Saarland University

Faculty of Natural Sciences and Technology I

Department of Computer Science

Bachelor Thesis

A Developer-Centric Approach to A Dynamic Android Permission System

submitted by

Oliver Schranz

submitted
September 30, 2013

Advisor

Sven Bugiel

Reviewers

Prof. Dr. Michael Backes

Dr. Matteo Maffei

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ______________________________

(Datum/Date) (Unterschrift/Signature)

Abstract. In the last few years, Android did not only evolve into the most widespread Linux based operating system, but also became the world market leader in the mobile operating systems market. Encouraged by the open nature of the platform and its popularity among end users, many researchers focused their work on Android. One of the most popular topics is the Android security architecture which isolates applications and only allows access to resources from outside the sandbox through well defined channels. For this reason, Android provides a privilege based security measure, relying on permissions that denote the right for applications to access a remote resource. Those permissions are granted at install time and there is currently no way to modify or revoke them afterwards, so even developers that are willing to improve the end user's privacy by supporting dynamic permissions do not have the facilities to do so on stock Android. Some experimental changes in Android 4.3 indicate that Google prepares such a system, but at the time of this writing, these features are hidden and marked as not approved. There have been different approaches to replace the current system with permissions that are dynamically revocable by the end user. But in contrast to solutions employing inline reference monitoring and rewriting techniques or system-centric approaches that necessitate a modified Android version, we propose a more developer-centric solution that solely runs at the application layer and thus works on stock Android. We encourage developers to use a newly designed API provided by our solution to access security and privacy sensitive resources, such as contact data or call logs. While offering the end user tools to be able to revoke and grant permissions at runtime, we provide the developer facilities to dynamically react to permission revocations and to prevent his applications from showing undefined behaviour or crashes. To provide this alternative permission system, we introduce new dynamic permissions that are provided by a support library to include in third party applications and enfoced by an application that guards the access to critical resources. Altogether, our solution proposes a new implicit contract for the permission system, which ensures that the end user has full control over the permission access of third-party applications and guarantees the developer to use protected APIs in every possible way as long as the user has granted the permission to do so.

Contents

Chapter 1

Introduction

1.1 Motivation

In the past few years, Android has become the world market leader in the mobile operating systems market. Its open source nature as well as its success on the consumer market offered a great opportunity for researchers to work with a fully open platform and simultaneously produce results potentially relevant for a huge mass of consumers.

Since then, a large body of literature on Android security has been established, especially to improve the Android permission system. Its intention is to guard access to critical system resources, like hardware and privacy-critical data, by sandboxing single apps and imposing an obligation to the developer to explicitly request access to these APIs.

When the user decides to install an application, he is prompted to approve the granting of all permissions requested by this app, or to decline and cancel the installation process. In fact, end users have neither the opportuniry to selectively grant or revoke permissions, nor the possibility to dynamically revoke them at a later point in time.

Concluded from the number of papers that evolved in the last years dealing with this problem, there is a huge demand for better solutions in this area. There are patches that propose a system-centric solution [1][2], but as they require a modified Android version, they are only gradually adopted by developers of the Android system. In order to overcome this deficit, subsequent solutions working on the application layer [3][4][5] are proposed to fill this gap.

The focus of this thesis is to propose a new, more developer-centric approach to a more flexible and dynamic permission system, showing its improved robustness compared to existing solutions, and prove its feasibility by introducing a feasible proof of concept for everyone to try out on stock Android.

In contrast to prior approaches where the threat model considers the app developer to be malicious in general, this thesis takes a different point of view,

in which the developer would like to provide the users of his applications more facilities to strengthen their privacy. In this scenario, the user benefits from additional options to protect his privacy, while the developer profits from the improved trust of the user in the application. The current Android permission framework would not allow for such a developer-centric approach to dynamic permissions. With this step towards the end user, the developer proves his awareness of actual privacy problems that may arise while using mobile applications with extensive privileges.
This approach provides a possibility to put the end user in full control while still giving the developer enough facilities to react to dynamic user decisions. While a system-centric implementation is favorable in the long run, the approach in this thesis does not require a modified Android version or an official patch and is thus deployable on all current Android powered devices.

1.2 Outline

In Chapter 2, a common ground is established and some background relating to the Android system itself is stated. Chapter 3 discusses related work on establishing more dynamic permissions on the Android system and their influence on this approach. In Chapter 4, we take a closer look at the design principles of this approach, while Chapter 5 discusses the realization and implementation of our proof of concept system on stock Android. Chapter 6 evaluates the results of this work with respect to security and usability. In Chapter 7, potential future work is described, based on this one, and Chapter 8 concludes the proceedings of this work. Finally, the last section states the references for the citations and related work.

Chapter 2

Background

In order to be able to understand current research on the area of Android security, we need to take some preparatory work and have a look at the concepts and internals of Android that this approach builds on.

2.1 Android's Main Application Components

There are four main components that make up the core of most Android applications. Those components are briefly introduced in order to find on common ground for the further reading.

Activities

An *Activity* represents a single view on the device screen and carries the graphical user interface. It is the main entrance point to most applications and its creation, starting, stopping and so on, strictly follows the so-called *Activity lifecycle*, depicted in Figure 2.1. The general GUI of an application is a combination of *Activities* which all inherit from the `Activity` class. *Activities* can be started from the application itself, but can also be triggered by an `Intent` from another application.

Services

A *Service* carries functionality that is intended to be executed without the need of a graphical representation or user interface. Most *Services* do background work, hence they follow a different *lifecycle* within the Android system. To implement a new *Service*, one needs to extend the base class of the same name.

Broadcast Receivers

Broadcast Receivers are callbacks for the system to be called on the occuring of so-called *broadcasts* (system wide messages). The most popular examples are so-called *short messages*. If the device receives a new SMS, a special *broadcast* is sent to the whole system which is only received by *Broadcast Receivers* that explicitly subscribed to *broadcasts* of this type.

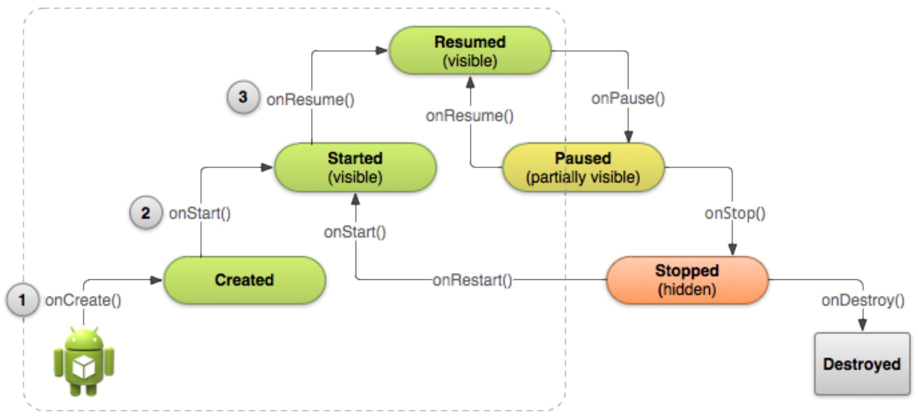


Figure 2.1: The Android Activity Lifecycle. [1]

Content providers

Those are Android's general pattern for managing and sharing almost arbitrary data. The idea is to offer a central access point to a set of data, a facade [6] to hide the internal structure of the data, but also to provide a simple and standardized method to offer insertion, updating, reading and deleting operations.

2.2 Android Inter Process Communication

Android's security architecture relies on isolation through sandboxing [7]. Hence, there is a need for clearly defined and observable channels to resources outside of the sandbox. As an application by default has no access rights to code or data of another application (user), it needs to rely on the inter process communication (IPC) mechanisms that Android provides.

For IPC, there is a demand for a secure, but also high performing mechanism, to quickly send secure messages and data between processes and to call methods remotely in other applications.

In Android, this work is done by the so-called Binder module [8].

There are restrictions on the type and size of data that can be sent via IPC. In Android, only classes that can be marshalled into the basic `Java` types (`int, float, byte, ...`), `Strings`, `Lists` and `Maps` of these types, or classes that again are marshallable into the mentioned types, can be sent via IPC. The reason is, if they can be marshalled into primitives, they can also be reconstructed (unmarshalled) back to the complex objects they were before. Taking on these restrictions mainly influenced the implementation strategy that we used in the end.

[1]This image is reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

2.3 Android Permissions

Dealing with Android permissions, one needs to know about how they are implemented and enforced, internally. As Android is based on an embedded Linux kernel [7], it inherits and uses many of its security measures to provide security on a preferable low level.

The first thing to know is that Linux was designed to be used in a multi-user environment, while still preserving security and privacy among all users. No user may be allowed to access the processes or memory of one of the others, so the system uses a security architecture to distinguish them by assigning every user a unique identifier and guard critical access by checking this ID. If there are resources that more than one user should have access to, those users are considered members of a Linux group, identified by a Linux group ID (GID).

Android builds its permission model on top of that isolation techniqe by assigning every installed application a unique identification number, the so-called user ID (UID).

The idea of permissions is to assign every resource a unique `String` identifier, the so-called *Permission Label.* A resource can be hardware like the camera, Bluetooth or WiFi module, or user data like contacts or phone numbers. In addition, access to an application interface can also be seen as a resource, but those correspond to user-defined permissions that we do not cover in this approach. The privilege, to be granted access to a resource, is called a permisson.
So, in order to use resources, applications need to declare that they plan on using the corresponding permissions, which involves adding *Permission Labels* to their manifest files. [9]

Internally, there are different types of resources. On the one hand, there are low level resources, like networking hardware, internal logs, file system access, and so on. As they can be accessed directly by application processes without the need to call another dedicated application that manages those resources, their permissions map to Linux group IDs [10]. So, in order to grant applications access to such a resource, their UID needs to be a member of the resource's Linux group.
On the other hand, most permission requests can be handled directly at the middleware layer. So for location based services, contact data, or access to third party application's interfaces which may be guarded by self-defined permissions, the communication to the application that provides the service or data takes place entirely on the application layer. For this reason, those permissions are only mappings from UIDs to *Permission Labels* in the `PackageManager` [11][12].
.

Another important property of the this security architecture is, that the described permission framework restricts all kinds of components in the same way, no matter what programming language was used. Native code, as well as `Java` code in the Dalvik [13] virtual machine, run in a single process. So from the system's perspective, the components appear as the same user with the same UID, resulting in the same assigned permission set.

2.4 A Sample API Call In Android

Because our work modifies the way, applications access resources located outside of the sandbox, we first need to have a look at Android's standard mechanisms for this kind of inter process communication. As we learned in Section 2.3, there are different types of resources that applications can request access to. But we also need to differentiate between the access from native code and from `Java`. As this approach does not cover native code, we take a look at a standard call to a permission protected resource, initiated by `Java` code.

Consider the *Contacts* and *Bluetooth* APIs. The first call that is placed in order to make use of the Bluetooth capabilities of an Android powered device, is always `BluetoothAdapter.getDefaultAdapter()`, to obtain a manager object to invoke further methods on. But let us assume, we already have the desired object and want to place a call that actually makes use of the Bluetooth functionality. So we invoke `startDiscovery()` to start searching for compatible Bluetooth devices nearby. For the contacts API, we use the `query` method on the responsible `ContentResolver`. As there is no need for us to handle `ContentResolvers` in detail, we omitt the exact code for simplicity. As denoted in Figure 2.2, the first part of the sequence is identical for both IPC calls. The method calls access functionality located in the *Application Framework*, the layer that provides the `Java` APIs for applications. The first difference is the permission check, triggered by the `PackageManager` introduced in Section 2.3. As the *Bluetooth* permission is based on a low level group id, a system call to the Linux kernel layer is is placed to check whether the calling application's UID is part of the bluetoot Linux group. In contrast, the *contacts* API is implemented as a service provided by another application, so the `PackageManager` only needs to check if there is a matching from the caller's UID to the *contacts* permission. We assume that the permission checks are passed in both cases. The call to start the discovery process now again triggers a call to a lower layer, to control the Bluetooth module built into the device. Again, to resolve the method call of the *contacts* API, there is no need to query deeper layers. The Application Framework now forwards the call to the contacts application that provides the service. The returned result is passed through the Application Framework layer back to the caller, using inter process communication.

