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

Smartphones, combining the mobility and personal nature of cell phones with the strengths of desktop computers, are fast becoming popular. 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

The increased capabilities and computational power of smartphones have enabled an increasing number of enterprises to deploy specialized services targeted towards these next-generation platforms. These services combine the mobility of traditional cell phones with the computational strengths of desktop systems to enable applications such as e-ticketing, e-prescribing, social networks, mobile healthcare and pervasive content creation and sharing systems. With the increase in use of open mobile architecture, security risks and attacks are also increasing on these devices [6, 26, 4]. Moreover, due to the mobility features and personal nature of mobile devices, security and privacy concerns are more threatening on these platforms than on desktop systems.

The enormous growth in the capabilities of mobile platforms has made it possible to deploy value-added services such as mobile payment, e-ticketing and social applications on these platforms. As a result, more and more applications and services have been deployed on these devices, which bring new business processes, pervasive information and content sharing, and mobile medical systems such as e-prescribing.

In the current scenario of mobile platforms, Android [9] 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 [10] 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 in the Android source code 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 formally define a subset of the existing security framework of Android, which suffices to define our proposed extensions rigorously; (2) we describe the extensions to the existing mechanism for incorporating usage constraints; (3) we create a policy enforcement framework that incorporates usage policies while granting permissions to applications for accessing resources; and (4) we describe and implement an extended package installer that utilizes an easy-to-use and intuitive interface for allowing users to specify their constraints.

Outline: The rest of the paper is organized as follows: In Section 2 we present the Android architecture and an application created to demonstrate Apex. This section also includes details of the target problem and challenges for designing a framework to address it. Section 3 formally presents the permission model of Android and the proposed extensions to this model. Implementation details of Apex are described in Section 4 and the extended package installer is presented in Section 5. A brief discussion on capabilities and limitations of Apex is presented in Section 6 and related work is provided in Section 7. Finally, Section 8 concludes the paper.

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 interprocess 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 *catogery* 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 (cf. Figure 1) 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

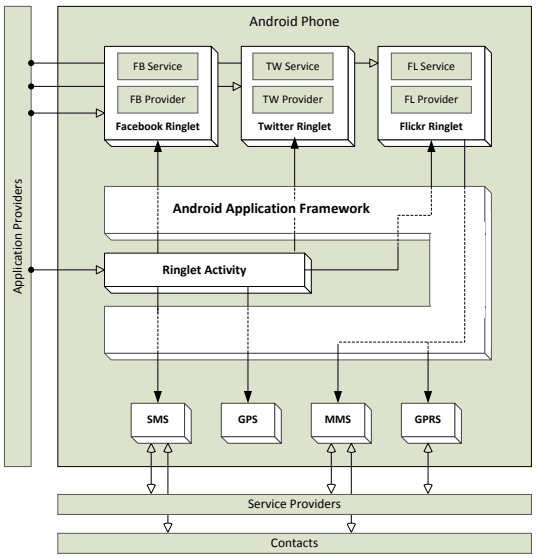

Figure 1: Ringlet Application Set: The Ringlet activity is shown as well as three other applications each responsible for handling data of and communicating with a single social network.

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 can not 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.

Assignment of a permission to an application results in providing unrestricted access to the respective resources. Android's existing security architecture does not provide a check on the usage of a resource. This, in our opinion is a significant limitation of the existing architecture. The user has no way of restricting the usage of a specific resource or denying access to one resource without losing the ability to use the rest of the application. The only way of stopping this sort of behavior is not to grant the required permissions to the application, which results in not being able to install the application at all. Moreover, in case of applications already installed on the phone, the user has no way of imposing such restrictions.

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. 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.

We begin by formally specifying the basic concepts used in the existing Android security architecture. We only cover a subset of the security architecture relevant to our discussion. This will be useful in specifying the semantics of our policy enforcement framework built on top of these basic constructs.

3.1 Applications, Components, Intents and Intent Filters

In traditional access control models [25, 27, 28, 23], policies revolve around the abstractions of subjects, objects and rights. In system-level permission models [22, 2, 20], 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.android.ringlet.fkringlet.POST`, data `"New Image Caption"`, category `"edu.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.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*.

In order to link intents of the applications and intent filters of components, Android application framework uses the concept of permissions.

3.2 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 [14]. 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:

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.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.

[1]"A feature can be protected by at most one permission". [13]

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. Permissions can either be granted – in which case they are always permitted – or denied – which results in failure to install the application. Below, we describe how we have enhanced this mechanism to include dynamic constraints on permissions.

3.3 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.

In the following section, we describe how these dynamic constraints can be incorporated in Android's existing security enforcement mechanism.

