FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Security Challenges in Mobile Middleware

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Peter Aufner
Matrikelnummer 0825073

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Mitwirkung: Markus Huber, MSc.

Wien, 16.09.2013

(Unterschrift Verfasser) (Unterschrift Betreuung)

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Security Challenges in Mobile Middleware

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Informatics

by

Peter Aufner
Registration Number 0825073

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Privatdoz. Dipl.-Ing. Mag.rer.soc.oec. Dr.techn. Edgar Weippl
Assistance: Markus Huber, MSc.

Vienna, 16.09.2013	(Signature of Author)	(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Peter Aufner
Attemsgasse 5/2/101, 1220 Vienna

Danksagung

Ich möchte mich an dieser Stelle bei meinen Eltern, Doris und Wilhelm Aufner, bedanken.

Sie haben es verstanden meine kindliche Neugierde über Jahre in wissenschaftliches Interesse zu verwandeln, indem sie mich geleitet haben Fragen zu stellen und Antworten zu fordern und mich schon früh mit Technologie in Berührung gebracht haben, die schließlich mein Forschungsgebiet werden sollte.

Acknowledgements

First, I would like to thank Dr. Edgar Weippl for his great supervision of my work, as well as Dr.-Ing. Darren Carlson for initiating the Ambient Dynamix project and thus providing the ground upon which I was able to learn very interesting things about Android and security on the operating system.

Then, I would like to thank the faculty of Informatics of the Vienna University of Technology for awarding me a grant that allowed me to buy devices to better test my results.

My thanks go to Markus Huber, MSc. as well, for his input in the final phase of my thesis.

Finally, I would like to thank Dipl. Ing. Jürgen Wallner for being a great friend over many years, for helping me remember the right reasons to do science when I was slowly forgetting them and for all his important input to my work.

Abstract

Mobile computing platforms, like smartphones and tablet computers, are becoming a commodity nowadays. To simplify development for these devices it seems like a good idea to offer a middleware solution so developers can pool common functionality into plugins, thus saving space on the device while enabling easier development of more functionality. However, mobile platforms like Android never expected integration in the sense, that one application would dynamically host pieces of code from different vendors and allow access to other applications, since doing so basically circumvents many built-in security measures of the operating system.

The already existing Ambient Dynamix Framework was chosen as a sample implementation for a middleware solution for this work, since it provides a modern platform for Android and is entirely open source. However, with a solution like this, several problems arise:

Android only provides a per-application permission system that does not allow to separate code inside a single application from other pieces of code. It is also difficult to prevent applications from accessing services provided by another application. Finally, the plugins coming mostly from remote sources need to be authenticated as various attacks could allow to intercept the download and replace a benign plugin with a malicious one, or even set up a completely malicious repository that could lead to a total compromise of a device.

So, the first step is to thoroughly understand the Android security model and the relevant differences to the desktop Java platform. Knowing those, a solution needs to be found that does not require any type of modification to the Android operating system. APIs may be used to gain information about the applications and their rights in order to prevent privilege escalation. For the task of preventing downloaded plugins from misbehaving, we will employ static code analysis.

In the course of this work, solutions for the problems are given. The Android security architecture is leveraged to introspect calling applications and assert their permissions. Two libraries have been created to check if there are differences between requested permissions of plugins and another one to handle signing of the plugins from the sources to prevent fraud.

Kurzfassung

Mobile Computer wie Smartphones und Tablets werden immer mehr zu einer Selbstverständlichkeit. Um die Entwicklung für solche Geräte zu vereinfachen, erscheint es als gute Idee eine Middleware Lösung zu entwickeln, um gemeinsam genützte Funktionalität in Plugins lagern zu können und so Platz zu sparen und Entwicklung von mehr Funktionalität zu erleichtern. Jedoch haben mobile Plattformen wie Android nie vorgesehen, dass eine Applikation dynamisch Teile von Code unterschiedlicher Hersteller beherbergt und anderen Applikationen Zugriff auf diese gibt. Dies umgeht das gesamte Sicherheitssystem von Android.

Das bereits bestehende Ambient Dynamix Framework wurde als Beispielimplementierung einer Middleware für diese Arbeit gewählt, da es eine moderne Plattform für Android bietet und vollständig open source ist. Mit einer Lösung wie dieser ergeben sich jedoch einige Probleme:

Android bietet nur ein pro Applikation Rechte System, das es nicht erlaubt den Code einer einzelnen Applikation in geteilten Bereichen zu sehen. Es ist schwer eine Applikation davon abzuhalten Services, die eine andere bietet, aufzurufen. Schließlich müssen Plugins von entfernten Quellen authentifiziert werden, da verschiedene Attacken erlauben den Download zu verändern und anstatt eines gutartigen Plugins ein bösartiges zu laden, oder sogar ein komplett bösartiges Repository einzurichten das zu einer kompletten Kompromittierung des Geräts führen kann.

Also ist der erste Schritt das Android Sicherheitsmodell und relevante Unterschied zu Desktop Java vollständig zu verstehen. Mit diesen Kenntnissen wird eine sichere Endnutzerlösung gesucht, die keine Modifikationen am Android Betriebssystem benötigt. APIs können genutzt werden um Informationen über Applikationen und deren Rechte zu erhalten um Privilege Escalation vorzubeugen. Um das System vor bösartigen Plugins zu schützen, werden wir statische Codeanalyse verwenden.

Im Rahmen dieser Arbeit werden Lösungen zu den genannten Problemen geboten. Die Android Sicherheitsplatform wird genutzt um die Berechtigungen von Applikationen zu prüfen, die Code aufrufen. Es wurden zwei Bibliotheken erstellt um die angeforderten mit den genutzen Berechtigungen abzugleichen und eine weitere um das Signieren von Plugins zu übernehmen um Fälschungen zu vermeiden.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Mobile devices play an important role in the daily life of many people nowadays. They have become the central information interchange as well as a way to pass time. The many uses of such devices create a requirement of relatively large storage capacities to store multiple, so called, apps, that all serve specific purposes. However, there is only a limited set of Application Programming Interfaces (APIs) and libraries provided by the operating system. Due to the secluded nature of the many applications, this leads to a situation where lots of storage is wasted by redundant code. Therefore it is important to find a solution that empowers the mobile devices by implementing a secure middleware that saves storage space and allows the device to run a larger amount of apps. It should also help the device fit into the proper place in an ambient living world, where it is the ideal remote and center to interface with many kinds of devices via standard communication means. In order to solve this issue, the Ambient Dynamix project has recently started the development of a middleware solution that aims to encapsulate more common functionality than the APIs of the Android mobile operating system provides. The project is open sourced and allows anyone to develop plugins without requiring any payment. Developers can apply to be hosted in the official repository or create their own repositories. The middleware uses a modern software architecture to utilize the resources the Android operating system offers, along with Open Services Gateway initiative (OSGi) technologies to efficiently handle the encapsulation of various plugins.

The official presentation describes Ambient Dynamix as a plug-and-play framework that adapts to the user's environment. It comes with a set of plugins that already cover a large variety of daily needs and allows developers to easily extend the functionalities through open APIs. *Adaptive OSGi-Based Context Modeling for Android.* [1]

Since Ambient Dynamix itself is treated as one of many apps by the Android operating system, there are conflicts with the native security framework of the Android operating system, that need to be resolved. Android will regard any plugin that is run inside Ambient Dynamix as being Ambient Dynamix. So, if there is a malicious plugin, Android will not prevent it from running. Without proper security measures, a single plugin could lead up to a total compromise of the mobile device. This could be rooting and the installation of a backdoor, or leakage of the device owner's contact information. Without the proper means, Ambient Dynamix could be abused by malicious developers.