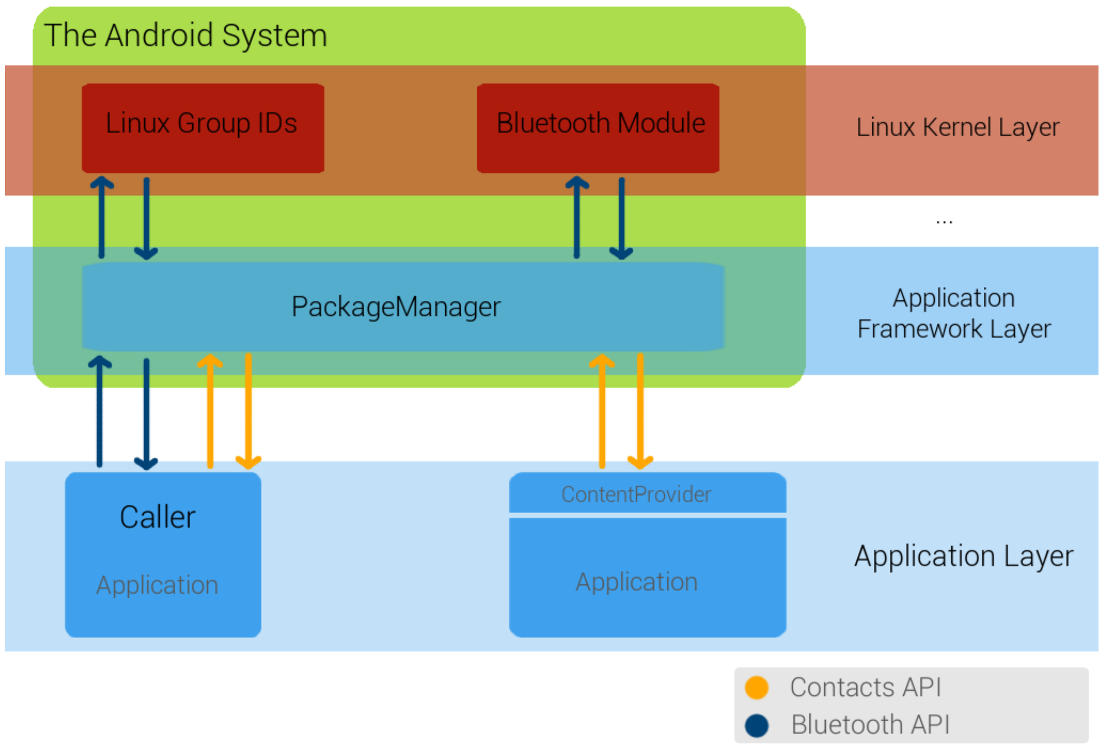


Figure 2.2: The sequence of API calls on stock Android to access the *Bluetooth Module* through the system itself and the *Contact Data* provided by an application

Chapter 3

Related Work

3.1 Inline Reference Monitoring Using Rewriting

The current state of the art of subsequent extensions to stock Android concerning permission-based security measures are inline reference monitoring [14] systems (IRMs).
The well known papers Dr. Android and Mr Hide [4], AppGuard [3], as well as I-arm-droid [5], introduce easy to use tools for end users to enforce dynamic permission revocation and granting at runtime.
But all subsequent solutions suffer from the same problems, arising from the usage of code rewriting:

- Revoked permissions can lead to undefined behaviour and random crashes, since the developer never created his app with the possibility of dynamically revoked permissions in mind.
- While acting on the application layer without any access to system resources, the monitoring adds overhead to every single call to an API guarded by a permission.
- An open problem for current IRM systems is the handling and monitoring of native code working with low level access to system resources [3][4][5].
- Since restricted to the application layer, the IRM applications themselves are an easier target for malicious third party applications than system-centric monitors. The problem is, that the inlined monitor runs within the application layer and has only the same privileges as the developer's code, while having no protection against native code or `Java` reflection.

Altogether, these approaches fight symptoms rather than causes. Their legitimation is their advantage to run out of the box on stock Android. There is no need for any modifications to the Android operating system. Every user can take action and install them directly to the device to improve his privacy, without burdening with a system update. So, basing IRM solutions on rewriting to modify third party applications is currently a necessity.

3.2 System-Centric Solutions

There have also been attempts to overcome the disadvantages of rewriting based solutions with system-centric implementations. While on the one hand, a modified Android system is a necessity, on the other hand, there are many opportunities resulting from implementing the solution as a part of the system itself.

- Being a part of the operating system, those solutions are far more robust and secure against attacks and exploitations by third party applications, because system software operates isolated from the application layer and with more privileges.
- Having access to system-reserved resources highly improves the performance.
- While running as privileged system code, there is no need to circuitous rewrite third party components. There is already a working mechanism for the permission check and enforcement that only needs to be modified, instead of being intercepted. A system-centric implementation is thus easier and more straightforward, because it does not need to worry about untrusted third party application code.

However, as subsequent extensions decrease the problems of the current system, at the end of the line, there is a clear demand for a system-centric implementation to solve the problem itself by getting down to the roots of the problem, instead of fighting symptoms.
There have been various attempts to implement such an approach system-centric.

The popular community-based Android fork CyanogenMod provided until recently the spoofing of data, but also complete denial of permissions, in order to provide a solution for end users to revoke permissions dynamically. After the feature was removed in version 8, some users requested to port this feature to actual versions [1]. The result was a discussion [1][15] between Steve Kondik, one of the core developers and founders of CyanogenMod, and community members. The reasons was, that CyanogenMod did not want to discourage developers to write applications compatible to their distribution [15], because this feature would introduce a very unstable environment for third party applications.

This statement appears exactly to similar approaches, like the hidden Jelly Bean 4.3 permission manager feature [2].
Although part of the official code base, this feature is hidden and currently not used in stock Android, but it gives a hint on what Google is planning to implement in the future. But the problem again is the same that Steve Kondik complained about and that the IRM solutions suffer from:
Simple revocation of application's permissions results in crashes and undefined behaviour. If there is no catch block that handles security exceptions by chance, the application cannot react in any way to this revocation and will most likely stop working properly.

AppFence[16] and TISSA [17] follow a different approach to increases the end users privacy by offering the opportunity to return empty or modified versions of privacy-critical data to third party applications. Furthermore, AppFence employs a framework to detect exfiltration of this data and provides means to block it. While these solutions do not modify the permission system to enhance the end users privacy like the approaches discussed before, they use the same thread model that considers the developer to be generally malicious.

The authors of Apex [18] went a step further and did not only modify the Android code, but stayed backwards compatible by keeping the standard security exception. Additionally, they provided a sample app to indicate how a developer could react to such an exception as a result of permission revokation. But this one step towards the developer by telling him that there might be a problem with the permissions, is still not enough.

At this point, this thesis tries to supplement the way towards the developer that became gradually visible in the previously mentioned papers. While there are only first indications of a harmonisation towards the developers, this work is completely designed to be not only developer-aware, but rather developer-centric.

Chapter 4

Design

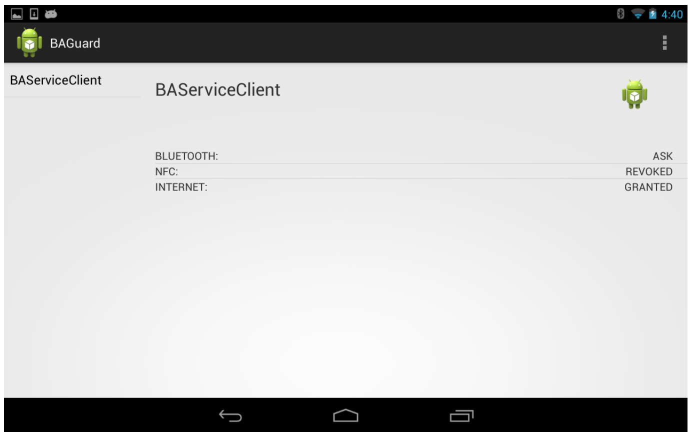
4.1 The Overall Design Principle

One of the most important underlying principles of thesis is the idea, that we do not want to assume all developers to be generally malicious. In fact, most security systems take the assumption of a harmful developer, in order to prepare for the worst possible scenario. Normally, this seems legitimate in this area, but we consider a different point of view. It is hard to believe that there are no developers that would appreciate the opportunity to give their users more possibilities to protect their privacy. There are whole companies that prosper while only working with open source and even providing many services for free, for non-commercial use for example. But as stated in Chapter 3, there is no way to provide the user dynamic permissions or a fine-grained control mechanism for them. This is the point where this approach comes in. No enforcement is done, no rewriting, no withdrawal of confidence or trust. The developers need to decide to use this system.

The system proposed in this work tries to give the developer exactly those facilities, needed to write applications in such a way that end users have more trust in them. We take a step towards a system where there is no need for the user to install third party applications to protect his privacy, because the applications are built in a way that enables and appreciates user decisions on privacy topics.

4.2 Architecture Components

Our system consists of two parts. On the one hand, the GuardApp that adopts the system's role to perform permission checks, and on the other hand, the GuardLib on the developer side that offers developers simple tools to write applications fitting into this new ecosystem.



BLUETOOTH: ASK
NFC: REVOKED
INTERNET: GRANTED

Figure 4.1: A screenshot of the GuardApp's details screen

4.2.1 The GuardApp

This part of the system carries two responsibilities.
The first and most important one is its function to provide and enforce a replacement for the current Android permissions, the so-called Guard permissions. Those newly created permissions are meant to be dynamically revocable by design and depict an opportunity for users and developers to use them instead of the original Android permissions. In contrast to the current permission framework, applications can request access to and make statements about their use of Guard permissions at runtime. In general, these special permissions closely correspond to the original ones that the Android system offers. The names and semantics are closely related to their Android complements to make them easier to understand for developers, as well as for users. There is no need to design them similar to the original one, but we decided to do so to encourage a faster acceptance by developers new to the system. For example, there are again permissions to guard access to the *Internet* and *Bluetooth* APIs. The enforcement of the Guard permissions happens by simply checking each API access request by an application, if the user has granted the corresponding permission. If the permission is not present, the GuardApp declines the request. In the remainder of this thesis, when referring to the Guard permissions we simply use the term *permission* and when referring to the stock Android permissions we will explicitly refer to them as *Android permissions.*

Second, in order to realize a dynamic access control for permissions, the GuardApp offers the end user the opportunity to decide on the granting or revocation of every single permission that an application requests access to. To achieve user interaction, the GuardApp provides an intuitive graphical user interface, a GUI, to provide easy tools for the end user to control the behaviour of applications by deciding on the granting state of their permissions. Figure 4.1 shows the GuardApp's *details screen* that displays the permissions of an application and offers tools to adjust the permission state.

