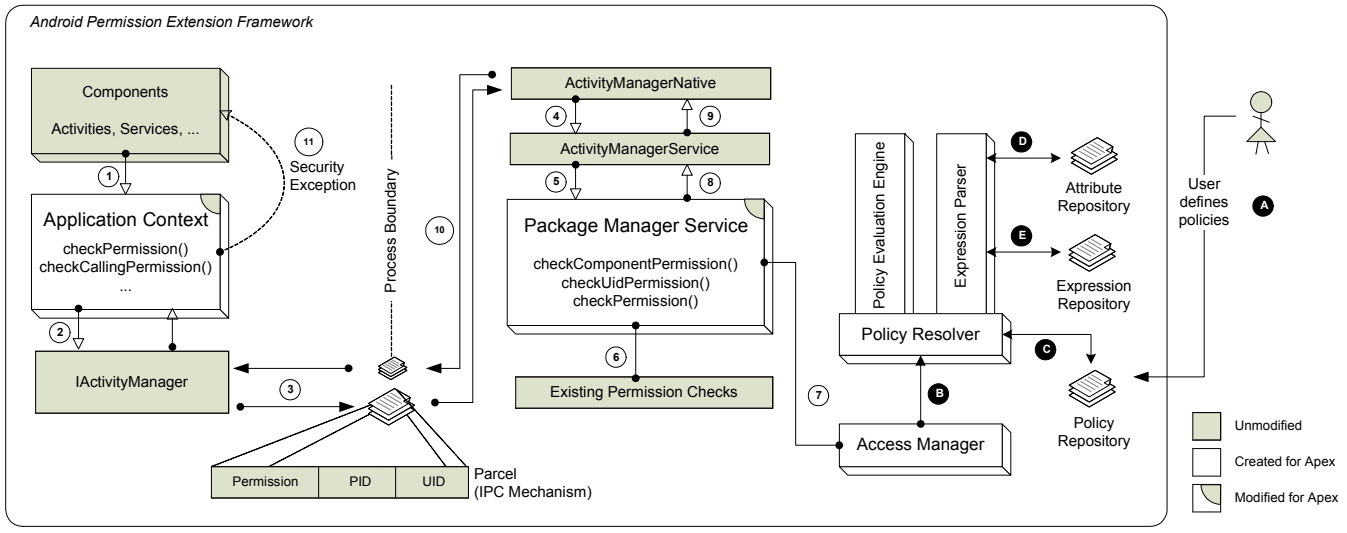

Figure 1 Apex Architecture

For incorporating runtime constraints for permissions, we have modified the `PackageManagerService` class. After checking the existing security permissions, control is passed to the `AccessManager`. For this purpose we have placed a hook in the `checkUidPermission()` of `PackageManagerService` that passes the UID and the requested permission to the `AccessManager`. `AccessManager` invokes the `PolicyResolver`, which retrieves the policy attached to the relevant application and evaluates it using the `PolicyEvaluationEngine`. The policies contain constraints for granting or denying the permission along with attribute update actions. Both of these are resolved using the `ExpressionParser`, which retrieves the attributes of the application from the attribute repository and also performs different operations on these attributes. Instead of hard-coding expressions, we have opted to define an interface for expressions, which can be used for extending the set of available expressions. This can be useful in future extensions of the framework by incorporating new expressions; for example those performing set operations. Currently, a set of commonly used expressions such as numerical comparison and datetime functions are included in the source. Below we describe the details of `AccessManager` and the relevant portion of our constrained permission evaluation mechanism.

3.1. Apex Policy Execution

The `AccessManager` class handles all permission checks related to the Apex framework. The user policies are represented in an XML file stored in the `SystemDir` of the Android filesystem. Figure 2 shows example high-level policies for the Ringlet application. The first three policies specify the constraint that the Ringlet application can only send five SMS/MMS messages each day. The first two of these save the number of messages sent in the `sentMms` attribute of the Ringlet application and the third policy resets the count when the permission is requested for the first time in a day. Note that the return value of the authorization rule in the third policy is `permit` i.e., the permission will be granted; whereas the second policy imposes a restriction by returning `deny` if the number of times used exceeds the allocated quota.