4. ANDROID PERMISSION EXTENSION FRAMEWORK

Based on the proposed policy model described in the previous section, we have developed an *Android Permission Extension (Apex)* framework. Figure 3 describes the architecture in brief. The existing Android application framework does not define a single entry point for execution of

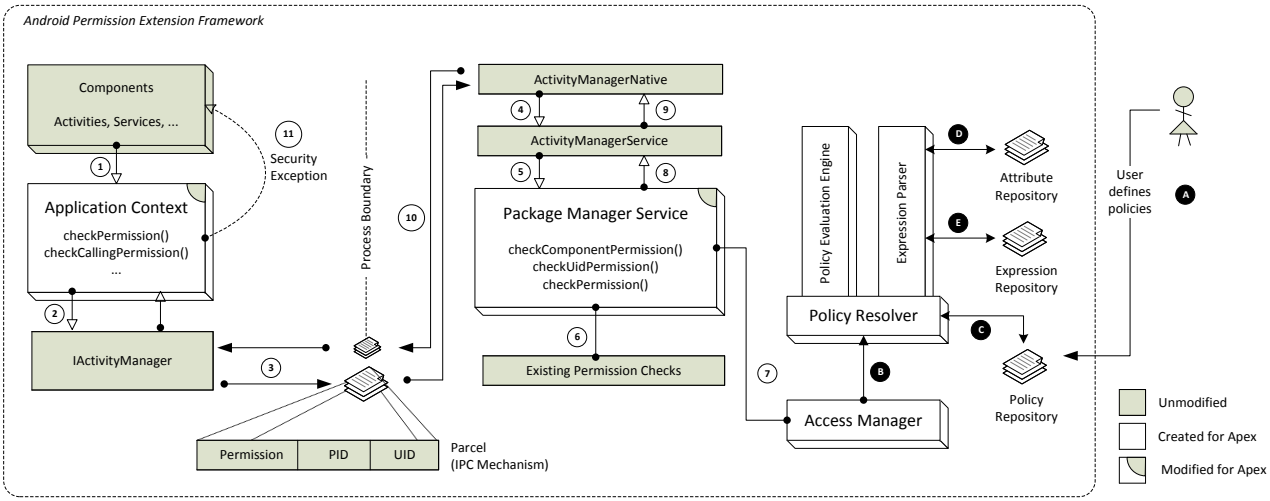

Figure 3: Android Permission Extension (Apex) Framework

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.

```
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;
```

```
deny_gps("edu.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 2: Example Apex Policies

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*[2]. 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.

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

[2]Source: Comments in Android source code for class: `com.android.server.am.ActivityManagerService`

```
<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 4: XML Representation of an Apex Policy

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.

4.1 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 requirements for the Ringlet application. The first three policies specify the constraint that the Ringlet application can only send five SMS/MMS[3] 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[4]. 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 4 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 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 result 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.

4.2 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

[3]Android associates the same permission with SMS and MMS messages.

[4]Appendix A shows a list of a few other permissions that a user might want to restrict

Figure 5: Usage Exceeded Exception in Ringlet: When Apex denies a permission, a security exception is thrown. This exception can be caught by the application, which can then signal to the user that access to a required resource has been denied.

`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 5 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 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.

4.3 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 (cf. Section 5) 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.

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

Table 1: Performance Evaluation Results

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.

5. 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 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 6 shows the GUI presented to the user when she installs an application.

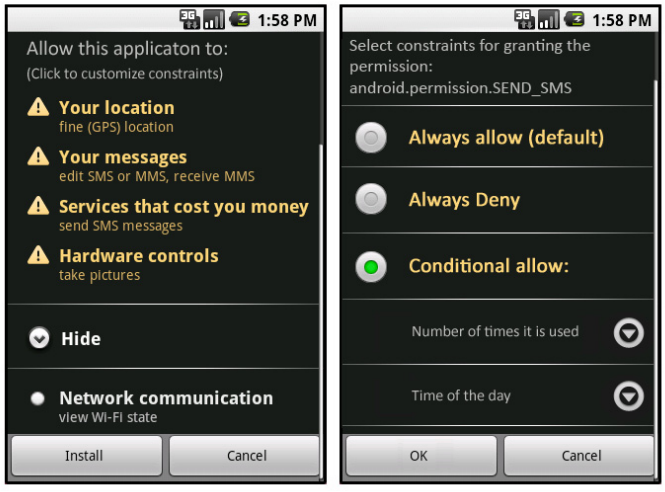

Figure 6: Poly Installation Interface: By clicking on a permission, the user can deny or impose constraints on that permission while still granting all others.

5.1 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 6) 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. DISCUSSION

Apex is capable of reasoning about permission checks based on application and system attributes. Apex hooks are placed in `checkUidPermission()` and `checkUriPermission()`. The first function covers all permissions associated with activities, services and broadcast receivers and the second covers all those associated with content providers[5]. As these are the only four types of components in Android, the permission mechanism of Apex is complete in that it intercepts all permission checks originating from all application components.