This work will address the solution of the following three security questions:

1. How can we prevent privilege escalation through Ambient Dynamix by applications that do not request permissions from the Android operating system but use plugins that would require them? 5.1

2. How do we prevent plugins, especially from untrusted sources, from exercising unexpected behavior, ones for which they do not have requested permissions? 5.2

3. How can the download process from repositories be secured in a way, so the user can trust, that they are connected to the right repository and get the plugins they requested? 5.3

1.2 Problem Statement

Due to the feature rich nature of mobile devices it is plausible to use them as head units for other devices like heart-rate monitors, pedometers or as Quick Response (QR) code readers. To simplify development for such specific tasks it seems like a good idea to offer a middleware solution. This way developers can access data gathered from other devices easily and also save space on the device by enabling multiple applications to share the same code for specific tasks. However, mobile platforms like Android never expected integration in the sense, that one application would dynamically host pieces of code from different vendors and allow access to other applications, since doing so circumvents many built-in security measures of the operating system. In other words, Ambient Dynamix, the middleware, has to request a huge set of permissions, while the calling applications, in theory, would not need to reveal what they are doing anymore, since they could just delegate the task to a plugin inside Ambient Dynamix.

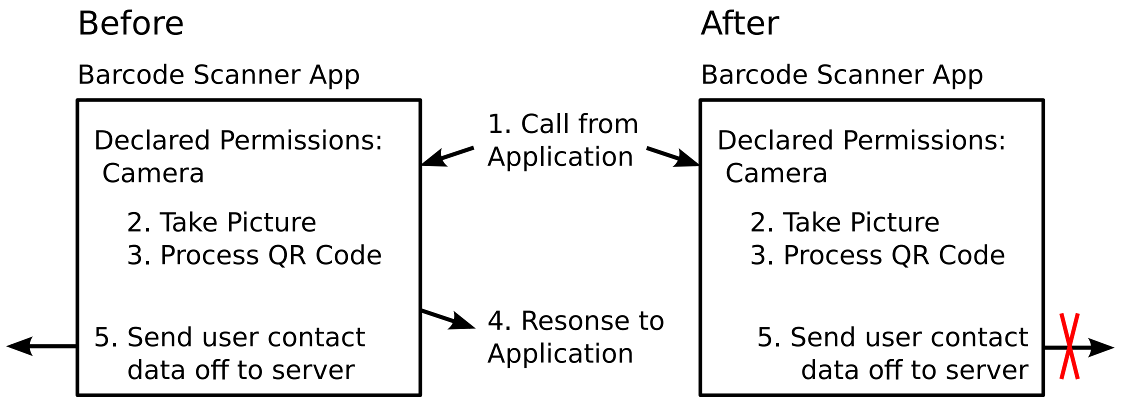
Before
Barcode Scanner App
Declared Permissions: Camera
2. Take Picture
3. Process QR Code
5. Send user contact data off to server
1. Call from Application
4. Resonse to Application
After
Barcode Scanner App
Declared Permissions: Camera
2. Take Picture
3. Process QR Code
5. Send user contact data off to server

Figure 1.1: A plugin that runs unexpected code before the security patches and the block afterwards

On the other hand, there are plugin developers that offer specific blocks of code. Let us presume some developer claims to offer a plugin that does nothing but read QR codes and in case they contain specific information, prepare it for use by another app. Normally, if this plugin was installed as a common app, the Android operating system would enforce security through the permission system. The plugins are running inside the perimeters of the Ambient Dynamix framework. Thus they have the same, namely almost all, permissions as the framework has. Therefore a strong mechanism is required to regain security by checking if a plugin really only uses the permissions it claims to use, or if it sneakily tries to access data it should not, i.e. access contact information and send them off to a third party. This idea is illustrated in figure 1.1. This sums up the second large problem area that will be addressed.

Finally, since the plugins can be downloaded dynamically, it is not only about the developer doing unwanted things but also about insecure communication. It is well-known, that on any type of network, be it the internet or a Local Area Network (LAN), other subscribers could introduce fake content. So, while the original package would be completely benign, someone with malicious intent could try to swap it out for something with nefarious purpose. This calls for mechanisms to authenticate both: the origin of the repository as well as a cryptographic signature of all packages to allow the app to detect such attacks and warn the user about what is going on.

1.3 Aim of the Work

This work aims to introduce the security goals mentioned in section 1.1 into the existing Android application Ambient Dynamix. A way to ensure the authenticity of plugins will be implemented, a check for the correct behavior of the plugins will be added and apps will only be allowed to use plugin content, that requires permissions they also requested. It is well-known, that security always has a trade-off to performance and usability. However, the code to enforce security shall be run directly on the mobile devices, therefore attempts will be made to keep the code fast on devices with limited processing power.

1.4 Methodological Approach

The first step is to thoroughly understand the Android security model and the modifications that were made to the original Java platform. Through the study of the matter so far, it has become clear, that some security features Java would offer are missing in Android. Still, a solution needs to be found that does not require any type of modification to the Android operating system, in other words, a secure end-user solution is the goal, this means:

- Assert, that plugins do only what they claim to do, enforced as Android would do it.
- Make sure users get the plugin they expect.
- Handle access from the various applications for the user.

APIs may be used to gain information about the applications and their rights in order to prevent privilege escalation. This means, the framework will be able to check the rights an application has acquired from Android and will not allow it to use plugins that would require more rights than it has.

For the task of preventing downloaded plugins from misbehaving, several approaches seem possible:

Aspect oriented programming may be used to modify their behavior. This means introducing pointcuts for actions like opening of files, that would prevent the action from actually taking place. These could be implemented in the SDK to enforce it on any plugin.

Another solution would be decompilation and static bytecode manipulation to remove unwanted code segments, or to simply forbid running of applications that want access to unexpected resources. This approach may only be feasible as a server-side solution though. Several steps, like restoring the files back to the proper state for use in the Dalvik VM, are required to execute this solution. These are rather resource intense and may quickly push a lower class smartphone to its limits.

The possibility of runtime debugging will be investigated to block plugins from malicious activity by interrupting them as necessary at runtime.

Finally, the examination of used methods and classes inside the compiled code of the various plugins will be investigated as a relatively simple way to statically analyze the structure of an application.

To provide authenticity of repositories as well as of packages, public key cryptography seems to be the best solution, especially since there are standard implementations openly available, thus allowing for proven security concepts without the need to reinvent the wheel.

1.5 Structure of the Work

This work is structured as follows:

- The chapter 2: explains the important concepts used in the creation of the solutions.
- In the chapter 3: State of the Art, we will examine work that has been done in the field of Android security and other topics related to the purpose of this work.
- The chapter 4: Methodology aims to explain the most important background knowledge of Android and Ambient Dynamix as well as the concepts used to create this work.
- In the chapter 5: Solution we will discuss the most important ideas that lead to and of course the final solution to the security questions that were open in Ambient Dynamix.
- An analysis and comparison between the State of the Art methods and the new solutions will be given in the chapter 6: Reflection.

CHAPTER 2

Background

2.1 Used Concepts

In this section, we will discuss the most important pieces of knowledge to start working on the solution.

2.1.1 A Brief Overview of the Android Security Model

To understand the difficulties encountered during the development of the security goals for Ambient Dynamix, let us first take a look at the security concepts of the Android operating system.

An Introduction to the Basic Structure of the Android Operating System

Figure 2.1 gives an overview of the structure of the Android operating system.

The topmost layer of Android is the application framework. It consists of several services like the PackageManager, or the Activity Manager, as well as hardware services to get location information for example.