The fourth policy specifies the time of the day during which permission to access the GPS should be denied to protect the privacy of the user and the fifth one restrict the use of Internet outright. Note that these high-level policies are for illustrative purposes only and are not used in the implementation. Since the existing Android security mechanism stores permissions in an XML file, we have also opted for an XML representation of these policies. These policies are stored in `SystemDir` of the Android filesystem as XML files. For performance enhancement, each file stores policies related to one specific application. Each application in Android is associated with a specific UID and all permissions are associated with applications instead of different packages. Therefore, Apex policies associated with a single UID are stored in one file and are applicable on all packages in the application this UID represents.

Figure 3 shows the XML representation of the first policy shown in Figure 2. The root node is the `<Policies>` element, which includes `<Policy>` elements, each corresponding to different policies associated with the application. The `Effect` of the policy specifies whether to permit or deny the

```
mms_count_allow ("edu.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.ringlet.Ringlet" as Ringlet,
"android.permission.SEND_SMS" as MMS):
  Ringlet.sentMms > 5 /\ Ringlet.lastUsedDay =
System.CurrentDay -> deny(Ringlet, MMS);

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

Figure 2 High-level Apex Policies

```
<Policies TargetUid="10029">
 <Policy Effect="Permit">
  <Permission>android.permission.SEND_SMS</Permission>
  <Constraint CombiningAlgorithm="edu:android:apex:ALL">
   <Expression FunctionID="edu:android:apex:less-than-
equal">
    <ApplicationAttribute AttributeName="sentMms"
default="0">
    <Constant>5</Constant>
   </Expression>
   <Expression FunctionID="edu:android:apex:date-equal">
    <ApplicationAttribute AttributeName="lastUsedDay"
default="eval(day(System.CurrentDate)- 1)">
    <SystemAttribute AttributeName="CurrentDate">
   </Expression>
  </Constraint>
  <Updates>
   <Update TargetAttribute="sentMms">
    <Expression FunctionID="edu:android:apex:add">
     <ApplicationAttribute AttributeName="sentMms"
default="0">
     <Constant>1</Constant>
    </Expression>
   </Update>
  </Updates>
 </Policy>
 <Policy> ... </Policy>
</Policies>
```

Figure 3 XML Representation of Policies in Apex

permission if the constraints are satisfied. The permission targeted by the policy is specified in the `<Permission>` tag. Policies include the conditions for authorization (specified using the `<Constraint>` tag) and the updates that are to be performed (captured in the `<Updates>` tag). Each constraint consists of one or more `<Expression>`s. The results of the expressions are combined using the `CombiningAlgorithm` specified by the constraint. Expressions apply functions on their operands and can be recursively defined. Functions are specified using the `FunctionID` attribute and provide a pluggable architecture for further extensions. Operands can be of three types 1) Application attributes – specified using `<ApplicationAttribute>` tag that takes an attribute name and a default value to be returned if the attribute doesn't exist in the attribute repository, 2) System attributes, which include attributes not associated with a single application such as the location of the phone and time of the day and 3) constants.

A policy may also include several updates, each of which is specified by an `<Update>` tag. The result of the update expression is saved in the attribute specified by `TargetAttribute`. If the constraints in a policy are satisfied, the updates have to be performed regardless of the effect of the policy. If any of the satisfied policies has the effect 'deny', the end result of the permission check is to deny the requested permission. Otherwise the permission is granted. Note that in our framework, even if a satisfied policy has the effect of denying a permission, *all* subsequent matching policies are still evaluated. This is so that the updates specified by other satisfied policies may be performed.

The representation of Apex policies in XML is a design decision motivated by the fact that the manifest file, which hosts the existing permission constructs, is also represented in XML. Android source code includes a light-weight and efficient XML serializer – `FastXmlSerializer` and a parser based on `XmlPullParser`. Both of these are based on the XML processing interfaces defined by the XMLPULL API [17]. They are used by the `PackageManagerService` for processing and writing constructs of permissions and have been utilized in Apex for efficient XML processing.