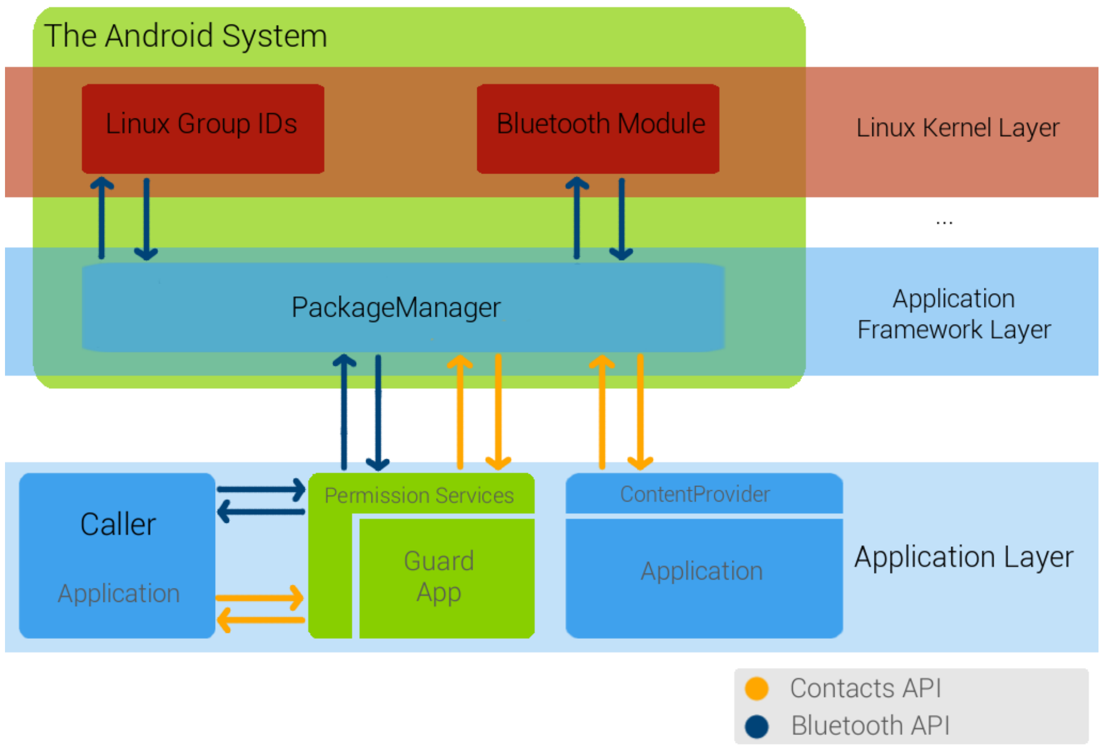


Figure 4.2: The *Contacts* and *Bluetooth* example from Section 2.4, extended to use the Guard System

Combining these two parts, the new system provides a way for developers to completely avoid and replace the default Android permission system implementation. However, as the GuardApp only replaces the original APIs, the developer needs to use the new ones in order to participate in our system. Figure 4.2 describes the usage of the Guard system with the example from Section 2.4.

4.2.2 The GuardLib

The library is meant to support developers who decided to use this dynamic permission system. It needs to be included in applications that plan on requesting access to Guard permissions. Its duties are to abstract from the internal communication with the GuardApp and to handle requests, as well as to offer tools, for a simpler use of the permissions and to support easy ways to react to permission revocation.

The internal communication mechanism between the GuardLib and the GuardApp is not meant to be used directly by the developer. But as all the permission requests and usage rely on this communication, we need to provide the developer a well-designed interface. For this reason, as depicted in Figure 4.3, we intro-

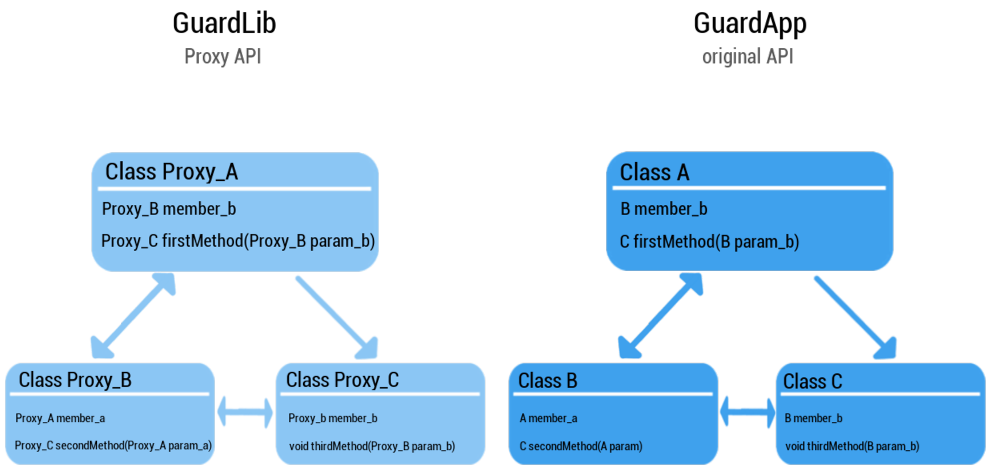


Figure 4.3: A proxy hierarchy that mimicks the structure of the original Android API

duce a proxy class hierarchy that almost behaves and looks like the one from the original Android API. But behind the scenes, the functionality hides in the GuardApp and the GuardLib is just a *remote control*. This fact is strongly hidden from the developer, who just needs to use the API classes that the library provides as they were the original ones.

A further benefit of the GuardLib is the chance for the developer to dynamically react to user decisions related to permission granting and revocation. As an application can lose the permission it currently needs for its computation, on the one hand, we need a mechanism to immediately stop the computation as a reaction to the user decision. However, it has to be prevented that applications show undefined behaviour or crash as a result of unexpected permission revocation. To handle this, we introduce a new exception that is thrown if the application does not have the permission related to the API, that it tries to use at that moment, just like the original security exception.
The difference is that in stock Android, you *can* catch this exception, but in most cases it does not give you anything because system permissions are given or absent for the full lifetime period of an application. There is no need to react dynamically to permission revocation, because this will never happen.
In contrast, in our design this behaviour is possible and even desired. So the developer can employ the standard `Java try-catch` mechanism to dynamically react to revocations as a result of user decisions. Furthermore, the developer is given a tool to check at every point in time whether a single permission is present at the moment.

Following is a selected example how a revocation could change the application's behaviour, which would not be possible without this dynamic approach:
An email application is synchronizing with a remote mail server. While this

work is still in progress, the user decides to revoke the *Internet* permission. The application switches focus to the catch block of the permission revocation exception executing its code: A `Toast`, a short on-screen text message in Android, tells about the fact that this app no longer uses the network capabilities and will remain in offline mode, as long as there is no possibility to reach the mail server. The consequence is, that the user can still read stored mails, write drafts, changes his settings, and so on. Using an inline reference monitoring system instead to revoke the *Internet* permission from a standard mail app, the application would most likely crash or show unexpected behaviour, which is definitly not the intention one has in mind while revoking the permission.

4.2.3 Prerequisites and Assumptions

There are a few assumptions and prerequisites on the usage on this system. First, an application using this library and thus using this dynamic permission system, should not have regular Android permissions. By using stock Android permissions, the application effectively bypasses all control mechanisms of the GuardApp. There is no need to use the API provided by the GuardLib, if the Android API is accessible without the risk of triggering a `SecurityException`.

Using this approach means abstaining from using the original permissions and solely using the Guard permissions. Developers consenting in using our system are assured to follow this design guideline. By omitting the Android permissions and choosing to use the Guard permissions, the developer, the user and the system itself take part in a newly designed contract. The system guarantees the developer, that while his app is given a specific permission, the application can use the guarded API in every possible way. The end user is guaranteed to have the power to stop every usage of a specific permission protected API for an application at all times. This contract allows for more trust and confidence towards applications and their developers, in contrast to the actual one.

A second requirement is the existence of an installed version of the GuardApp. This is an assumption towards the end user, who needs to understand that the system relies on this app to allow third party applications to use the Guard permissions and that the GuardApp needs to have all Android permissions in order to forward API access requests. Without it, no application designed with the GuardLib and programmed against those proxy APIs can work properly. Similar to the developer, the user also needs to actively decide to use this approach, to empower himself to decide on permission granting and revocation at runtime. In addition, the user also needs to be aware of the fact, that the GuardApplication needs to have all *all* Android permissions to work properly.

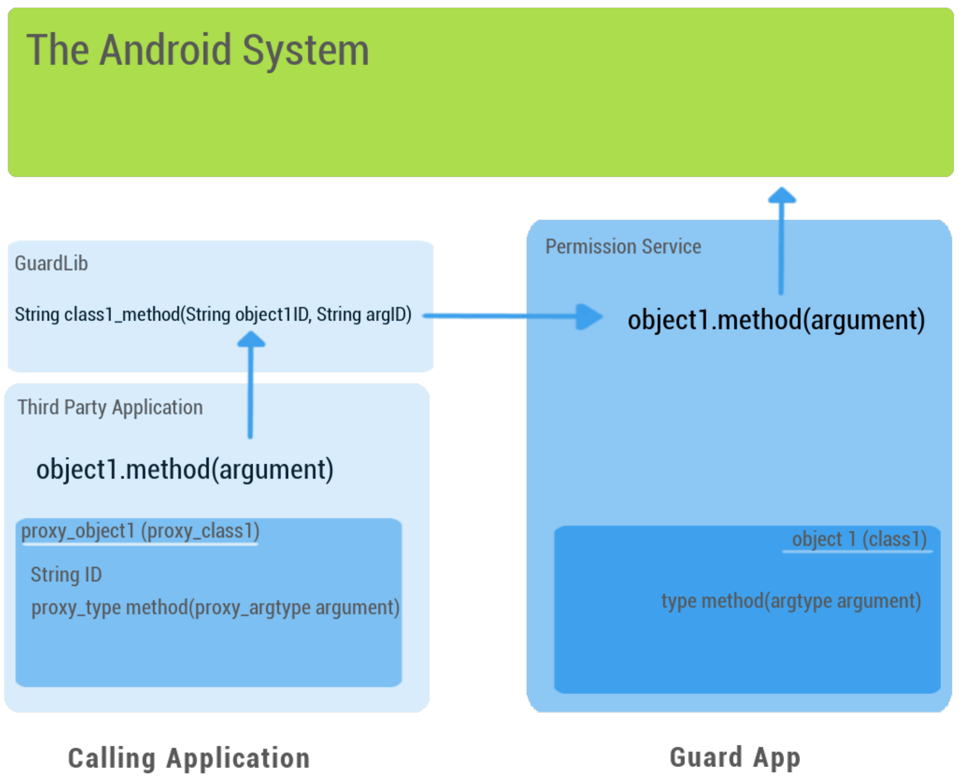


Figure 4.4: Sequence of an API call from the GuardLib API to the invokation on the original Android API

4.3 The Communication between the GuardLib and the GuardApp

The communication between the GuardLib and the GuardApp completely relies on IPC, thus we need to make sure to match its restrictions, listed in Section 2.2.

The connection itself is established using remote service binding [19]. The idea is, that the GuardApp provides public *Services* that the GuardLib can connect to. At this point, the library can send requests by simply using remote procedure calls (RPCs) with parameters that fit the IPC restrictions. These connections are used implicitly by the proxy objects that are returned to the developer by the library.

Figure 4.4 illustrates the invocation of a proxy object's method. On the GuardLib side, when `object.method(argument)` of the provided proxy API is called, the library calls a remote method `class1_method` and provides it with the IDs for the object to call the method on and for the `argument`. On the GuardApp side, the implementation of `class1_method` identifies the intended `object`, the method and the `argument` to invoke `object.method(argument)` on the original Android API. Finally, the result is returned back to the GuardLib side.

4.4 Maintaining Two Mirrored Hierarchies

In order to correctly identify the methods to be called on the GuardApp side, we need to establish a matching method between the class hierarchy of the original Android API and the mirroring proxy API. The similarity of the hierarchies is indicated by their shared structure and the almost identical class names. So for the developer, there seems to be nearly nothing different in using the GuardLib API instead of stock Android's, because the naming and signatures are very similar.

To make sure that every object that the developer creates on the GuardLib side and every method on these objects has exactly one counterpart, we need to create a bijection between the objects of the GuardLib side and those on the GuardApp side. The first thing to do, in order to ensure this property, is to stick closely to the original class hierarchy while creating the proxy hierarchy. This means that even between the classes, there is a bijection. While this is a necessary condition, it is not sufficient. The objects also need to build this projection in order to make sure the methods are called at the right place.
Imagine the following example which makes use of the Bluetooth API:

The first thing to do is to obtain the `BluetoothAdapter` by invoking `BluetoothAdapter.getDefaultAdapter()`. While working with *Bluetooth* connections, one gets an instance of the class `BluetoothDevice` [20] for every device (smartphone, tablet, laptop, ...) that is bundled with the current device through a call to `getBondedDevices` on the `BluetoothAdapter`. Each of these returned objects represents a remote device. If the system is not able to distinguish among them, it could be the case that private data like photos or messages are sent to the wrong recipient, resulting in a critical privacy leak.