Below that are the native libraries that allow to use SQLite databases for data storage or Webkit for web browsing. This layer also holds the Android runtime consisting of the DalvikVM, Android's implementation of a virtual machine in which the applications are run, and the core libraries.

The bottom layer of Android is based on the Linux kernel. It provides what is common functionality like memory- and process management but has been enhanced with power management capabilities for mobile devices as well as the logger and Inter-Process communication(IPC).

It is important to know, that on Android every application runs in its own virtual machine that is mapped with a unique user id. A reference monitor mediates IPC calls between the various running apps. This is important knowledge for the Ambient Dynamix framework, since it is only capable of occupying a single user on the Android system. Android only sees the many small plugins as one application and cannot differentiate their permissions for us, as well as the files

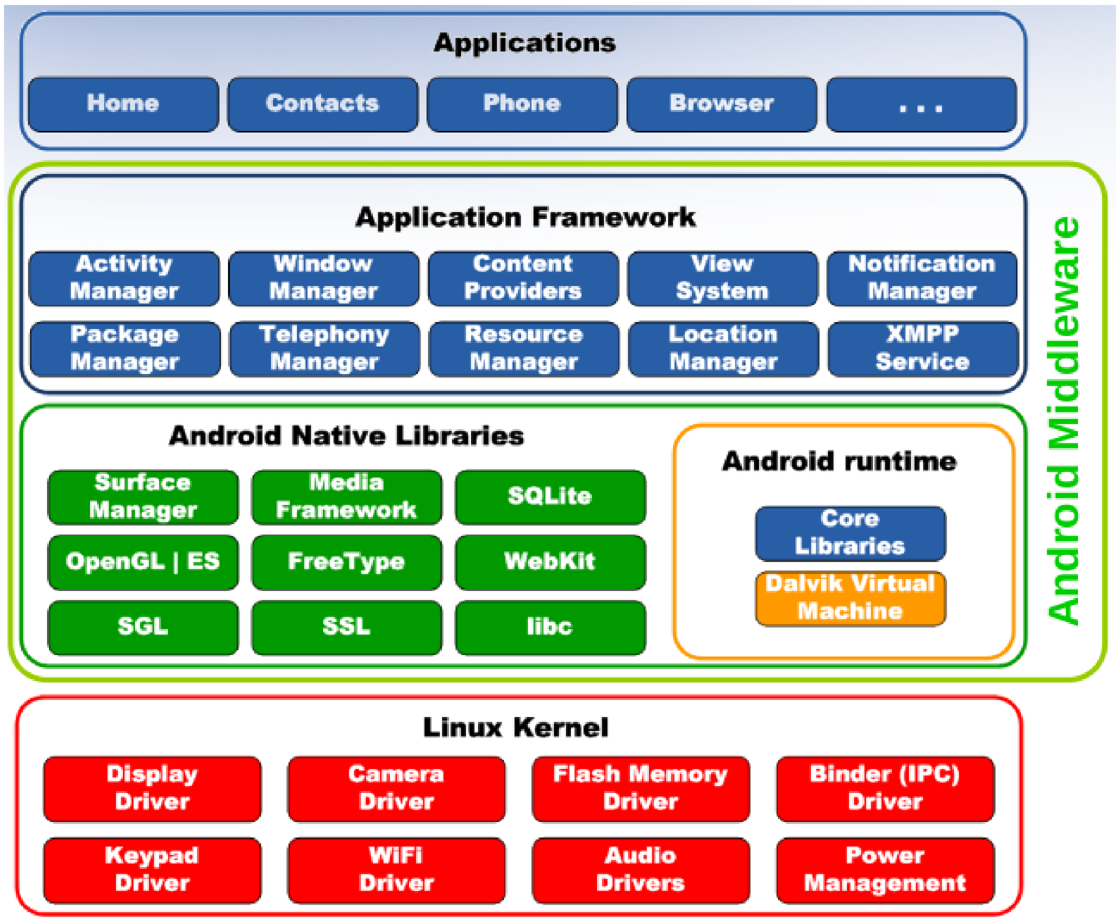

Figure 2.1: A model of the Android operating system as shown in [2]

they should be able to access. Therefore we need to do this ourselves in a way that reflects the same security, that the Android operating system would offer.

Going back to applications, these consist of four main components: Activities, Services, Content Providers and Broadcast Receivers. Three of these parts are included inside Ambient Dynamix:

1. Activities: They build the graphical front end of Ambient Dynamix.

2. Services: A service is active in the background to keep the infrastructure of Ambient Dynamix running at all the time.

3. Broadcast Receivers listen for other applications that request functionality from a plugin.

Components expose APIs for communication, for example, services expose Start, Stop and Bind. This interaction is illustrated in figure 2.2 found in the presentation *Android Security* [2].

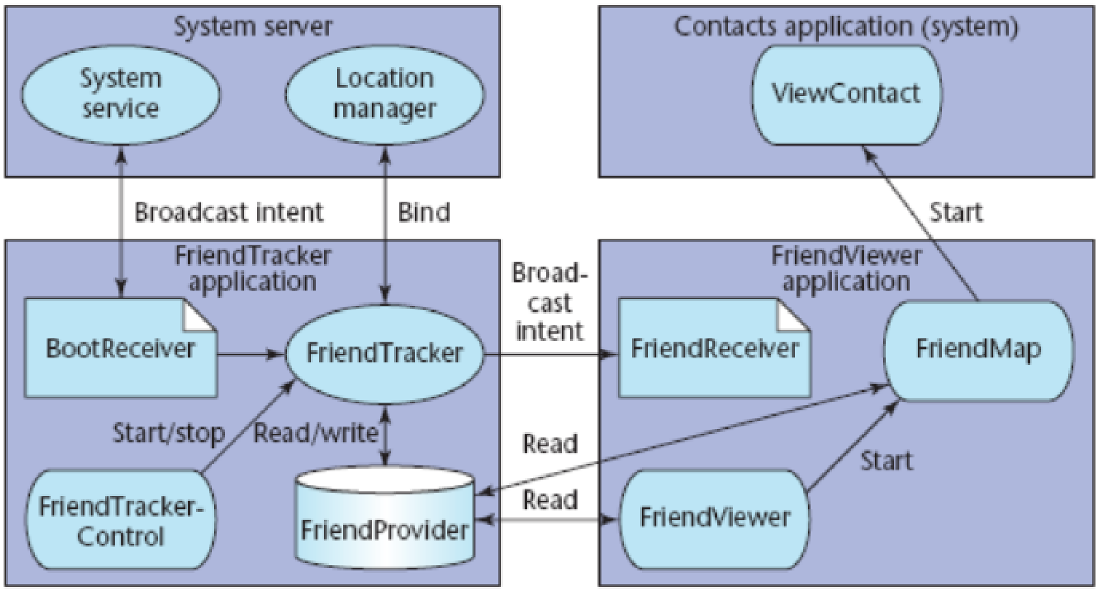

Figure 2.2: Inter-Process Communication in Android as shown in [2]

Inter-Process Communication

This is the core of what makes Ambient Dynamix really useful as it allows other applications to communicate with the plugins. However, it is also the core of the dangers of the use of Ambient Dynamix. Basically, we as the developers need to find out, who is calling our code and then decide for ourselves what, if anything, we are going to do then. Android cannot help us here.

On Android most communication between processes is done by so-called Intents. These are broadcast messages to which applications can subscribe. If an intent is called, the proper application can respond. This is a very lightweight approach to Inter-Process communication that is based on the concept of message passing. Intents use objects that allow the communicating parties to gather some information about the other application, like their permissions. This is an important help for verifying permissions. But, Android does in no way filter who is allowed to call whom. So any application that is sent an Intent object, needs to decide on its own, what to do with it.