However, Apex relies on the existing Android architecture and as such there are certain limitations to what it can support. For one, since the existing permission mechanism only receives the PID, UID, and the permission, it is not possible for Apex to reason beyond these entities. For example, intents are not passed onto the permission check mechanism.

[5]In this paper, we have detailed the implementation of only the first hook. The semantics of the second are similar and are omitted from this paper for the sake of clarity.

Consequently, Apex cannot refer to the `data`, `category` or `extra` fields of the intent in its policies. Thus, the user cannot specify a policy that, say, restricts an activity from dialing a specific phone number. She can either allow the DIAL permission, deny it outright or impose constraints based on application or system attributes.

Although the ability to include this extra information during constraint evaluation may add significantly to the expressive power of the policy model, it would require a major change to the Android architecture. For now, we have opted to sacrifice this expressiveness for the sake of acceptability in the developer community.

Another point worth noting about the proposed architecture is that the existing Android applications expect to be granted all permissions that they request. Apex, however, makes it possible to grant a subset of these permissions. We envision that in the future, applications will be able to define different 'profiles' – each profile having a slightly different behavior – based on the permissions granted by the user. For example, the weather application presented in Section 1 might have two profiles: one that uses GPS and another that expects the user to input location information.

Finally, note that in cases where multiple policies may be applicable and some of them have update actions, ordering of the policies becomes significant. The reason is that the evaluation of one policy may change the state of the system and thus affect the results of subsequent evaluations. However, investigation of such scenarios requires detailed policy analysis which is outside the scope of this paper and remains part of our future work.

7. 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, [24] 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.

Android is based on the Java programming language and no fine-grained access policy mechanism exists for Android. However there are some previous efforts related to security policy models created for Java in particular and other embedded architectures in general. Martinelli and Mori [21] describe how they enhance the Java security framework for history based access control. The important aspect of this paper is that it focuses on the distributed nature of the environments in which Java applications execute. This distributed nature requires some sort of security policies to be applied to applications originating from arbitrary sources. Since the applications may potentially contain malicious code, allowing them to execute without some type of sandboxing may lead to serious threats.

However, since this approach uses Java security manager architecture that is not utilized by Android, it is not useful for addressing the concerns raised by this paper.

Security extensions for J2SE have been somewhat reduced in capacity in order to port them to the power-limited arenas of mobile platforms. The J2ME architecture does not include a security manager equivalent of J2SE. This poses serious issues for anyone interested in incorporating security policies in the J2ME architecture. Ion et al. [18] have defined a mechanism for extending the Java Virtual Machine to enforce fine-grained policies in J2ME. They extend the J2ME function call mechanism to check the calls against a user-defined policy before allowing or disallowing a specific operation. This extension is useful for J2ME based mobile devices. However, the implementation is the equivalent of the J2SE security manager and is not useful in Android, which has a completely different security model.

Evidence based access control model, proposed by Khantal et al. [19], describes a firewall, which collects evidences about different services and makes access decisions based on these evidences. Evidences can be exchanged or traded and may evolve over time. This leads to a dynamic access control mechanism with evidence providers and consumers in the

web services paradigms. The model is very efficient and is especially suited to smaller devices.

None of these approaches are targeted at Android and since the permission mechanism of Android is significantly different from those of other environments, there is a need for creating a policy model that is suitable for Android both in terms of expressive power and computational feasibility.

8. CONCLUSION AND FUTURE WORK

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.

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 by analyzing the constraints and permissions to verify that broader security goals are not being violated.

Moreover, we envision a change in the Android manifest file to allow application developers to describe why they request a specific permission and the extent to which they require it. It would allow developers to aid the user in answering questions such as "why should this application be granted GPS?" and "how many SMS messages should be allowed for this application?" to facilitate them in setting constraints for applications at install-time. This would require a change in the schema of the manifest file and has to be brought about through a backward-compatibility preserving, non-destructive change.

9. 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] Jesse Burns. Exploratory Android Surgery. In *Black Hat Technical Security Conference USA*, 2009. Available at: `https://www.blackhat.com/html/bh-usa-09/bh-usa-09-archives.html`.

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

[6] Jerry Cheng, Starsky H.Y. Wong, Hao Yang, and Songwu Lu. SmartSiren: Virus Detection and Alert for Smartphones. In *MobiSys '07: Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, pages 258–271, New York, NY, USA, 2007. ACM.

[7] W. Enck, M. Ongtang, and P. McDaniel. Understanding Android Security. *IEEE Security & Privacy*, 7(1):50–57, 2009.

[8] 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.