The result of the policy evaluation is propagated to the application layer using the existing Android IPC mechanisms.

3.2. Result Propagation

The Android framework returns one of two possible values as a result of the permission check. These are the `PERMISSION_GRANTED` and `PERMISSION_DENIED` public fields of the `PackageManager` class. If the permission is granted, Apex returns `PERMISSION_GRANTED`. To differentiate between the denial of a permission based on static checks and that resulting from the constraint checks, we have included a new member field `PERMISSION_CONSTRAINT_CHECK_FAILED` in the `PackageManager` class. If the result of the policy evaluation is `deny`, the `AccessManager` and subsequently `PackageManagerService` returns this value to the requesting process. In the `ApplicationContext` class, the `enforce()` method is used to create an instance of a `SecurityException` with a custom message declaring that the constraint checks have failed. The exception is then thrown and eventually caught by the application that requested the permission. Figure 4 shows the error message displayed by the Ringlet application when it was denied permission to send an MMS message. We have opted not to change the `SecurityException` class for the sake of backward compatibility. Existing application code catches security exceptions and a change in this mechanism might break down existing code.

Note that the inclusion of a new public member field in the `PackageManager` class constitutes a change in the public API of the Android SDK. A change of this

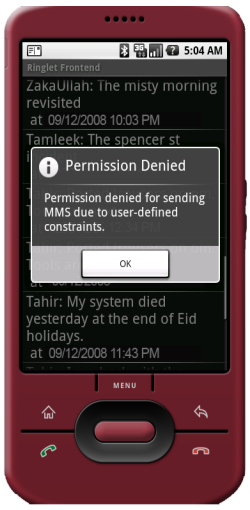

Figure 4 Permission denied as a result of constraint violation

nature cannot be incorporated in the publicly available Android source code without an approval through the Android source review process [15]. However, we believe that this is only a minor change in the API and is useful for the purpose of communicating the reason of permission denial to the requesting applications.

3.3. Performance Evaluation

The primary users of mobile phones in general and Android in particular are usually unable or unwilling to sacrifice performance for security. Moreover, the computational power of most smartphones, while being superior to traditional cell phones, is still lower than desktop computers. It is therefore necessary that the security policy model not overly tax the computational capabilities of the phone. Writing policies is a one-time operation, is currently performed at install-time and therefore does not cause any reduction in runtime performance. The evaluation of dynamic constraints and execution of update actions however, is a recurrent task and is performed for all applications for which a policy exists. Note that by saving policies related to each application in a single file, XML parsing can be completely avoided for those applications for which no policy file exists, thus significantly improving performance.

To measure the performance hit caused by execution of Apex policies for the Ringlet activity, we have carried out some preliminary experiments. Table 1 shows the time taken by the existing security mechanism as well as that by Apex to resolve certain permission checks. These tests have been carried out on the Android emulator on a desktop PC with CPU speed and network latency set to emulate a real phone device. The *increase* in the amount of time taken for policy evaluation is rather large but note that the raw values are still in an acceptable range. A permission check taking approximately 70ms is certainly tolerable. Also note the minimal change in the time taken for the permission evaluation for browser application, for which no policy has been defined. This minimal performance hit, coupled with the usability of Apex make our framework suitable for use in the consumer market. Below, we describe how this usability has been achieved using an extended Android installer.