Another approach to Inter-Process communication on Android is the use of remote methods. They work similar to remote procedure calls as they are known from Java. Methods can be published through Android's interface definition language (AIDL). They can then be called like local methods that are executed in another process.

Permissions

Anyone who has owned an Android based device and installed an application on it, has already come in touch with the concept of Android permissions. These offer a simple mapping of special classes to human understandable permissions. For example, if an application wants to use the

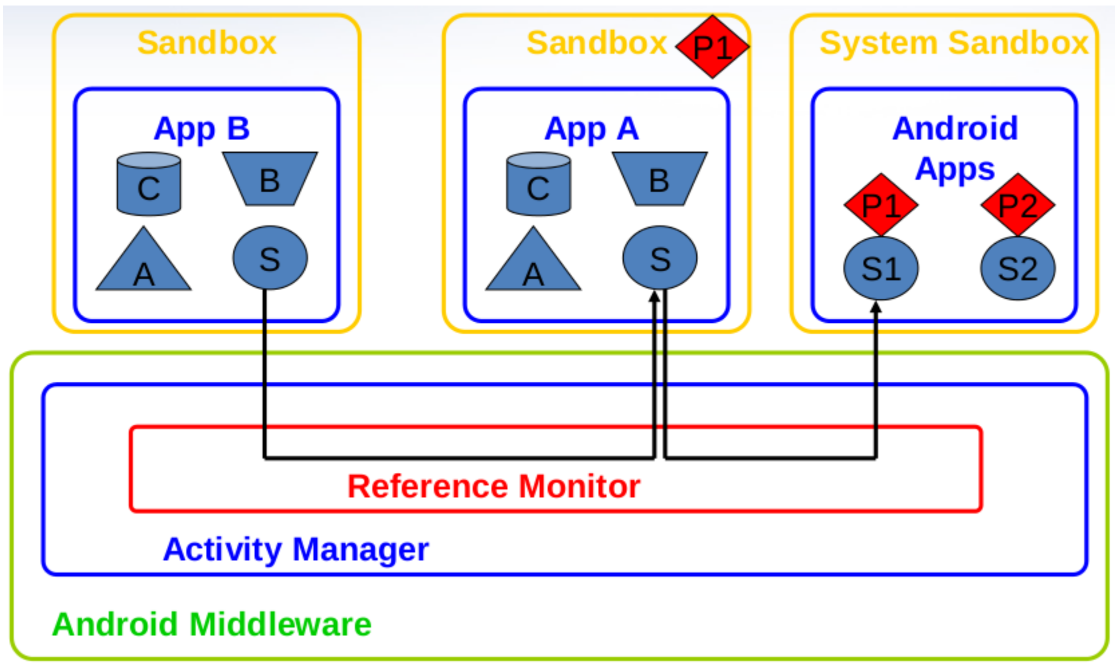

Figure 2.3: Privilege escalation on Android exploiting a chain of security permissions as shown in [2]

camera, it will have to ask the user for the given permission. This, in theory, allows a baseline of security given through the user, who can check the permissions and see, if there is anything strange. For example, a video game asking for permission to access the user's contacts. However, as with desktop firewalls for PCs, this feature also leads to users quickly just saying 'yes' to everything, since they just want to use the software and automatically presume that it really needs whatever permission it requests. Also, there is no option to not grant an application certain permissions and still install and use it. So the users usually end up agreeing to whatever they need to, just to use an application they want.

Important in the case of Ambient Dynamix is the fact, that permissions are granted application wide and cannot be dropped. In other words, if there are parts of an application that should have more rights than another, it could only be done by splitting the application up into multiple applications. Otherwise the whole code will run with the same permissions.

An important loophole with Android permissions is, that one could daisy-chain applications to gain more and more permissions, as shown in figure 2.3 For example, if an application has a certain type of permission granted and exposes an interface to a service that uses functionality of that permission, any calling application can use that service and bypass requesting these permissions. It is worth mentioning, that components are auto-exported and therefore made public to any calling app. While Ambient Dynamix already inquired the user about calling applications, it did not provide any checks for the actual permissions of that app. The solution presented in this work fixes this issue.

Improvements to the Inter-Process communication of Android have been suggested in [2], specifically the implementation of a policy based security framework that hooks into the system and decides which actions are benign and which might be malicious. Most importantly it aims to give the user the means to decide on a fine-grain basis, which actions are trustworthy in which context. Yet, all of them require a patched firmware to be installed instead of the original one, which can be problematic if the device manufacturer has not made all their drivers public.

Applications as Users

The whole of Android security is centered around the fact, that each application is given its own unique user id and group id. This allows Android to keep the applications strictly separated by only employing the standard POSIX file permissions.

On a short side note: Rooting the device effectively undermines that concept since suddenly any application can request root privileges and in doing so access any information from any application.

Differences to Java

It would not do Android's programming language justice to just call it Java. It is mostly the same, but there are certain differences that are of no concern to a regular programmer, but when developing a middleware like Ambient Dynamix, suddenly become troublesome. The most important difference is, that Android makes way more use of reflection than one would when programming for Java. The use of reflection is even considered the gold standard when developing an application that should be backwards compatible while allowing it to use the newest features of newer Android versions. It is probably due to that fact, that the central security feature, the SecurityManager, cannot be modified programmatically from within an application.

Android code is also compiled to run under the so-called Dalvik Virtual Machine, which is the standard for the Android operating system. The fact, that the bytecode is not the same as Java bytecode anymore, makes it necessary to completely rewrite tools that would interact with bytecode. As of now, no solution is available to perform aspect oriented programming at runtime. Weaving can be done at compile time though. Also, static bytecode manipulation is impossible, since any software that should be run on the system needs to be digitally signed, so one would have to re-sign the code on the device, if it was modified.

Other than that, there are various APIs provided to communicate with other processes, as mentioned in this section, or with the various hardware features and sensors of the devices as well as to create the user interface.

Differences between Dalvik and JVM

While it is possible to code Android programs in mostly Java, there are problems with libraries that require more direct access to low-level features. Especially bytecode manipulation, like runtime weaving in aspect oriented programming, is a problem without obvious reason. Therefore it is worth understanding what the big differences between plain Java and Dalvik are:

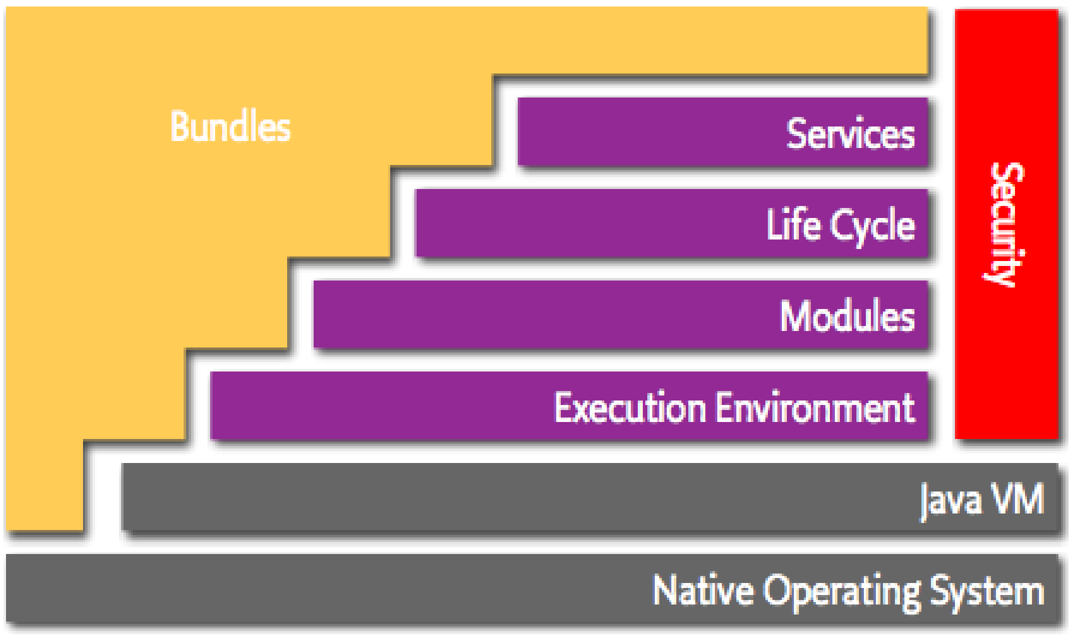

Figure 2.4: The layer model of OSGi as shown on the website of the project [5]

While desktop Java Virtual Machines are mostly stack-based. The Dalvik Virtual Machine(DVM) is strongly register based. This allows for optimal processor usage on the lower powered mobile handsets. Furthermore the DVM was adapted to make the best out of the reduced instruction set that is used by the ARM based processors, it is mainly run on. All in all it has been optimized to run on smartphones.