So, to not confuse instances of the same class, we need to identify them with a unique identifier. Every class can follow its own advance to do so. In some cases, there already are identifiers to be used, for example hash functions. In other cases, they need to be assembled from the data or semantics an object holds.

Note that, as soon as we have our injective and surjective projection, we can uniquely identify which call was intended by the developer and translate it to the original Android API, which then returns the results back through the communication line between the GuardApp and the GuardLib, described in Section 4.3.

4.5 Designing the Guard Permissions

Our new permissions are designed to be more dynamic than the current ones. Basically, the Guard permissions are geared to the original permissions by simply guarding the same APIs and resources. Hence, there are again permissions for network usage, camera, Bluetooth, phone data, user data, location and so on. We could also have used more fine-grained permissions, like the ones proposed in AppGuard [3] and Dr. Android and Mr. Hide [4] for example, but because of time constraints, this is left for future work.

However, these new permissions differ from the default permissions in one point. There are four available states for a permission towards a single application: *Granted*, *revoked*, *timed* and *ask*. The fact that a permission is granted or revoked is straightforward and needs no further explanation. More interesting are the other two states.

Basically, a *timed* state denotes that a permission is granted for a specific timespan. It is based on the idea, that users may want to grant a permission just for a specific time period, for security or privacy reasons. A permission can be granted this way for an arbitrary combination of amounts of seconds, minutes, hours and also days. Right after the counter's expiration, the permission is treated as revoked as long as its state is not changed again.

Also mentionable is the *ask* option for permissions, which in its core obtains the decision on a single API usage from the user, every time a call to this API is made. The ask permissions show a single dialog that prompts the user to accept or decline the access request of an application, just like the permission dialog introduced in iOS 6 [21]. However, in contrast to the iOS feature that is restricted to only a few critical resources like contact data or location, every permission can have the *ask* status.
One example for a usage of the ask option:

An application administrates the short text messages (SMS) on the phone to back them up, store, archive and import them. In most of the cases, it would be an appreciated behaviour to load incoming messages into the application. But what if the user is using his mobile in the private sector and for his business, too. Then he may want to distinguish which SMS belong to his company and which ones are private. So, if the company dictates the usage of this app for backup reasons, we do not want our private SMS to be loaded and backed up on corporate servers. The solution is to set the *read SMS* permission to ask. This results in a dialog, showing up every time a broadcast for a new SMS is received. Now, the user can decide if the application can read the full message, the sender, and load it to the collection, or if it may not and thus just ignores it. This is an example where the user can expand new 'features' in third party apps where they are not provided by default or simply not even thought of by the developer.

Naturally, there are also cases where *ask* is infeasible. If an application downloads a large image, the user would be asked for his approval every time an array of bytes is read from the socket, which can be very annoying, and furthermore, the download is extremely impaired.

Chapter 5

Realization

5.1 Communication Through Remote Service Binding: AIDL

As seen before, the communication between the GuardLib and the GuardApp fully relies on remote service binding. The idea is that on the GuardLib side, there exists an object which represents a *Service* running in the process of the GuardApp. Every action that is taken on this object will be right forwarded to the original *Service*, a capability provided by the Android system through the Binder module. But to use this feature, we have to declare an interface that helps the two applications to agree on a common set of provided operations. The original *Service* is not accessible by the third party app, so it is necessary to share this special interface between the applications.

Android supports this type of communication through the *AIDL*, the *Android Interface Definition Language* [22]. The basic idea is to describe the methods of an interface in a *.aidl* file, which is shared between the communicating applications. The Android SDK automatically generates the interfaces from the file. This is possible because the definition of parameters in an .aidl file only allows types and classes that are unmarshallable and again marshallable. These types are called `Parcelables`. To make a complex class parcelable, it needs to implement the `Parcelable` interface to provide a way to unmarshal an object to its primitives, and to provide a way to rebuild the object again, using a static builder. Because all parameters are guaranteed to be parcellable or primitives, each side can work with the interface without the risk of not knowing a class that needs to be used.

So having the interface available, the binding procedure delivers an object that needs to be cast to the shared interface and can be used to remotely control the real *Service* implementation on the other side of the communication line.

The First Naïve Approach

At an early stage of the implementation, it became clear that the whole class hierarchy of a permission protected API needs to be rebuilt in order to use it with the system presented here. The most forward way to do that would be to implement the GuardApp's *Services* in such a way that the *Service* methods are simple wrappers for the original Android API methods. These wrapper methods would take the same arguments and nearly have the same method name as the original ones.

Listing 5.1 shows the structure of such a wrapper method of the *Service* to be called by the GuardLib side, created in order to make the Android API call `type method(argument)` available.

```
type method_(argument)
{
   return method(argument);
}
```

Listing 5.1: A wrapper for `method`

With this wrapper call, we can request the result of `method(argument)` without failing a permission check, because the call is made by the GuardApp, not by the third party application that uses the GuardLib. We can also return the result of the Android API call `method(argument)` back to the GuardLib side, because as this object was sent from the Android system to the GuardApp via IPC, it is a `Parcelable` and thus can be sent again via IPC to the GuardLib. But the GuardLib side cannot use this object, because each method of this object needs a *permission*. On invocation of such a method, the UID of the caller would be the one of the third party application, which is not a member of the Bluetooth Linux group, and the permission check would fail. So these methods also have to be invoked on the GuardApp side to prevent a `SecurityException`. In general, all objects that provide API methods need to stay on the GuardApp's side.

A Better Approach: Remote Control

After noticing that there is no way to send wrapped API class objects to the third party app in a way that they can be used without failing a security check, we abandoned the idea to send full complex objects. The new approach builds on the idea that the GuardApp stores all the objects. While the GuardApp has access rights to all API calls and manage the objects, the GuardLib side only needs to tell the GuardApp which methods to be called on which objects with which parameters. So, we introduce a communication protocol over IPC between the GuardApp and the third party application that uses the library. In order to abstract from this internal communication mechanism, the GuardLib offers proxy classes to the developer. They look like the original one from the Android API, but the implementation of their methods are just stubs that use our protocol to tell the GuardApp which method to call. So, without noticing, the developer only uses objects of proxy classes that forward his method invocations.

5.2 Implementation of the Hierarchies

We already talked about mirroring and rebuilding the original class hierarchy on the GuardLib side to provide proxy classes, but there are some problems that are not immediately so obvious.
One thing that we need to make sure is to establish the bijectivity. Every proxy class instance belongs to exactly one class instance of the original API and vice versa. The challenge here is to connect two hierarchies in a way that fits the strong constraints of Android IPC. The solution to this problem is rather simple: For the intermediate step, the communication, the hierarchy gets flattened down in the following way:
To make a method of the original API accessible, a flattened version of this method needs to be created for the interface shared between the GuardApp and the GuardLib. Let us assume we want to provide the method described in Listing 5.2 to the GuardLib. With the attribute *data-only*, we denote types that can be safely transmitted directly via IPC because the only carry data and have no methods that could trigger Android API calls. Those are primitive data types, `Strings` and objects of classes that only wrap data. We distinguish several cases of how the GuardLib and the GuardApp have to abstract the original Android API:

```
type method(argtype argument)
```

Listing 5.2: Typed sample method

1. **`type` and `argtype` are data-only**

   This is the easiest considerable case. The shared interface would look almost like the original one, with just an addition to the method name to not confuse them. Listing 5.3 describes the implementation on the GuardApp side.

   ```
   type method_(argtype argument)
   {
      return method(argument);
   }
   ```

   Listing 5.3: Only considering data-only types

   On the GuardLib side, one only needs to call `method_` on the casted binder object, which was provided during the service binding to the GuardApp's *Service*, to trigger the RPC to the implementation above. As the arguments are data-only, they can easily be transferred from the GuardLib to the GuardApp and the other way round. So the result is, that at the GuardApp side, `method_` is invoked with the argument passed through IPC, resulting in a call to the original method of the Android API with that parameter.

2. **`argtype` is data-only and `type` is complex**

   This is the type of method to get API objects from. A call returns a new instance of an Android API class. But as noted before, there is no way to send this object, that does not only store data, but have methods that trigger Android API calls, directly to the GuardLib side, because its usage would result in a security exception. So a unique ID needs to be created to identify that object among other instances of the same class. After the creation of this ID, the object is stored in a `Map` on the GuardApp side, with the ID as the key. What we return to the library is just the key, in our case a `String`.

The shared interface method, corresponding to the implementation in Listing 5.4, would be `String method_(argtype argument)`

```
String method_(argtype argument)
{
   type newobject = method(argument);
   String ID = type.computeID(newobject);
   typeStorageMap.put(ID, newobject);
   return ID;
}
```

Listing 5.4: Complex return type

This time, the GuardLib side also need some logic. As the only received return value is an ID , the library creates a new proxy object to mimick the instance of the class `type` that is stored on the GuardApp side. This proxy object stores the ID of its counterpart and sends this identifier to the other side, every time the instance is used in the third party application' code. The type of the proxy that mimicks an instance of the class `type` will be called `proxy_type`.

3. **`type` is data-only and `argtype` is complex**

   This time, the methods need complex objects as their arguments.

   The GuardLib is the entrance point, this time. In order for the developer to make a call to `method_`, he needs to pass a complex object as the argument. But what the developer passes as a 'complex' object is essentially just a proxy instance. Remember the previous case: If the developer wants to obtain an instance from the Android API, the GuardApp stores the original one and returns the ID. Then the GuardLib wraps this ID with a proxy and returns this proxy back to the developer's code. So the method signature that the library provides to the developer to hide its internals is in fact `type method_(proxy_argtype argument)`.

   Behind the scenes, the GuardLib extracts the ID from the proxy object and passes it as the argument to the remote method. Listing 5.5 corresponds to the shared interface method `type method_(String id)`.

```
type method_(String ID)
{
   argtype argument = argtypeStorageMap.get(ID);
   return method(argument);
}
```

Listing 5.5: Complex argument type

Here we see the opposite of the storage operation explained in the second case above: An instance that was stored on the GuardApp side before is now retrieved using its unique identifier.

4. **`type` and `argtype` are both complex**

   This case is a combination of the second and the third case. The `argtype` parameter is essentially a `String` ID which will be used to retrieve the original instance, which is meant to be the real argument. The newly obtained instance from an Android API class needs to be stored in a `Map` corresponding to its class, with its also newly created ID as the key. Altogether, the shared interface entry is `String method_(String id)`, with the corresponding implementation in Listing 5.6

```
String method_(String ID)
{
   argtype argument = argtypeStorageMap.get(ID);
   type newobject = method(argument);
   String newID = type.computeID(newobject);
   typeStorageMap.put(newID, newobject);
   return newID;
}
```

Listing 5.6: Complex types only

   To place this call, the GuardLib logic also needs to do some conversion. From the given `proxy_argtype`, the ID gets extracted and used as a parameter, and for the new ID returned from the GuardApp side, the GuardLib creates a new `proxy_type` instance to store that ID and return it to the developers code.

The Flattening Process

We discussed several types of methods, but this is not already the whole flattening process. There are not many cases in reality where there is one big class that contains all methods. In more realistic code examples, methods of object1 return instances of class2, which maybe return instances of class 3, and so on. This is the hierarchy that is implicitly build in the Android APIs and which we need to rebuild. To cover those, a little addition to our flattening concepts discussed before needs to be made:

The example signature is still `type method(argtype argument)`, but this time, the invocation on a specific object `object.method(argument)` is considered. The full implementation, including the additions to identify the object to call the method on, is depicted in Listing 5.7. Assume `object` is an instance of the class `basetype`. The whole solution breaks down to simply adding another argument to the method signature in the shared interface. This argument of type `String` helps the GuardApp to identify the object to invoke the method on. This results in an improved version of the method signature in the shared interface:
`type basetype_method_(String baseid, String argid)`

```
type basetype_method_(String baseID, String argID)
{
   basetype baseobject = basetypeStorageMap.get(baseID);
   argtype argument = argtypeStorageMap.get(argID);
   return baseobject.method(argument);
}
```

Listing 5.7: Invocation on complex object with complex parameter

The cases where the return type is not data-only do not need a seperate explanation, because we only need to add the already seen code to store the obtained instance and return the ID.
Of course, multiple parameters and wild mixtures of data-only and complex arguments are also allowed and work the same way.