Table 1: Performance Evaluation Results

Action	Application	Time taken for existing checks (ms)	Time taken with Apex (ms)
Sending SMS	Ringlet	34	103
Accessing GPS	Ringlet	17	94
Accessing Camera	Ringlet	25	47
Access Internet	Browser	27	29

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 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. She can allow the permission outright, deny the permission completely or specify constraints on the permission such as the number of times it can be used or the time of the day during which it should be allowed. This serves multiple purposes:

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`. In the existing Android framework this activity is responsible for handling all application installations. It presents the user with an interface that lists the permissions requested by the application and allows the user to accept all permissions or deny the installation of the application. We have modified this functionality to

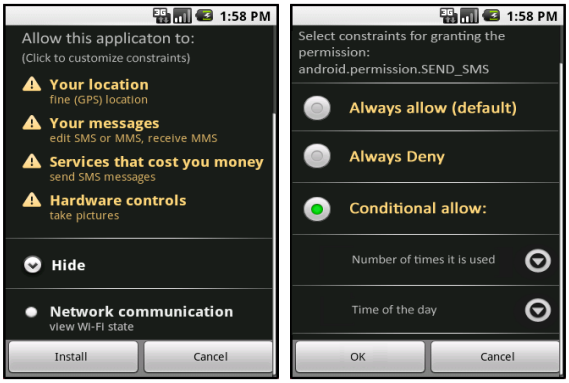

Figure 5 Poly Android Installer interface

enable the user to click on a specific permission and set the constraints for its usage. The constraints are organized in a user-friendly list of commonly used conditions. Once the user sets these constraints, Poly creates an XML representation of these constraints and store the policy in the system directory from where it can be read during policy resolution. Figure 5 shows the GUI presented to the user when she installs an application.

5. Constraint Modification at Runtime

One of the limitations of existing Android security mechanism is the inability to revoke permissions after an application has been installed. If a user wishes to revoke a permission, the only choice she has is to uninstall the application completely. Apex allows the user to specify her fine-grained constraints at install-time through Poly. However, once the user starts using the application and comes to trust the application, she may decide to grant more permissions to the application for improving her experience with the different features of the application. Consider, for example, that after using the Ringlet application (cf. Section 2.2) for a few weeks, the user comes to trust that the application will not misuse her location information and wishes to use the GPS feature of the application for including her location in the messages. At this time, she should be able to grant Ringlet the permission to access GPS. For modifying the runtime constraints on permissions, we have created a shortcut to the constraint specification activity of Poly (cf. Figure 5) in the settings application of Android (`com.android.settings.ManageApplications` class). This allows the user to modify the constraints she specified at install-time, even after the application has been installed. Using this interface, the user can grant the GPS permission to Ringlet application after she trusts that this information will not be misused. Similarly, the user can also deny access to a specific permission after install-time if she suspects that an application is misusing a resource.

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.

6. Related Work

To date, no efforts have been reported at addressing any of the problems described in Section 2.3 for the Android platform. Android source has recently been made available to the open source community and as such there is little scientific literature available on the security mechanisms of Android. Kirin [7] is an enhanced installer for Android that extracts the permissions required by the application from the manifest file for each application. These permissions are validated against the organizational policies to verify their compliance to the different stakeholder requirements. The stakeholder security requirements are represented as *policy invariants*. The installer eliminates the need for user's install-time decisions about granting the permissions to the application. It validates the permissions automatically against the policy invariants. If the application's permissions do not comply with these invariants, the application is not installed. However, there are two differences between Kirin and the approach presented in this paper: 1) The installer validates the permissions of an application only at *install-time*. There is no method to check runtime constraints. For example, a stakeholder policy that implements a limit on the usage of a particular resource cannot be enforced. 2) Associating permissions with components is not just restricted to the manifest file [14]. An application can also include a call to `Context.checkCallingPermission()`, `Context.checkPermission()` or `PackageManager.checkPermission()` to ensure that a calling application has the required permissions. Since Kirin only extracts permissions from the manifest file it cannot include this extra runtime information in its inference.

Similarly, [18] have described SAINT – a mechanism aimed at Android that allows *application developers* to define install-time and runtime constraints. However, note that this framework gives the option of policy specification to the application developers and not the user. Our work, on the other hand, is user-centric in that it allows the user to decide which resources should be accessible to which applications. We believe that, as the owner of the device, the decision to grant or deny access to device resources should remain with the user and not the application developers.