Security-wise this allows for spawning a new virtual machine for every app run on the device, thus adding an additional layer of protection by keeping applications strictly separated.

What makes the difference for bytecode manipulating libraries is, that the bytecode of Dalvik is not the same as Java bytecode. Rather during compilation, Java bytecode is generated which is then converted using the dx tool to be run by the DVM. This also includes third-party, precompiled Java libraries a project may include. Therefore problems may arise when very complex libraries are included into Android development projects.

Finally the core functionalities are based upon C++ libraries as discussed in *Android: Grundlagen und Programmierung* [3].

2.1.2 An Introduction to the Ambient Dynamix Framework

”Ambient Dynamix (Dynamix) is a lightweight software framework that enables mobile apps and websites to fluidly interact with the physical world through advanced sensing and actuation plug-ins that can be installed on-demand.” *Ambient Dynamix Homepage* [4]

Let us first introduce the OSGi architecture:

OSGi is a set of specifications that standardize dynamic component loading in Java. By allowing components to hide the implementation details, it allows for dynamically composed applications of easily reusable code elements. The means of communication is services. This simple model has a large impact on the whole software development process.[5] Following is a short explanation to the various components shown in the illustration 2.4.

1. Bundles - Bundles are the OSGi components made by the developers.
2. Services - The services layer connects bundles in a dynamic way by offering a publish-find-bind model for plain old Java objects.
3. Life-Cycle - This API is used to install, start, stop, update, and uninstall bundles.
4. Modules - This layer defines how a bundle can import and export code.
5. Security - This layer handles the security aspects.
6. Execution Environment - It defines what methods and classes are available in a specific platform.

Ambient Dynamix is based on OSGi, which is a mechanism to dynamically manage code packages inside Java applications. It allows to keep functional units, easily load them during run-time and define dependencies between them. The implementation used for Ambient Dynamix is Felix by the Apache Foundation.

What Ambient Dynamix allows is code reuse in a way, that would never be possible on Android, unless large changes to the whole application concept of Android were made. The idea of reusing functional units of code once for multiple applications has never been thought of on Android. There are APIs readily available, that cover many important functions to access various services and devices inside the system, but there is no package manager as it would be known from operating systems like Linux. This, though, feels like a large mistake on a mobile platform. While it would make management of installed software quite more difficult than with the current model, it would allow to save lots of storage space, if used properly.

Ambient Dynamix relies on standard Android APIs for Inter-Process communication as explained in section 2.1.1. By the nature of encapsulating the whole functionality of different plugins, it is forced to break the permission system of Android in order to allow for the plugins to work. Illustration 2.5 shows the whole process on how Ambient Dynamix communicates with the internet, sensors and other applications as well as the important aspects of the inner workings of the application:

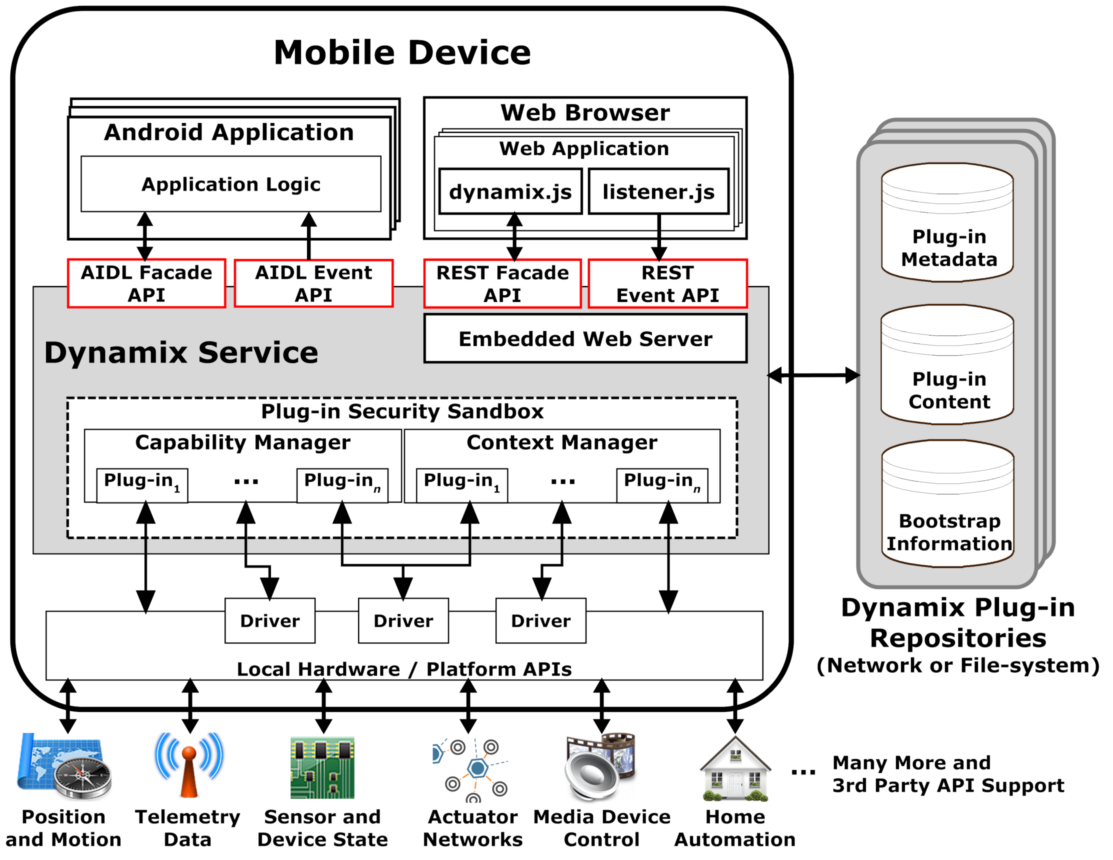

Figure 2.5: The architecture of Ambient Dynamix as shown in *The Architecture of Ambient Dynamix* [6]

In order for an application to use the framework, it is required to implement the Ambient Dynamix API and request the use of a specific plugin from the Ambient Dynamix plugin repository. If the plugin is already installed on the system, permissions will be asked once and afterwards the plugin will be executed on behalf of the calling application. Otherwise the required plugin can be obtained from the official repository, or a private repository on the internet or a local repository stored on the SD card.

2.1.3 An Introduction to Public Key Cryptography

In *Handbook of applied cryptography* [7] Public key cryptography is defined as a form of asymmetric encryption. It generally relies on the generation of a public and a secret key, the former being freely distributed, the latter being kept only by the key owner. In order for two parties to exchange messages, the sender must hold the public key of the receiver to encrypt the message, who then uses her private key to decrypt the message. While this solves the problem of how to exchange the keys, it adds overhead to the encryption and decryption operations. The decryption techniques may also be used for data authentication.