The flattening itself is reflected in the name convention of the shared interface's method signatures. A methodname `class1_class2_class3_method_(id1, id2, id3, further_args)` indicates, that `method` should be invoked on the instance of `class3` specified by `id3`, which corresponds in a way to the instance of `class 2` identified by `id2`, which again belongs to the instance of `class1` indentified by `id1` in some way.

Appendix A shows an example *.aidl* file from the project code.

5.3 Realization of the Guard Permissions

In its core, the implementation of the Guard Permissions is nothing more than an `enum` that implements the `Parcelable` interface. The `enum` values are the identifiers for the different permissions, like *Bluetooth, Internet, NFC, ReadSMS* and so on. The reason to make it parcelable is to make the permissions also available for the GuardLib, even though they are originally part of the GuardApp. It gives us the opportunity to send them across the process borders via IPC, which is very useful in the process of implementing the developer tools.

What makes up a big part of the GuardApp's code is the realization of its duty to manage and check the permission state of third party applications. All decisions taken by the user are stored and accessible in the GuardApp. So, for every request by an arbitrary application, the permission state can be checked and the GuardApp can determine if the call is privileged. This check is embedded in every method that is provided to the GuardLib side for RPCs. If the request is legal, the code goes on without further checks, but if the requesting application does not have the permission to use the method, the check fails and a specific exception is thrown back across the process borders to the developer's code, where it can be caught.

In case of the easier permission status values, *granted* and *revoked*, the checking procedure only needs to read the stored data and accept or decline the request. But further computation is needed if the permission status is *timed* or *ask*. We will have a look at this.

The data corresponding to a UID-permission-pair is stored in a so-called *permission status bundle*. For *granted*, *revoked* and *ask*, objects of this class only stores the current permission state and the time of the last change. But *timed* requires a bit more data to be stored. Intuitively, the deadline, the time when the timed granting of the permission runs out, also needs to be available. This gives us the opportunity to check if a *timed* permission should be interpreted as granted or not, at every point in time. The only thing to do is to check if the current timestamp is before or after the stored one.

The last and also the most complicated permission status is the *ask* value. We recall its idea as asking the user for an accept or decline decision, every time the application code places a call to an API method that is protected by a Guard permission with the *ask* status. As usual, the Guard system checks every API call if it is privileged or not. If the permission status is ask, a graphical pop-up *Activity* is displayed.
While the execution of the code halts, the user can make his decision and from this point on, the system reacts as if the user decision was a stored permission status value which it now needs to interpret. The result is, as usual, an exception or a continued code execution.

Although the graphical pop-up is currently the only way to ask for case decisions, the code is written more flexibly in a way that also allows for more complex solutions. One example how such a solution could look like is listed in Chapter 7, Future Prospects.

5.4 Identification of Different Applications

In the previous section, it was already assumed that our solution is able to definitly distinguish between all third party applications that pose requests to the system. It is crucial to always know who requested which permission guarded API access, in order to decide if it is a legitimate query. If this would not be the case, the system would be in danger of breaching the contract concluded with the end user. Imagine a scenario where the end user revokes permission p from the application *app1*. Let us assume there is also *app2*, which has been granted the permission p before. Now, *app1* uses an API call protected by p. The focus moves on to the permission check of the Guard system. But what happens if the system confuses *app1* and *app2*, is that the permission check determines, that *app2* has the permission p and thus the call is legitimate. The result is that *app1* carries on with its API call, but in fact, the user did not allow this. To prevent such mistakes, there is a demand for a dependable tool to distinguish third party applications that use the Guard system.

To this end, we identify the caller right at the place where the request comes in. If we do not use information provided by third parties, we are less likely to be fooled by an attacker. So the solution is to use a mechanism that is built-in in the Android Binder module. In the Android `Java` code, the `Binder` class provides a static method called `Binder.getCallingUid()` [23], which provides the UID of the remote caller that initiated the RPC.
One peculiarity that a developer needs to know about this method is, that if the

call to the method where `Binder.getCallingUid()` is used was not triggered by an RPC but originated from a local call, the returned UID is the own identifier of the application itself. This is the case when we bind to a local *Service*. We do not care about this property because this call is only revoked in RPC triggered methods.

5.5 Data Management

The GuardApp's responsibility to manage the user decisions and to decide on legal or illegal use of protected APIs requires a storage and management solution for all the data.

5.5.1 Data Management at Runtime

Any decision that the user makes concerning an application-permission-pair is stored for frequent use in different `Maps`. It is cascaded in such a way that for any given Guard permission (key), there is an inner map (value), which assigns an application's UID (inner key) to the permission status chosen by the user (inner value). If the system is about to check the permission p for some UID and p is no element of the keyset, the check is considered to be failed, in order to minimize the amount of data to be stored. An implementation that considers non-existing triples as a granted permission would technically be no problem, but we decided to revoke permissions by default because we consider the end users privacy more important than the full-fledged app functionality. To save time and space, we only ensure few properties of the projections defined by the `Maps`:

Guard Permission → UID (Outer `Map`)

The `Map` only contains user decisions that were already made. So, a missing entry indicates that the user has not decided on that permission ever before, which justifies our reaction and makes the projection non-total. It is also not injective, because there may be a UID u and permissions *p1* and *p2*, so that $p1 \rightarrow u$ and $p2 \rightarrow u$ holds.

To talk about the surjectivity, there is no need to put a mapping from a permission to all UIDs into the `Map`, because there may be applications that do not use this permission at all. This is the reason why establishing surjectivity on the set of UIDs is not reasonable in our case.

Finally, the projection is functional. This is ensured because it is one of the fundamental principles underlying the concept of `Maps` that are used in programming. For every key, there should be at most one corresponding value, so there is exactly one value or none, which is the definition of the functionality property.

UID $\rightarrow$ Permission State (Inner `Map`)

The argument against totality here is the same as the one against the surjectivity in the outer projection. Some permission-UID pairs will never occur, so we do not need entries for every UID. Injectivity does also not hold, because there may be more than one permission-UID-pair that has the state *granted*, for example. The projection does not need to be surjective, because there may be cases where there is currently no permission set to the state *ask*, for example. Functionality again is given through the usage of a `Map`. These are exactly the semantics that fits to the projection, because for every permission-UID pair, there may only be one permission status or none. A permission that is granted and revoked at the same time does not make any sense.

5.5.2 Persistent Storage

Another responsibility of the GuardApp is the long-time storage of the data between several sessions. The data needs to be saved, no matter how many times the GuardApp or a third party application using it gets closed or the system shuts down. There should also be no need for the GuardApp to be active, while using the Guard system. The app part that is visible to the user (the user interface) is only responsible for providing the end user a control mechanism for his decisions, so if the user only wants to use a third party app relying on the Guard system, we do not want the GuardApp to always be actively running, too.

The solution is to store all the data in a serialized form. `Enums` as well as `Integers` can easily be serialized to be stored in the memory, so we can put all the `Map` cascades to the file system. Using the private storage of the GuardApp, Android makes sure that no other application can access or alter the data [7][24]. But that is still not enough. All GuardLib users are binding to *Remote Services* provided by the GuardApp, meaning there are potentially many apps, that try to access the data simultaneously. Furthermore, the GUI part of the GuardApp also loads and alters permission data while the user inspects and manages his application's permission states.

For this reason, we created the so-called `PermissionDataGateway` class as a central data access point. Every data query needs to go through this single point that encapsulates the data. To avoid data loss or corruption, the initialization mechanisms of the gateway are completely thread safe. The class itself is static, so no multiple instances can raise concurrency problems. The singleton pattern [6] would have also been an alternative. We did not use the Android standard data isolation pattern of *ContentProviders* introduced in Section 2.1, to prevent being forced to use a design inspired by relational databases. The usage of `Maps` reflects our intuition of permission-UID matchings and `Maps` are completely serializable, so we decided to use the serialization approach.

With these provisions, it does not matter in which order which applications try to access the data. There are no concurrent modifications possible.

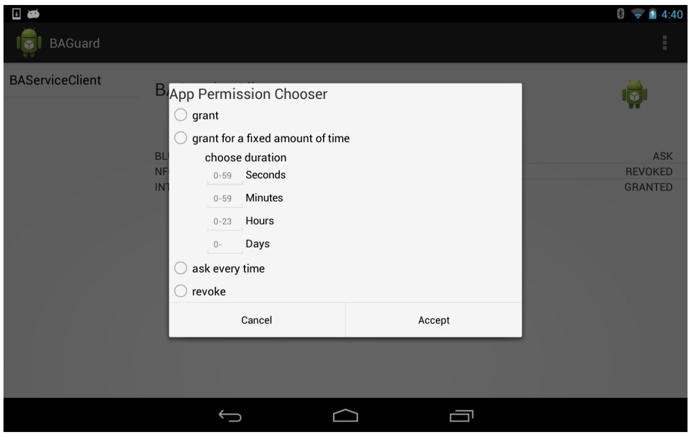


Figure 5.1: A screenshot of the permission chooser dialog

5.6 The Graphical User Interface

The part of the system, that end users perceives as the 'GuardApp', is the graphical user interface, that the system provides to allow for intuitive permission management.

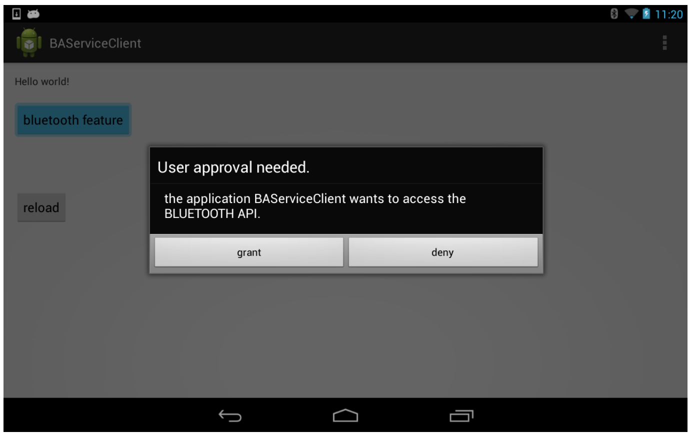
The landing screen shows a list of all applications that use our system to choose from. Right after choosing one, details like the full name, the icon, and the requested permissions, are displayed (cf figure 4.1).

By tapping on a permission, a pop-up screen shows up to change the permission status, if desired. It provides the possibility to choose *grant*, *revoke* and *ask* as single *radio buttons*, but also the option *timed* with the possibility to enter the amount of seconds, minutes, hours and days, that the timed permission should be seen as being granted. The full screen is shown in Figure 5.1.

The whole layout is `fragment` based, to be as flexible as possible. The concept of `fragments` was officially introduced in code level 11 (3.0, *Honeycomb*) and only supports devices that run a system with at least that version. But there is an official support library, based on *Activities* to provide the same functionality for applications that choose to support lower Android versions too.

One very important part of the GUI is the *ask dialog*, as depicted in Figure 5.2. This is the screen that shows up, every time an application makes a call to a permission protected API whose status is *ask*. At this point, the code execution becomes frozen and the user is prompted to accept or decline this single API usage.

The first attempt to implement this feature would be to place this code on the third party app side, as a part of the GuardLib. Because this app is currently running, it would be reasonable to start the dialog as a part of this app's user interface. But two problems arise with this approach:



User approval needed.

the application BAServiceClient wants to access the BLUETOOTH API.

Figure 5.2: A screenshot of the ask dialog

1. The permission check on which the system notices that the permission status is *ask* takes place on the GuardApp side. So, right in the middle of an RPC execution, how should the message, that the third party app needs to display the dialog and make the decision, being returned to the GuardLib, and how to continue with the call if the user chooses to accept. It would require a complicated messaging system across the process borders using IPC.