[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] Stefan Haustein and Aleksander Slominski. XML Pull Parsing. Available at: `http://www.xmlpull.org/`.

[18] I. Ion, B. Dragovic, and B. Crispo. Extending the Java Virtual Machine to Enforce Fine-Grained Security Policies in Mobile Devices. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 233–242, 2007.

[19] N. Khantal, J. Helander, B.G. Zorn, O. Almeida, and I. Center. Evidence-Based Access Control for Ubiquitous Web Services. In *W2SP 2008: Web 2.0 Security and Privacy 2008 Workshop at the IEEE Symposium on Security and Privacy*, 2008.

[20] B.W. Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[21] F. Martinelli and P. Mori. Enhancing Java Security with History Based Access Control. *Lecture Notes in Computer Science*, 4677:135, 2007.

[22] NSA. Security-Enhanced Linux (SELinux), 2009. Available at: `http://www.nsa.gov/selinux/`.

[23] OASIS. XACML 2.0 Specification Set, 2005. Available at: `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`.

[24] 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.

[25] Jaehong Park and Ravi Sandhu. Towards Usage Control Models: Beyond Traditional Access Control. In *SACMAT '02: Proceedings of the seventh ACM Symposium on Access Control Models and Technologies*, pages 57–64, New York, NY, USA, 2002. ACM Press.

[26] R. Racic, D. Ma, and H. Chen. Exploiting MMS Vulnerabilities to Stealthily Exhaust Mobile Phone's Battery. *IEEE SecureComm*, 2006.

[27] Ravi Sandhu. Rationale for the RBAC96 Family of Access Control Models. In *RBAC '95: Proceedings of the first ACM Workshop on Role-based access control*, page 9, New York, NY, USA, 1996. ACM Press.

[28] Ravi S. Sandhu. Lattice-Based Access Control Models. volume 26, pages 9–19, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

Appendix A: Potentially Risky Permissions

Several permissions on an Android phone are potentially problematic and may lead to threats if granted to third-party applications. In the following table, we list some of these permissions [14] along with the threat they may cause if assigned to untrusted or malicious applications.

Permission	Description	Threat
ACCESS_FINE_LOCATION	Allows an application to access fine (e.g., GPS) location	Privacy concern
BLUETOOTH_ADMIN	Allows applications to discover and pair bluetooth devices	Privacy concern / malware threat
BRICK	Required to be able to disable the device	Hardware damage
CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer	Financial loss
CHANGE_CONFIGURATION	Allows an application to modify the current configuration, such as locale	Minor annoyance / malware threat
CLEAR_APP_USER_DATA	Allows an application to clear user data	Data loss
DELETE_PACKAGES	Allows an application to delete packages	Minor annoyance / data loss
FACTORY_TEST	Run as a manufacturer test application, running as the root user	Malware threat
GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service	Privacy concern
INSTALL_PACKAGES	Allows an application to install packages	Malware threat
INTERNET	Allows applications to open network sockets	Malware threat / privacy concern / financial loss
MOUNT_FORMAT_FILESYSTEMS	Allows formatting file systems for removable storage	Data loss
READ_CALENDAR	Allows an application to read the user's calendar data	Privacy concern
READ_CONTACTS	Allows an application to read the user's contacts data	Privacy concern
READ_LOGS	Allows an application to read the low-level system log files	Malware threat
READ_SMS	Allows an application to read SMS messages	Privacy concern
RECORD_AUDIO	Allows an application to record audio	Privacy concern
SEND_SMS	Allows an application to send SMS messages	Financial loss
SIGNAL_PERSISTENT_PROCESSES	Allow an application to request that a signal be sent to all persistent processes	Malware threat
VIBRATE	Allows access to the vibrator	Minor annoyance / battery loss
WAKE_LOCK	Allows applications to keep processor from sleeping or screen from dimming	Battery loss
WRITE_APN_SETTINGS	Allows applications to write the APN settings	Privacy concern / malware threat
WRITE_CONTACTS	Allows an application to write the user's contacts data	Malware threat
WRITE_SECURE_SETTINGS	Allows an application to read or write the secure system settings. (Currently cannot be requested by user-installed applications)	Malware threat
WRITE_SETTINGS	Allows an application to read or write the system settings	Malware threat
WRITE_SMS	Allows an application to write SMS messages	Privacy concern / malware threat

Table 2: A partial list of potentially risky permissions: In the most severe cases, allowing all the requested permissions to an application may lead to irreversible data loss. Rating the application as malicious after losing data would be small consolation to the user especially considering that the malicious developer can release the same malicious application under a different name to cause further damage. Also note that this is a partial list of permissions and potential threats. They are further complicated by the fact that applications can define their own permissions to protect their own data – loss of which may lead to further privacy and financial concerns.