RSA is the most widely used standard, also employed by the popular open source solution: GPG. This standard is based on the factorization problem, for which no linear time algorithm has been found yet.

There is a wide range of attacks against this standard, like attempting to gain the plaintext from a given cyphertext by trying to guess the decryption key. This is possible since modulo operations are applied, limiting the space of possible keys. Further attacks may also be found in the book *Handbook of applied cryptography* [7]. Even this book, first published in 1997 already suggests to use a key-size of at least 1024 bits. Due to the fast development in computing technology, it is more appropriate to use larger keys of at least twice, or better four times the size. Another work *Selecting cryptographic key sizes* [8] from 2001 suggests the necessity of larger key lengths as well.

The most complicated part in public key cryptography is how to share the public key. For using technology like GPG, specialized servers are in place that allow for reliable distribution of the public keys, so users can easily send encrypted messages. For software distributions like in Linux or also Ambient Dynamix it is possible to simply ship the keys of the official repositories along with the software. This builds a base for the chain of trust that is necessary to later download more packages.

2.2 Methods and Models

Following, is a short discussion of two concepts, that played an important role in the development of the solution.

2.2.1 Aspect Oriented Programming on Android

As stated in the section about Languages, the most important difference between Android and Java is the bytecode representation of the programs. Therefore common concepts like Aspect Oriented Programming (AOP) do not work as expected when used dynamically during run-time. This is simply due to the fact, that the common libraries have not been fully adapted to this bytecode yet. AspectJ, for example, supports compile time weaving but no runtime on Android.

2.2.2 Bytecode Manipulation

Another common operation on Java is the manipulation of bytecode during runtime. Again, this is not yet fully possible on Android, but the asm library has been extended to the asmdex library which allows to inspect the bytecode fully and do certain operations.

2.3 Languages

In this section a few important differences between desktop Java and Android's flavor of it will be discussed.

2.3.1 Differences Between Android and Sun Java

	Java	Android
Debugging	Between any two apps via JNI	ADB over USB or Wi-Fi
Security	Using SecurityManager and classes	Enforced by OS via Permissions for API calls
Hardware access	Drivers and OS	API calls
Code Signing	Optional	Required for distribution
Virtual Machine	JVM (stack based)	Dalvik (register based)
IPC	JNI, Network, etc.	Intents and AIDL
Data Storage	Anywhere on disk or in OS specified user dir	Inside app private data dir or on mass storage
Logging	Via Libraries or System.out or .err	Log class
Loading additional code	Include another jar to classpath	Another .apk file that must reside inside application's data dir
Execution	Normal Program	Services in Background, Execution usually only in foreground

Table 2.1: Differences between Java and Android

Table 2.1 was created through research in the respective developer documentations for Android [9] and Java [10].

Developing for Android in general feels pretty much like developing for Java with an additional SDK to access devices like built-in web cams and to interact with other applications as defined in the Android security model. From a security perspective, certain differences can be seen immediately. First of all, the lack of being able to modify the SecurityManager. This is a Java solution to protect from unwanted use of reflection and to restrict code in the way it runs. This would have been very useful for the securing of Ambient Dynamix, but it would not be available without modifying the whole system. Since the whole of Android programming is pretty much based on reflection, both by the operating system but also to keep backwards compatibility manually in applications, it seems like a better choice not to allow developers to modify this element. Also, usually an app will come as a whole package with code, that needs to be signed. It is not a use-case that an application intentionally loads executable code into the system at run-time. Android is pretty much reliant on the idea, that there cannot be any malicious code as also mechanisms, like the so called StrictMode, can simply be switched off programmatically at any point in the source.

CHAPTER 3

State of the Art

Since the work covers several fields of security, all working together to make Ambient Dynamix a secure mobile middleware solution, we will examine the state of the art for the various concerns separately.

3.1 Literature Studies

The following subsections will give an extensive overview of research, that has been done in the fields of information security relevant to this work.

3.1.1 The Android Security System

Bridges, Kishore, and Pedo have examined the gap between security privileges requested by Android applications and the ones they actually use in *Android App Security Analysis* [11]. In their analysis they found some interesting statistics: ”Out of the 30 apps analyzed, 43.33% requested access to the Internet, 30% solicited the user’s location, 30% petitioned for full network access, and 26.67% asked to read the user’s contacts and to view the phone state. Some apps also requested hardware permissions, for example, 26.67% of the apps called for the ability to control the device’s vibrate function.”[11] It does also seem a little concerning that 30% of their tested apps, wanted to read the user’s contacts data.

It is even more concerning, that according to their findings, popular apps like Twitter are able to run underprivileged. That is despite the fact, that the developer documentation claims, that any API call must have the proper privileges reflected in their manifest. However, they put this finding in relation as it could also be carelessness of the developer. In fact, the app crashes during run-time if the non-privileged action is executed. If a developer added some functionality, that required the permission but never uses it, then it would all be fine with the Android permission system. The same was found during the development for this work, as I intentionally tried to run an underprivileged app and it crashed, just as expected.

Another problem of the same category is the one of over-privileged applications, that is tackled by Felt et al. in *Android permissions demystified* [12]. This is actually quite important for the understanding of the Android security system, as this case illustrates how the permission checking on Android actually works. It also means, that it is nearly impossible to re-implement a permission check that is feasible in user space, as we will attempt in this work. In *Curbing Android permission creep* [13] this problem has been examined as well and an Eclipse plugin was developed, that checks exactly which permissions are necessary for an application to run before it's deployed. They created their database by parsing the Android developer documentation, unfortunately, they didn't go into any details on how their final permission map was obtained.

Another concern of the Android permission system is the misuse of privileges. For example, the referenced study found a small application, that claimed to offer password protection, requested the permission to dial numbers. It even contained strings that hinted at the application making use of this privilege. If it was really written with a malicious intent, it could, for example, call expensive numbers without the user's consent. For the case of Ambient Dynamix it shows two things: First, that a user must really trust Ambient Dynamix a lot to allow it to run on their device, since it requests many permissions and could do basically anything with them. The other problem this work addresses is, that if a plugin requests permissions it does not need to declare what it is going to use them for, but only that it is going to use them. So, while we can check and make sure that if code is found requiring a certain set of permissions, it must request them. It is still up to the skeptical user to ask, what it needs them for.

Finally, Shin et al. aim for a complete formal proof of the security of the Android operating system in *Towards formal analysis of the permission-based security model for android* [14] . Their idea is based on a state machine with transitions that are caused by different actions from applications and the user.

To sum it up, while the Android permission system in and of itself is not broken, the way it is used can be quite wrong at times. From my own glance at a popular app, it could also be seen that this is often due to analytic software shipped with apps. They track user input, or use ad-networks that ship targeted advertisements directly into the application. However, the biggest concern is, that users simply start ignoring them, since they just want to use an app and do not care what permissions it needs. Rather, they just think the developers know what they do.

3.1.2 Privilege Escalation

The well-known term 'rooting' basically refers to a permanent form of privilege escalation common on Android mobile devices. Basically, it circumvents the security system by installing a 'su'-binary on the system. It allows any user to claim the identity of another user, specifically the identity of root, the user who can do anything on the system. In *A framework for on-device privilege escalation exploit execution on Android* [15] a framework is explored that would allow the execution of several well-known exploits from directly on the device. While this is good news for anyone wanting to root their Android device, it shows that a malicious piece of code could also be abused to compromise a device without the need of any additional hardware.