2. From a security perspective, it is never a good idea to place such code on the untrusted side of the communication line. By altering the GuardLib, an attacker could show a fake dialog or always tell the message, that the user has chosen to allow it, to the Guard side. To prevent such attacking scenarios, the code needs to be located on the trusted GuardApp side.

These two cases show the necessity to have the *ask dialog* as a part of the GuardApp's code base. But there is also one problem with this version. The RPC changes the execution focus to the GuardApp's *Service* implementation, but the displayed GUI is still the one of third party app. So we need a mechanism to start an *Activity* on the Guard side to display the dialog and return the GUI focus to the third party app where the request came from, right after the user decision.

The standard way on Android is to create an `Intent`, a small IPC message which triggers special behaviour in the system or another application, with a flag to create a new *Activity*, and invoke `startActivity` on the *Service* object itself. This is possible because the `startActivity` method is declared in the `ContextWrapper` class, a helper class for `Context`, which passes on this functionality to its derived classes like `Activity`, `Service` and so on.

This approach ensures, that the dialog is displayed as a part of the GuardApp's GUI, but also that the focus is returned to the *Service* code execution afterwards, because the `Activity` hosting the dialog closes autonomously.

5.7 Remote Exception Handling

At the time of this writing, stock Android does not fully support exception handling across the process borders, so exceptions that are raised within the execution of an RPC yield problems[25].
On the emergence of an uncaught exception in the remote process, the Binder module generally raises a `RemoteException` to be thrown back to the initiator of the RPC. The consequence is, that whatever type of exception is thrown in the *Service*'s code, the only thing that the other end learns about the failure is, that something went wrong. This fact makes it hard to propagate back exceptions, that are regularily thrown by API methods, like IO related functionality for example. The `RemoteException(String)` constructor seems to be a solution if we catch the original exceptions and throw the new remote exception by hand. The `String` argument could potentially carry a message back to the origin of the call, but unfortunately, this constructor was not introduced before API level 15 (*Ice Cream Sandwich Major Release 1*, 4.0.3) [26], which means we cannot use it if we plan on continuing support for older releases.

Fortunately, there are exceptions to the rule [25]. The `SecurityException` is one of the few exceptions that are not generated by the Binder module but propagated back correctly to the calling process if raised in the *Remote Service*. We make use of this insight by throwing a `SecurityException` only if an Android API access request is unprivileged, or in other words, if the required permission is currently revoked. This approach offers the opportunity to distinguish, on the GuardLib side, between all kinds of general exceptions that are reduced to `RemoteExceptions`, and `SecurityExceptions` which indicate revoked permissions.

Every time a `RemoteException` is raised as a consequence of an RPC call, the debug log prints ”Remote exceptions across processes are currently not supported.”, which indicates that this feature might be supported in future releases. But until then, we have to resort to alternative approaches or simply ignore *Honeycomb*, *Gingerbread* and lower releases and wait for the remnants to update, so we can use the constructor with the `String` argument.

5.8 Developer Tools

Our system is designed in a way that allows for and even appreciates dynamic permission revocation and granting while still maintaining a developer-centric approach. So there is a demand for easy and stable tools and solutions to provide enough facilities for the developer to write reasonable applications for this new ecosystem.

5.8.1 An Exception for Revoked Permissions

The most important facility to support dynamic reaction to revocation is to provide a mechanism to automatically stop the code execution and return an error message back to the application code. This is exactly what exceptions are for in

`Java`. The permission check itself is located on the GuardApp side of the communication channel. If failed, the check procedure throws a `SecurityException` which, as we explained in the previous section, can be propagated across the process boundaries back to the originator of the call. But in order to accomodate the developer, we also introduce a new unchecked exeption on the GuardLib side, the `PermissionRevokedException`. Potentially, every call to a method that triggers a permission guarded Android API call can result in a `PermissionRevokedException`.

5.8.2 The Permission Libraries

Section 4.3 explained that the GuardApp offers several *Services* that the GuardLib binds to. In fact, there is one *Service* for each permission that the system offers. For this reason, those are called *permission services.* To abstract from the communication protocol, the GuardLib provides interfaces to use these connections implicitly with each API call. Those interfaces called *permission libraries* do not only hide the IPC details, but they also convert the different exception types and provide means to wrap IDs into proxy objects and vice versa. Again, this is a service towards the developer. To use an API, the developer initiates the corresponding *permission library* offered by the GuardLib. From that point, the *permission library* manages the *Service Connection* to the corresponding *Permission Service* of the GuardApp, converts exceptions and proxies to IDs, and forwards the developer's API call by initiating remote procedure calls. For the developer, the usage of the library is nearly the same as if he were using the original Android API. While using Android APIs, in some cases there is a system service or static class that provides the first instance of an API class, which is often a manager object carrying further API methods. In the *Bluetooth* API, for example, the very first thing to do is to obtain the default instance of the `BluetoothAdapter` by invoking
`BluetoothAdapter.getDefaultAdapter()`. In our system, the *permission libraries* get initiated during the `onStart` lifecycle callback. So, for third party applications using our solution, the functionality of the `BluetoothAdapter` is governed by the `BluetoothLibrary` which offers the same methods with almost the same signatures.

5.8.3 The Control Service

But there are more accomodations towards the developer. Beside the *Permission Services* that the GuardApp offers, there is also a *Control Service* for permission independent services. On the GuardLib side, the communication through this interface is encapsulated in the `PermissionLibrary` class. The implemented method signatures are depicted in Listing 5.8.

```
boolean permissionGranted(GuardPermission p)
throws PermissionStatusAskException;

void announcePermissions(List<GuardPermission> permissions);
```

Listing 5.8: The fraction of the `PermissionLibrary`'s signature that encapsulates the calls to the *Control Service*

The first method, `permissionGranted`, allows the opportunity to place a request to ask for the granting state of a specific permission. If the return value is `true`, the permission state is *granted* or *timed* while the time limit is not reached yet. `false` is returned if the state is *revoked* or *timed* with an exceeded deadline. In addition, a `PermissionStatusAskException` is thrown if the status is *ask*, because the system cannot make a reliable statement in this case. The developer can use this as an approximation, because it may be the case that right after the access and reading of the current permission state, the user changes it. So it is not perfectly reliable, but it is also very improbable that this happens right after the method call was initiated.

The *Control Service* also provides another feature that offers the developer the opportunity to increase the usability of his applications drastically.

Recalling the storage mechanisms within the GuardApp described in Subsection 5.5.1, there are not all possible UID-permission combinations stored from the very beginning on. The permissions that the application uses do not appear in the data stock, until the user reaches a part of the application that requests this permission. In consequence, the user needs to run an application, and every time a part of the app that uses a not already known permission, the permission check fails. In consequence, the user needs to switch to the GuardApp's user interface to make his decision and then, on the second run, the feature is ready to work. Users would need to repeat this step for each new permission that is used at runtime. Because this behaviour does not allow for seamless usage of new applications, we introduce a feature to overcome this handicap. With the `announcePermissions` method, the `PermissionLibrary` class, which encapsulates the *Control Service Connection*, provides the possibility to tell the system about all permissions that the app as a whole is able to use. Again, there is no necessity for the developer, he can freely decide if he wants to use this offer to improve the user experience for his application's end users. The only thing, that needs to be done in order to use this feature, is to fill out a list with all the `enum` values of permissions that might be used and use this list as a parameter for the method. Internally, the library uses the *Control Service* to send this set of permissions to the GuardApp side, where it is stored and managed next to the other permission related data in the `PermissionDataGateway`.

The result is, that after the first start of the application, all permissions that the application can use (if the developer decided to tell all) are available in the GUI for the user to make a decision on the status to pick. So after this single interruption, the user will be able to use the application without further inconvenience. This mechanism replaces the static permission acquiring process in stock Android, which parses the manifest file [9] in order to read an application's permission requests.

5.9 Extending the Main Component's Lifecycles

In order to connect to the GuardApp and make its API available for the developer as soon as possible, the Guard system extends the main component's lifecycles and introduces the *permission libraries* to abstract from the initialization process. The ways the *permission libraries* are obtained and used are more standardized than the process of requesting the several 'manager objects' in the original Android APIs. The idea is to unify the procedure of obtaining those and thus to make the whole procedure more simple and consistent. In order to achieve this, we introduce a new mechanism to take care of all the *permission library* initializations, usage and tear downs.

This section will describe the extensions to the lifecycle of *Activities*, while only depicting differences to the described modifications for *Services* and *Content Providers*. *Broadcast Receivers* are only covered implicitly because they are only considered *valid* by the system as long as `onReceive` is executed, meaning the system can decide to kill the *Broadcast Receiver*'s process while waiting for an asynchronous callback [27]. To avoid this disadvantage, one of the other three components has to make the libraries available for the *Broadcast Receivers* and take care of their initialization and tear down.

5.9.1 onCreate

Before any library instance can be obtained, the overall `PermissionLibrary` needs to be initiated via `PermissionLibrary.init`. The `PermissionLibrary` class has three distinct duties.

1. It is the only class that the developer can use to create the actual *permission libraries*, introduced in Subsection 5.8.2.
2. It provides the method signatures to access the features in the *Control Service*, described in Subsection 5.8.3.
3. As all actual *permission libraries* inherit from this class, it sets up some common functionalities that all subclasses use, like binding to a *Service* and managing the corresponding connection.

The initialization ensures the connection to the *Control Service* to be established.
The next step is to fill up a list with constants corresponding to permissions for which a library should be prepared. This list is given as an argument to the `PermissionLibrary.requestPermissionLibraries` method, which triggers the creation and preparation of the libraries.
The call serves as a factory method [6] and returns a `Map` with the requested library objects stored as values for their corresponding permissions as keys.
To obtain the correctly typed library, one only needs to use the `Map` with the proper permission as the key and cast the result to the desired subclass of `PermissionLibrary`.
It is encouraged to store these instances centrally to be accessible by every part of the application.

5.9.2 onStart and onStop

In the `onStart` callback, we initialize the libraries that we obtained in the previous step. Of course, the *Activity* should only initialize the ones that it really uses. In a scenario where there are multiple *Activities*, there may be several such libraries, but not every one needs to be initialized by every component.

The initialization process is started by a call to the the method `reconnect()` on the *permission library object.* It is important to not use the libraries before they are initialized, because without an established connection to the GuardApp's *Remote Service*, no single method works.

Content Providers do not have an `onStart` callback, so they move the initialization to `onCreate` right after obtaining the instances. While *Services* do also not have an `onStart` method, explicitly started *Services* have the `onStartCommand` method to place the initialization in and bound *Services* move the initialization code to `onCreate`.

If we connect to *Remote Services*, we also need to close the connection eventually. This happens in the `onStop` callback, which is invoked every time the *Activity* loses focus or gets closed. *Services* use the `onDestroy` method. As *Content Providers* are created with their hosting process and are never destroyed while the process is still running, there is no need to care about unbinding a *Service.* If the system closes the host process, all the remaining resources of a *Content Provider* are cleared automatically [28]. To start the tearing down of a library, one calls `disconnect()` on the specific `PermissionLibrary` instances and the static `tearDown()` on the `PermissionLibrary` class.

This is the unified way to handle all the *permission libraries* in the application's code. All other usage of permission protected APIs nearly behaves the same as the original one from the Android system.

5.9.3 The New allLibrariesReady Callback

We already stated that there is no point in using uninitialized libraries, and because a *Service Connection* is built on a callback technique, we again propagate this information back and make a callback to the application as soon as *all* libraries are ready. For this reason, the *Activity* needs to implement an interface to have the `allLibrariesReady` callback. Figure 5.3 describes how this method, which will be called by the GuardLib if all *permission libraries* are ready and safe to use, fits into the *Application's* lifecycle. For the developer, this simply means that he should move the main code of the *Activity* from `onStart` to `allLibrariesReady`, besides implementing the interface.