Another recent work related to applications security on Android, proposed by [8], is SCanDroid. It is a tool for automated reasoning about information flow and security verification of Android applications. It extracts information from the Android manifest file and the application source code to decide if the application may lead to unwanted information flows. While this is an exciting idea, it relies on the availability of the source code of the application in question – a rather impractical assumption in the current situation of the Android developer community. Moreover, the tool does not restrict access to any

resources as a result of its computation. It is merely for the sake of analysis. Finally, the common user cannot be expected to execute such a tool and a bridge has to be created between the average user and the tool for wide-spread adoption of this concept in the consumer market.

7. Conclusion

The massive increase in the consumer and developer community of the Android platform has given rise to important security concerns. One of the major concerns among these is the lack of a model that allows users to specify, at a fine-grained level, which of the phone's resources should be accessible to third-party applications. 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, 18, 8] in that not only does it define an easy-to-use policy language, it is also user-centric. It allows users to make decisions 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.

While we have successfully incorporated Apex in Android, a lot remains to be accomplished for fully exploiting the potential of the framework. For one, the installer currently incorporates a small number of constraints and a study of user requirements would help in deciding which constraint types are the most useful for a larger user community. Secondly, the problem still persists that users may unknowingly grant permissions that violate a larger security goal. A conjunction of Kirin [7] with our extended package installer may remedy this problem.

References

[1] Apache. Apache Harmony - Open Source Java Platform, 2009. Available at: http://harmony.apache.org/.

[2] D.E. Bell and L.J. LaPadula. *Secure Computer Systems: Mathematical Foundations*. 1973.

[3] Dan Bornstein. "Dalvik Virtual Machine", 2009. Available at: http://www.dalvikvm.com/.

[4] Burns J. "Exploratory Android Surgery". *Black Hat Technical Security Conference* USA, 2009. Available at: https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html.

[5] Burns J. "iSEC Partners: Developing Secure Mobile Applications For Android", 2009. Available at: http://www.isecpartners.com/files/iSEC_Securing_Android_Apps.pdf.

[6] Cheng J, Wong S., Yang H., and Lu S. "SmartSiren: Virus Detection and Alert for Smartphones". *MobiSys '07: Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, pp 258–271, New York, NY, USA, 2007.

[7] Enck W., Ongtang M., and McDaniel P. "Understanding Android Security". *IEEE Security & Privacy*, vol 7 issue 1. pp 50–57, 2009.

[8] Fuchs A., Chaudhuri A., and Foster J. "SCanDroid: Automated Security Certification of Android Applications". *Submitted to IEEE S&P'10: Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

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

[10] Google. Android Market, 2009. Available at: http://www.android.com/market.html.

[11] Google. Android Reference: Application Fundamentals-Components, 2009. Available at: http://developer.android.com/guide/topics/fundamentals.html.

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

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

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

[15] Google. "Android Submission Workflow", 2009. Available at: http://source.android.com/submit-patches/workflow.

[16] Google. "What is Android? – Android Developer Reference", 2009. Available at: http://developer.android.com/guide/basics/what-is-android.html.

[17] Haustein S. and Slominski A. "XML Pull Parsing". Available at: http://www.xmlpull.org/.

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

Mohammad Nauman is a researcher working in the field of collaborative systems and usage control. His research interests include remote attestation of distributed systems and security on smartphone platforms. He has a Masters in Software Engineering and is currently pursuing his PhD in Computer Information Systems. He also serves under the Expert Group for JSR321: Trusted Computing API for the Java Platform. He has published several articles in conferences of international repute and has presented his findings on several occasions to an international audience. He is also an author of two books – one on search in social networks and another on application of remote attestation in real-world scenarios.

Sohail Khan is an R&D scholar and has completed his Masters in Software Engineering from School of Electrical Engineering & Computer Science in the National University of Sciences & Technology Pakistan. His research interests include information security and identity management. He focuses on secure and trusted mobile platforms, applications, and services. He is currently working on creating a scalable and secure identity management framework as part of an e-Government system for deployment throughout Pakistan.