3.1.3 Detection of Attacks Against the Intent System

The research presented in *Automatic Detection of Inter-application Permission Leaks in Android Applications* [16] aims to automatically detect three different types of attacks:

1. Permission Collusion: One application raises its privileges by using another, more privileged application.
2. Confused Deputy Attack: Here, an unauthorized caller invokes protected actions in another application.
3. Private Activity invocation: In this case Activities that were not meant to be called from the outside are activated from another application.

The three attacks listed above describe the most common kinds of attacks against the Intent system. In order to automatically evaluate Android applications for these weaknesses, they first built a tool called the permission mapper, that extracts secured API calls from the Android source code. Another application, called the Rule Generator builds the taint analysis rules relating the sources to the corresponding sinks. On the other hand, the manifest is extracted from the application and then fed to the Decision Maker, which checks if weak Intent-Filters are in place. During their analysis, they identified that several applications could be abused to leak e.g. sensitive location information. A similar project has also been completed and described in *A framework for static detection of privacy leaks in Android applications* [17].

In *Analyzing inter-application communication in Android* [18] Chin et al. aim to help Android developers to secure their application by implementing a tool they call ComDroid. They identified another type of attack against the Intent system, which leverages the message oriented nature of the Android Intent system, by intercepting messages and thus gaining information that a third party application should not get, or by using this technique to intercept messages and prevent them from ever being received by the intended receiver. Basically they also aim to mitigate the attacks by giving developers guidelines on how to properly secure exposed services and Intent endpoints.

With this knowledge it is worth studying *Developing secure mobile applications for Android* [19] for a comprehensive overview on how to code securely for the Android platform.

3.1.4 Security in OSGi Environments

In *Supporting the secure deployment of OSGi bundles* [20] a toolsuite is developed that supports the whole OSGi signature standard and deployment of bundles. OSGi has the problem, that while it is possible to preserve sanity of the host infrastructure, three main ways to insert malicious code exist: either during development, or by creating malicious bundles or by creating counterfeit repositories. In this work a toolsuite is generated to sign bundles in a way the developer can be held accountable for what they release. Yet, the standard is not implemented into the main release.

Phung and Sands have evaluated ways to secure OSGi based environments using Aspect Oriented Programming (AOP) in *Security policy enforcement in the OSGi framework using aspect-oriented programming* [21]. Based on AspectJ, a well-known AOP solution, they implement a reference monitor-style policy set and different security states inside the solution. This solution is meant to be used in extensible vehicle software. It is important to mark, that they identified actions, that may not be worrisome by themselves but if repeated too often, could be. For example, if an application utilizes a cellular gateway, to send many SMS messages. They use pointcuts, which are blocks that trigger upon a certain action, to catch a security relevant event in their case. Then the advice is applied to trigger an action based upon a variable inside the aspect objects, which monitors the security states. With all that, their framework is capable of suppressing, replacing, inserting and truncating actions. All these operations are based upon the code that an application has executed so far.

3.1.5 Android Bytecode Instrumentation

Bytecode instrumentation allows one to gain an understanding of an application without having access to the source code. It works by decompiling the given .apk and bringing it back to a human understandable, or at least programmatically analyzable, form. This will be important for the solution, as it gives a bullet proof way of finding the API calls that are being made, that cannot be concealed.

In *Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation* [22] the idea of manipulating an Android application after installation on the device is discussed. During their work, they created two applications that allowed more fine-grain control over permissions and a simple ad remover. The important part of the solution is, to do any bytecode manipulation directly in-vivo, meaning on the phone. So, they found bytecode manipulation to be a great way to dynamically monitor a given application without modifying the operating system or the need of a computer to run ADB. The steps for the process described in [22] are as follows:

1. Code is extracted from the .apk files and converted to Java bytecode.
2. The code is modified with proper manipulation tools, then converted back to Android bytecode.
3. The code is repackaged as a new application with their own signature.

The developed toolchain could also be used on a computer, but is obviously performing well enough on a mobile device. Performance may be improved, if the step of converting the code to plain Java was avoided, but on the other hand, many tools are available to handle Java code, so it definitely is the more convenient solution. Besides the proof that such a solution actually works, the authors also made some interesting findings, e.g. the fact that third-party ad modules may cause up to 75% of the battery consumption of a given application. By removing the ad module from the app, this energy can be saved. The second application greatly helps in improving the security of applications by allowing to selectively revoke certain permissions of an application. For example, if it does not make any sense that an application would need access to contact data, bytecode manipulation can be used to block these API calls. Examining the feasibility, the authors found a median of 120 seconds before the launch of an application if bytecode manipulation was done. They claim it would be a reasonable time for a user to wait, if they were asserted, that the application would behave as expected for set permissions afterwards. Overall, the solution seems pretty reasonable and stable and allows arbitrary modifications to applications. Of course, there are inconveniences to it, for example, it could break the update mechanism, since the .apk file is re-signed with a different certificate and the process needs to be reiterated upon every new version of the application. So, while it is a thorough academic proof, that doing such manipluations are feasible, the real world application might not be that easy. Also, it would probably be better for the end-user to just submit their .apk file to a web service, have it apply modifications as wanted, i.e. remove an advertisement framework, and then get the fresh .apk file back, since two minutes of the phone running at full clock speed, probably drains the battery quite a bit.

3.1.6 Cryptography in Package Management Solutions

Ambient Dynamix as of today uses an XML file that contains a list of links to the various packages available, along with the permissions they declare. This is highly insecure as it allows attackers to manipulate the package management process at various stages:

1. An attacker can spoof the URL of the repository via DNS spoofing or another man-in-the-middle technique. This would redirect the application to a totally different repository without it noticing.

2. Since the index file is not cryptographically signed, no one can detect if it is the right one or the wrong one. So this is the first file that must be signed to reveal attacks.

3. When the user has decided for a package to install, the download starts. The source for the file to download may already be spoofed, but again, the file could also just be swapped for another file. So, it is necessary to cryptographically sign this file.

In order to improve this situation, we'll first examine popular open-source package management solutions and then see a detailed analysis on mistakes, that were overlooked by them.

OpenSuSE - RPM

OpenSuSE in general allows insecure sources, which means in this context, that it is simply a store of many rpm files, without anything to prove that they are as the author and repository maintainer intended them to be.

”Starting with SuSE Linux 10.1 this problem [of insecure code] has been addressed by using a combination of cryptographic signatures and checksums. ” *Secure Installation Sources* [23]

For signatures a simple GPG solution is used. First of all, there is a file called products that lists the absolute paths to all products on a media. This file is the first that needs to be signed and the public key to check the signature has to be stored in products.key. This is secure since with asymmetric cryptography, a malicious file could only be crafted if the private key was available. The signature itself is located inside the file products.asc. If there is no products file inside the repository, the contents file is examined. This must exist for each individual product and contain a SHA1 checksum for each package inside the repository. Finally, each individual package may optionally be signed as well. So, there is a chain of trust from the initial key to the package level. This ensures that any file inside a repository is exactly what the end-user thinks it is. To sum it up, we have a very simple solution to ensure both, that a file is not corrupted by creating checksums and to ensure authenticity through standardized public key cryptography.[23]

Debian - DEB

Debian has decided to go the same way as OpenSuSE did, when it comes to signing of repositories and packages. The main difference is, that there is no option to also pass checksums as OpenSuSE provides. On Debian the necessary steps also involve the creation of a GPG key pair using the standard GPG tool. Then, the dpkg-sig tool is used to create a signed deb file. Once this is done for the packages, the reprepro tool is configured to know the correct keys and the public keys must be made available so anyone who uses the repository can verify the packages. From *Setting up signed Apt Repository with Reprepro* [24]

Special Considerations Concerning the Android Platform

Why Eve and Mallory love Android: An analysis of Android SSL (in) security [25] sheds light on the various failures for implementing secure transports on Android. First of all, the implementation of TLS and SSL classes on Android is fairly complex for many developers. Some developers simply opt for trusting any certificate, which is better than going completely plain text for transmission but doesn't provide any security form a MitM attack. Some allow the connection to be made with any hostname. This may lead to strange situations like an anti-virus app, that recommends deleting itself after a counterfeit signature database was injected. Furthermore, failures in secure communication aren't properly forwarded to the user interface. The end user usually only sees, that some error occurred, and no connection could be made but isn't informed that something malicious might be going on.