[2]This image is based on work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

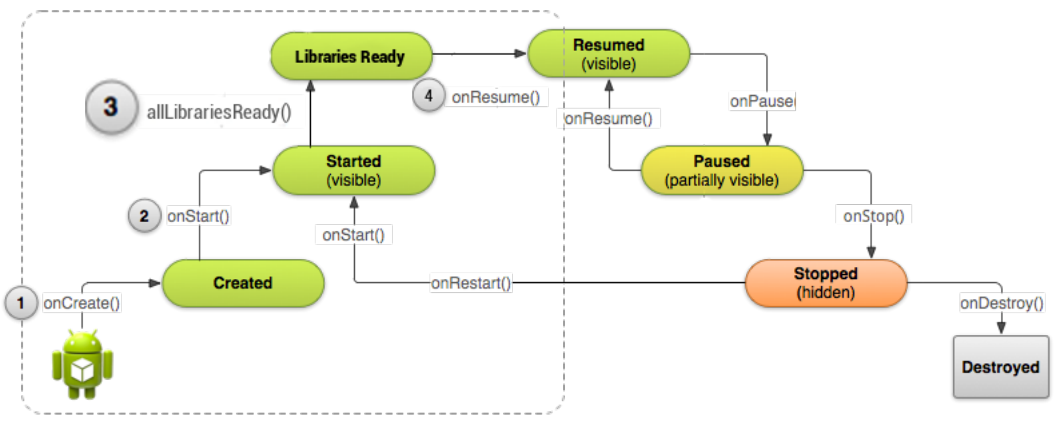


Figure 5.3: The Android Activity Lifecycle, expanded with the allLibraries-Ready callback.[2]

5.10 A Sample Application

To better illustrate the feasibility of this developer-centric approach to dynamic Android permissions, we provide at this point an example application. The example shows the usage of the Guard system by means of the *Internet* and *Bluetooth* permissions, which already have their implemented `PermissionLibraries` available at this point of time. As a concept work, not all permission APIs are implemented yet, only those needed to show the feasibility of the approach. Because most of the APIs can be implemented the same way, the rest is left out to future work.

The application shows two different patterns on how to react to permission revocation in the middle of an application's execution. For the first one, the sample application uses the *Internet* permission to download the source code of the Google starting page to prove the feasibility of the approach, but if the permission is revoked, the invokation of this functionality is simply skipped. A brief `Toast` tells the user about the fact that there was nothing done as a consequence of the lack of legitimation through the permission revocation.
This easy pattern is meant for additional features that are not part of the core functionality. The idea is to simply disable them and if the user could be interestedin knowing about this, display a short explanation for the sake of completeness.

The second pattern describes a scenario with features that need explicit invokation by the user. In our example, the initiation happens by clicking on a button. All the code in the `onClick` callback for this button needs the *Bluetooth* permission, so it is wrapped by a `try` and followed by a `catch` block to react to a `PermissionRevokedException`. So if the permission is not available, the focus jumps to the `catch` block where the execution is abandoned, the button becomes disabled and a short `Toast` tells the user, that this feature is currently not available, because it needs a permission that the user revoked.

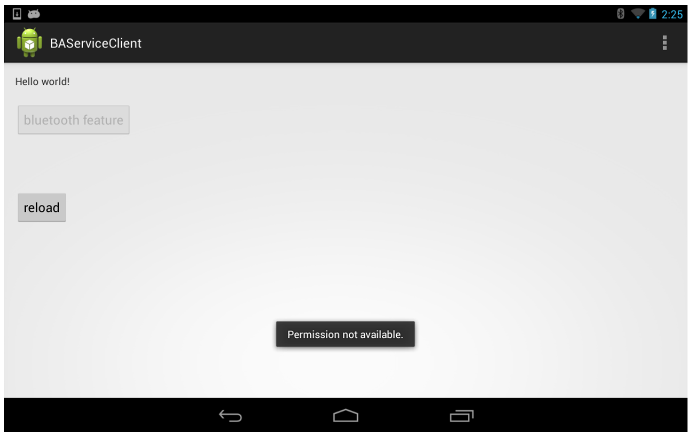


Figure 5.4: The sample application's home screen after the *Bluetooth* permission has been revoked

Now, the user finds himself back on the first screen of the application, depicted in Figure 5.4, but this time the feature button is disabled. The only way to use the feature is to start the GuardApp and to change the permission status so that the application can access the *Bluetooth* API. In general, the user needs a mechanism to recheck if the permission is still not available and to re-enable the button if the permission was granted meanwhile. An apropriate solution is the 'reload' button, located right under the feature button. But we do not like to put the burden on the user to always hit the button after a permission status change to have the feature button enabled. Furthermore, if the application is started for the first time, the button would be enabled although the permission might already have been revoked. But the button does not get disabled before the first hit on it.
To prevent this and to further improve the usability, there is an addition to this pattern. The code executed as a result of the 'refresh' button click is a conventional `Java` method, so it can also be invoked it in the code itself. In the `allLibrariesReady` callback, a call to this button handler method is simply made to ensure, that every time the user switches focus to the application, the re-check takes place and the button's enabling status is correct. Recall that we placed this callback right after `onStart` in the starting sequence of an application, so it is invoked right after `onStart` finished, and because at this point it is guaranteed by the GuardLib that all *permission libraries* are correctly initialized and ready to use, we do not risk to use the `PermissionLibrary` call to `permissionGranted` on an unititialized *Service Connection*.
The result is that when the user starts the *Activity* the first time and the permission is revoked, the button is disabled. Then, after switching to the GuardApp, granting the permission in some way and switching back to the application, the callback sequence is traversed again and the button is enabled.

Note that there are of course also plenty more of these general design patterns for a developer-centric environment with dynamic permission revocation facilities and we are looking forward to some new ones that potential testers may develop.

Chapter 6

Discussion

In this chapter, we briefly evaluate our solution in terms of security and usability. Although we cover a major part of the current Android APIs with our approach to mirror their class hierarchies, there are also permission protected APIs that rely on other mechanisms and thus cannot be covered by this approach. The *SMS* API for example completely relies on *broadcasts*, for which we illustrate a possible solution in the Future Prospects Chapter, Section 7.1.

6.1 Security

The Guard system was designed in a way that prevents or impedes known attacks on similar systems.

The Confused Deputy Attack [29] to Achieve Privilege Escalation

Every system that handles anything similar to 'privileges' needs safety arrangements to assure, that no actor can exploit another one to use its 'privileges'. In Android, the 'privileges' are the permissions that applications hold. And in our special case, the only part that holds original Android permissions is the GuardApp. We already touched on the subject that an application that is fully privileged (recall the GuardApp has *all* Android permissions) must not be turned into a confused deputy. To prevent this kind of attack by a malicious third party application that uses the GuardLib to communicate to the GuardApp, the whole code that can be controlled remotely is encapsulated in the shared interface implementation inside the public *Services* that the GuardApp provides. Because the GuardApp can distinguish requesting applications in a reliable way, explained in Section 5.4, and there is a bijection between the original API hierarchy and the GuardLib's proxy hierarchy, described in Section 4.4, there is no way for an attacker to use the GuardLib in order to turn the GuardApp side into a confused deputy.

Malicious GuardApp Clones

The risk, that someone takes the bytecode of an application, includes some malicious code, and publishes this poisoned version as if it were the original one, is always present. The only way to weaken the attack surface for such approaches is to provide authentification mechanisms and, talking about Android, this would be the fact that an application can be downloaded from a trusted app market and a known developer. But Android allows for the installation of arbitrary untrusted .apk files (packaged applications). So the only real defense is a watchful user. The user needs to install the GuardApp only once and from a trusted source, so the expense of assuring to download and install the correct one, should be moderate.

Fake Dialogs and UI Redressing Techniques

UI redressing techniques and *Touchjacking* [30] (*Clickjacking* on a touch screen powered device) are methods to overlap or substitute trusted areas of the GUI in such a way, that a touch stroke triggers the execution of arbitrary content different than the original one. An example would be to replace a button that does harmless computation with one that triggers a privileged action, like wiping the hard disc or calling high cost phone numbers. This scheme, often used to obtain money or access rights by fraud, is a subcategory that again belongs to the confused deputy problem. But in this case, the confused deputy tricked into executing a malicious order, is the user. In the Guard ecosystem, the 'privileges' to be obtained by fraud are the permissions, so the attack scenario would be to trick the user into granting additional permissions, or to hijack the *ask dialog*. The system design provides two precautions to prevent this kind of fraud. We already touched on the first one in section 5.6, the realization of the GUI. To prevent subsequent modification of the *ask dialog*, the responsible code is intentionally located on the trusted GuardApp side. The consequence is, that a malicious application cannot alter the original dialog, as it is not a part of the GuardLib. So, another attack scenario could be an application that provides an alternative *ask dialog* to trick the user into granting additional permissions. This would require a mechanism to change a permission state from a third party app, for example the possibility of sending an intent to the GuardApplication to do so. But as a second measure of precaution, we abstain from providing such a mechanism to prevent exactly such vulnerabilities. One could argue that it would be a nice feature to have for situations, where the third party app can ask the user to change the permission state because a revoked permission is necessary for core functionality, but the risk that malicious applications would abuse it is too high. As a result, there is only one way for the end user to change permission states, which is to open the GuardApp itself and change the status by hand.

Third Party Applications Using Guard *and* Android Permissions

Subsection 4.2.3 introduced the new contract that our solution provides for developers and users. One part of this ageement is to completely abstain from the default permission system stock Android provides. The reason is, that developers that do not stick to this rule and request stock permissions while also taking part in the Guard system can simply bypass every security check that the GuardApp employs. Having a permission on stock Android, there is

no need to use the GuardLib's API to access critical resources.
In order to prevent applications from masquerading as being privacy-aware by using the Guard system but also use Android permissions to ignore the user decision, the GuardApp has to decline the application's request or mark it in a way that tells the end user, that this app is not trustworthy. In Section 7, Future Prospects, we illustrate how such a security measure can be implemented.

6.2 Usability

Today, no system that adresses a userbase that is as large as the one of Google's Android operating system, can afford to ignore usability. So there are not only security features elaborated in our system, but also some, which only adress the improvement of the usability of the overall system. In order to achieve the convenience of the user, we offer well structured and well designed tools to provide the options and abilities that a dynamic permission system offers, in a preferably comprehensive way.

The convenience of the user, while using an application, partly relies on the skills of the developer, but also heavily relies on the design of the whole system, that the application is built on.
One of our contributions is the graphical user interface of the GuardApp in general. Its duty is on the one hand to display the current permission states of third party applications, and on the other hand to provide a comprehensive tool to decide on them.

Another one is to provide the opportunity for applications to advertise the permissions they want to use at the very beginning. The result is, that the user can decide on the permission states without the need to try out all features of an app to disclose all needed permissions. Right after the first start of the third party application, the GuardApp's graphical user interface is able to display all the needed permissions for the user to investigate and to decide on.

The main goal of this usability improvement is to ensure, that the user does not have a significantly decreased user expercience, while using Guard system applications compared to the usage of standard Android apps. The possibility to decide on permissions *before* running the core features of the application, in contrast to interrupting the app usage to change the permission state, yields a non-negligible gain in the end user's convenience factor.
The ultimate goal is that the user perceives no difference in the usage of stock Android apps and Guard system apps while implementing a dynamic permission system.

Chapter 7

Future Prospects

This thesis assumes the role of a proof of concept work. So, in the design and implementation phase, many ideas for additional features arise, but cannot be implemented because the time limit demands a concentrated work on the core functionality.

7.1 Improvements for the Current Application Based System

Broadcast Based APIs

The biggest part of Android's permission protected APIs can be seen as simple class hierarchies which this approach can mirror. But there are also others that need to be considered, if the goal is to implement all permissions, acquirable for application developers. The SMS API, for example, is fully based on *Broadcast Receivers*, which are currently not covered by this work. The straightforward way to implement this functionality is to receive the broadcasts on the GuardApp side and forward them as explicit intents to the third party application.

Announce Permissions at Install Time

We put a lot of effort in improving the usability of the whole system, but there is still more to be done on this issue. The *announcePermissions* feature ensures that there is no need for the user to inspect each functionality of the app before being able to decide on the status of the correlated permission. But the application needs to be started once in order to use this feature. This can be avoided by implementing a mechanism that reacts to the *new application installed* broadcast on the GuardApp side and explicitly asks the newly installed application for the permissions it is able to use. The responding counterpart would be implemented as a part of the GuardLib.

Further Patterns For Developers

With our sample application, we presented some useful patterns which help developers to make their applications aware of the chances and risks of a dynamic permission system. There are of course lots of additional patterns that could help developers as well, so there is also a demand for even more illustrative example code.

Discovering Breaches of Our Contract

One feature that would significantly increase trust in the overall system, would be to check applications that communicate with the GuardApp the first time, if they have requested access to Android permissions, and block them if so. The reason for this is, that the security properties of the system demand a cooperating developer in order to hold, so by additionally using the Android permissions, the developer is able to bypass all control of the user, that this system desires to grant.

More Fine-Grained Permissions

As already mentioned, some approaches [3][4] suggest to replace the current Android permissions by more fine-grained ones to give the end user even more facilities to control the application's use of critical resources. For simplicity, we did not implement this in our approach from the very beginning, but it yields a further improvement of the privacy situation of the user. Furthermore, this could also improve the understanding of the developer towards the exact responsibility of specific permissions, which currently not seems to be given, as stated by [31] and [32].

Using this approach to more fine-grained permission on the *Internet* permission, to only permit access to adresses from a whitelist of servers, could easily be implemented in our approach, too. All that is needed is a new permission state that carries the whitelisted servers and that can only be assigned to the *Internet* permission, plus a GUI element to manage the whitelist.

Arbitrary Complex Automated Decision Making

We briefly mentioned that the *ask* option is not restricted to obtaining user decisions from. The code is programmed against an interface `InteractionAgent` to keep the code base open for extensions, without the need for modifications. For example, there could be a learning agent that observes the behaviour of either the user, or the applications, to be in a position to answer the *ask* requests autonomously. In combination with the server URL based fine-grained *Internet* permission proposed before, the agent could observe the accept and decline behaviour of the user to learn which servers are to be considered trustworthy.

7.2 Towards A System-Centric Implementation

While the improvements above only apply to the proof of concept prototype, the ultimate goal is still a proper implementation of a dynamic, developer-centric permission system, like the one proposed here, in the Android operating system and as a part of the software development kit (SDK) itself.
To realize such a system based on the idea of this work, one would need to move the GuardLib's code and parts of the code from the GuardApp to system components, like the `PackageManager`, which currently takes care of permission checks as an entry point to the Android middleware.
Critical resource APIs would mostly stay the same, because all the change happens behind the scenes and the developer only needs to know, how to use the newly given tools. Furthermore, in addition to the prototype introduced in this work, a system-centric implementation could also cover user-defined permissions the same way as system permissions and could also handle native code. The implementation of the tools proposed in this work would be straightforward, because there is already a mechanism in the Android system to check whether a permission is present, located in the `PackageManager` class. Probably the most delicate issue would be modifying the lower levels to grant and revoke membership in the Linux groups, corresponding to the permissions that are based on group membership.

However, as these are major changes to the environment that third party applications run in, this approach suffers from legacy compliance with all current applications and requires a gradual roll-out or a double-tracked transition strategy.
Maybe this is exactly the reason for hiding the new permission manager extensions [2] in *Jelly Bean* 4.3, which in its current design does not really seem to be developer-aware and causes application crashes due to uncaught security exceptions for dynamically revoked permissions at the moment.

Chapter 8

Conclusion

We have presented a feasible prototype for a developer-centric dynamic Android permission framework. As a supplement to current research in this area that we discussed in section 3, this approach substitutes Androids permission-based access control mechanism by our new approach that allows for dynamic permission granting and revocation at runtime. In order to prevent undefined behaviour and crashes of third party applications, the proposed system follows a developer-centric approach by providing facilities to react to decisions of the end user at runtime. This work bridges the gap between proposed permission framework improvements or substitutions, and the developers that need a stable environment to programm their applications in. In conclusion, we see many opportunities to further enhance this work. On the one hand, on the introduced prototype, but also on the other hand by initiating a proper system-centric realization without losing the developer-centricity. However, we hope that there will be a dynamic permission framework in stock Android one day, and as it was demonstrated in several papers and theses, including this one, this does not seem to be far-fetched.

Appendix A

The Shared AIDL Interface of the Internet Service

```
package de.uds.infsec.os.baguard.controller;

interface IInternetService
{
  //URL
  String createURL(String spec);
  String URLgetAuthority(String urlID);
  int URLgetDefaultPort(String urlID);
  String URLgetFile(String urlID);
  String URLgetHost(String urlID);
  String URLgetPath(String urlID);
  int URLgetPort(String urlID);
  String URLgetQuery(String urlID);
  String URLgetRef(String urlID);
  String URLgetUserInfo(String urlID);
  void openConnection(String urlID);

  // HttpURLConnection
  void URLCONsetConnectTimeout(String urlID, int timeoutMillis);
  void URLCONsetDoInput(String urlID, boolean newValue);
  void URLCONsetReadTimeout(String urlID, int timeoutMillis);
  void URLCONsetRequestMethod(String urlID, String method);
  void URLCONgetInputStream(String urlID);
  void URLCONconnect(String urlID);

  //HttpInputStream
  int URLInStreamReadBuffer(String urlID, inout byte[] buffer);
  int URLInStreamReadSingle(String urlID);
  int URLInStreamReadBytes(String urlID, inout byte[] buffer,
              int offset,int length);
  int URLInStreamAvailableBytes(String urlID);
  void URLInStreamClose(String urlID);
}
```

Bibliography

[1] Cyanogenmod issue tracker. *request to again include the removed 'Revoke App Permissions' feature.* May 2013. URL: `https://jira.cyanogenmod.org/browse/CYAN-28`.

[2] Ron Amadeo (androidpolice.com). *App Ops: Android 4.3's Hidden App Permission Manager, Control Permissions For Individual Apps! [Update].* 2013. URL: `http://www.androidpolice.com/2013/07/25/app-ops-android-4-3s-hidden-app-permission-manager-control-permissions-for-individual-apps/`.

[3] Michael Backes, Sebastian Gerling, and Christian Hammer. "Appguard-real-time policy enforcement for third-party applications". In: (2012). URL: `http://scidok.sulb.uni-saarland.de/volltexte/2012/4902/`.

[4] Jinseong Jeon et al. "Dr. Android and Mr. Hide: Fine-grained security policies on unmodified Android". In: (2011). URL: `http://drum.lib.umd.edu/handle/1903/12852`.

[5] Benjamin Davis and Ben Sanders. "I-arm-droid: A rewriting framework for in-app reference monitors for android applications". In: *Mobile Security ...* Dvm (2012). URL: `http://mostconf.org/2012/papers/28.pdf`.

[6] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[7] The Android Open Source Project (source.android.com). *Android Security Overview.* URL: `http://source.android.com/devices/tech/security/`.

[8] Thorsten Schreiber. "Android binder". In: (2011). URL: `http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf`.

[9] The Android Open Source Project (developer.android.com). *'Manifest.permission' API reference.* URL: `https://developer.android.com/reference/android/Manifest.permission.html`.

[10] Google (android.googlesource.com/). *Android source code 'frameworks/base/-data/etc/platform.xml'.* URL: `https://android.googlesource.com/platform/frameworks/base/+/master/data/etc/platform.xml`.

[11] W Enck, M Ongtang, and P McDaniel. "Understanding android security". In: *Security & Privacy, IEEE* (2009). URL: `http://ieeexplore.ieee.org/xpls/abs\_all.jsp?arnumber=4768655`.

[12] The Android Open Source Project (developer.android.com). *'PackageManager' API reference.* URL: `http://developer.android.com/reference/android/content/pm/PackageManager.html`.

[13] Dan Bornstein. "Dalvik vm internals". In: *Google I/O Developer Conference.* Vol. 23. 2008, pp. 17–30.

[14] U Erlingsson. "The inlined reference monitor approach to security policy enforcement". In: January (2003). URL: `http://dspace.library.cornell.edu/handle/1813/5628`.

[15] Cyanogen (Steve Kondik). *Statement against the 'Revoke App Permissions' feature.* May 2013. URL: `https://jira.cyanogenmod.org/browse/CYAN-28`.

[16] Peter Hornyack et al. "These aren't the droids you're looking for". In: *Proceedings of the 18th ACM conference on Computer and communications security - CCS '11* (2011), p. 639. DOI: `10.1145/2046707.2046780`. URL: `http://dl.acm.org/citation.cfm?doid=2046707.2046780`.

[17] Yajin Zhou et al. "Taming information-stealing smartphone applications (on android)". In: *Trust and Trustworthy Computing* November 2009 (2011). URL: `http://link.springer.com/chapter/10.1007/978-3-642-21599-5\_7`.

[18] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. "Apex: extending Android permission model and enforcement with user-defined runtime constraints". In: *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security.* ASIACCS '10. Beijing, China: ACM, 2010, pp. 328–332. ISBN: 978-1-60558-936-7. DOI: `10.1145/1755688.1755732`. URL: `http://doi.acm.org/10.1145/1755688.1755732`.

[19] The Android Open Source Project (developer.android.com). *'Bound Services' API guide.* URL: `http://developer.android.com/guide/components/bound-services.html`.

[20] The Android Open Source Project (developer.android.com). *'BluetoothDevice' API reference.* URL: `http://developer.android.com/reference/android/bluetooth/BluetoothDevice.html`.

[21] Apple (developer.apple.com). *iOS 6.0 new developer-related features.* URL: `https://developer.apple.com/library/ios/releasenotes/General/WhatsNewIniOS/Articles/iOS6.html`.

[22] The Android Open Source Project (developer.android.com). *Android Interface Definition Language (AIDL).* URL: `http://developer.android.com/guide/components/aidl.html`.

[23] The Android Open Source Project (developer.android.com). *'Binder' API reference.* URL: `http://developer.android.com/reference/android/os/Binder.html`.

[24] The Android Open Source Project (developer.android.com). *'Internal Storage' API guide.* URL: `http://developer.android.com/guide/topics/data/data-storage.html#filesInternal`.

[25] Aleksandar Gargenta. *Deep Dive into Android IPC/Binder Framework (html version of AnDevCon IV talk on Dec 5th, 2012).* URL: `https://thenewcircle.com/s/post/1340/Deep_Dive_Into_Binder_Presentation.htm#slide-12l`.

[26] The Android Open Source Project (developer.android.com). *'RemoteException' API reference.* URL: `http://developer.android.com/reference/android/os/RemoteException.html#RemoteException(java.lang.String)`.

[27] The Android Open Source Project (developer.android.com). *'BroadcastReceiver' API reference.* URL: `http://developer.android.com/reference/android/content/BroadcastReceiver.html#ReceiverLifecycle`.

[28] Dianne Hackborn on the 'Android Developers' Google group. *Post: Creation and termination of Content Providers.* URL: `https://groups.google.com/forum/#!msg/android-developers/NwDRpHUXtOU/jIam4Q8-cqQJ`.

[29] Norm Hardy. "The Confused Deputy". In: *ACM SIGOPS Operating Systems Review* 22.4 (Oct. 1988), pp. 36–38. ISSN: 01635980. DOI: `10.1145/54289.871709`. URL: `http://portal.acm.org/citation.cfm?doid=54289.871709`.

[30] Tongbo Luo et al. "Touchjacking attacks on web in android, ios, and windows phone". In: *Foundations and Practice of Security* 1017771 (2013). URL: `http://link.springer.com/chapter/10.1007/978-3-642-37119-6\_15`.

[31] Timothy Vidas, Nicolas Christin, and Lorrie Faith Cranor. "Curbing Android Permission Creep". In: *W2SP*. 2011.

[32] Adrienne Porter Felt et al. "Android Permissions Demystified". In: *Intents* 22 (2011), pp. 627–637. ISSN: 09581669. DOI: `10.1145/2046707.2046779`.