Attacks Against Common Package Management Solutions

In *Package management security* [26] Cappos et al. examine several possible attack vectors against popular open source package management solutions: APT and YUM.

Their three main principles are:

1. Don't trust the repository.
2. The trusted entity with the most information should be the one who signs.
3. Don't install untrusted packages.

They identify community repositories as a threat to the security of the operating system, since any attacker could just set up their own repository and spoof a benign one. Doing so is indeed rather easy, if they set up a proper chain of keys, since repositories are usually simply based on a web server that provides a specified set of files in a specific data structure. This may be mitigated by signing the repositories properly, but the key needs to be imported once, which leads to a possible attack scenario, when a new repository is added. Even worse are attacks against mirrors of main repositories, in case the keys are modified, which are implicitly trusted. If an attacker manages to control such a mirror, they may very easily spread malware that way, or perform other attacks.

Keeping that in mind, a wide array of attacks is possible. These include slowing down the package update process to a near halt, by artificially providing very slow download speeds, this concerns both YUM and APT. Also, both package managers are susceptible to a repository index file of unlimited size, so simply transmitting endless gibberish is entirely possible. Still, there are even simpler attacks: If a repository is taken over by an attacker, they could start providing any package they please inside their own repository, both APT and YUM are susceptible to this attack, depending on the execution. If their package version is newer than the one in any benign repository, the package manager might simply accept the malicious version. This would be particularly effective if a common package like the kernel is infected. An even sneakier attack would be to provide false dependency requirements thus leading the victim to install a malicious package, or at least fill the hard drive with downloaded packages since signatures are only checked after download. One could also create a package that provides everything and thereby make sure it will be installed eventually.

Stemming from the results of [26], several security principles that may be applied in general are proposed:

1. Validate repository communication. This can be achieved through checks of file sizes and data rates.

2. Track signature times and refuse old metadata.

3. Use HTTPS. If used properly, this significantly reduces the chance of man-in-the middle attacks.

4. Guard mirrors. This significantly lowers the chances of impersonation.

5. Sign metadata and packages.

6. Check whether metadata is correct.

For this examination we do not inspect how package structures are kept, or dependencies are resolved, as neither of these are a concern of the Ambient Dynamix project yet. Rather we will examine which mechanisms play a role to ensure authenticity, especially looking at the attack scenarios listed above. In order to write a new secure package management solution, it seemed like a good idea to examine how large projects ensure authenticity of packages. Two of the major solutions from large Linux distributions were picked. The one is the RPM system on OpenSuSE and the other one the deb package manager as it is used on Debian and the very popular Ubuntu Linux distributions. Both package managers also find use in the enterprise world: RPM as part of the SuSE Enterprise solutions or as part of RedHat and Debian is a very stable solution for web servers too.

3.1.7 Security Considerations for User Interfaces

While systems can be implemented with security in mind, a bad user interface for interacting with security relevant steps during program execution, may severely threaten the security of the whole solution. In *Humans in the loop: Human-computer interaction and security* [27], several key factors of keeping ”humans in the loop” of security are discussed. They work out the general difficulty, that security concepts are often perceived as burdensome by younger computer users and that often times the complexity of security configurations, causes errors. So, there is room for improvement with the user interaction. Though, it can’t all be blamed on bad user interface designers but also on the underlying concepts of security on modern computer systems. Most of them either come from the world of mathematics or have a military background, neither of which are associated with ease of use. So the question is left, if it isn’t worth to rethink some security concepts to better adapt to the needs of real humans instead of mathematical constructs.

Dourish et al. have conducted a series of interviews at an academic institution as well as an industrial research lab to work out several security related concerns in devices in *Security in the wild: user strategies for managing security as an everyday, practical problem* [28]. They focused on three main categories:

1. Attitudes towards security
2. Expectations of security as a barrier
3. Security as a whole: offline and online

Attitudes Towards Security

In the first category, they concluded, that several negative feelings are associated with security. First of all, frustration, which is based on the perceived hindrance of security for use of technology and an obstacle to the completion of work.

Then there is a certain level of pragmatism, i.e. using a computer full of malware only for a certain task where it wouldn’t matter. Therefore a certain translucency must be given, so the decision of security trade-offs can be made.

Finally, a feeling of futility arose when connected to security. This means, that users felt, that the ”bad guys” are always one step ahead and therefore it doesn’t even make sense to try and stop them.

Expectations of Security as a Barrier

For the second category, it turned out, that people tend to think of the solution to a security problem as an obstacle in their daily life. I.e. they made firewalls responsible for keeping hackers out, but also for lost emails. This observation leads to an interesting field of tension, because it turns out, that several views may form towards solutions. For example, if a solution to a single problem is put in place, i.e. only a virus scanner, it might be perceived as not enough, or as a reason to stop worrying at all, since it’s expected to solve all problems. Another problem is, that securing one aspect, i.e. stopping hacker attacks through firewalls, diverts attention from using a completely insecure means of connection like unencrypted Wi-Fi.

Security as a Whole: Offline and Online

Finally, the category about security as a whole showed, that many users made a strong connection, that actions taken online, i.e. releasing personal information, could lead to severe consequences offline, i.e. being stalked. The other finding here is, that the mixture between virtual and real network equipment easily confuses people. For example network related printer problems, that arise due to a connection to the wrong Wi-Fi network.

The research shows, that all these perceptions lead to a state, where users try to delegate ”security” to someone, like a good friend, or something, like a firewall, and try to stop worrying about it. Furthermore, a sort of security by obscurity arises, by, for example, making cryptic email responses that aim to only make sense in context. So the researchers have come to the conclusion, that security is ultimately an end-user problem, which can be eased by deployment of secure environments but the final decisions to stay secure need to be made by the end-user.

User Interface Design for Mobile Devices

Software Engineering Issues for Mobile Application Development [29] gives a good overview of some important design related questions concerning development for mobile devices. One very important conclusion they have drawn concerning security is that it is hard to attack the embedded software itself on mobile devices. However, it is possible to install software that significantly alters the expected behavior. [29]

Oberheide and Jahanian have aimed at working out the key differences of security concerns between mobile devices and stationary devices in *When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments* [30]. The main problem, they discovered, is the constraints on mobile device resources. This leads to a situation where traditional means to detect malware cannot be employed, because they are simply too resource intense. Due to the increased difficulties in spread, the likelihood of mobile botnets is way smaller than classic botnets.

While the ARM architecture offers features like trust zones and NX flags, techniques like ASLR, that require large amounts of RAM to really make sense, fail. When it comes to security on the platform itself though, quite different and maybe even better paradigms, than on desktops, are in place. Application delivery is secured by user interaction, if an application shall really be installed. Trust levels separate the amount of privileges a single application has on the system. Finally, mobile platforms have vastly improved system isolation techniques in place, that ensure that one application can’t interfere with another application easily. So, they conclude that certain paradigm shifts are necessary when developing for mobile devices, in particular, to accomodate the weaker platforms missing capabilities in comparision to desktop computers and lots of forward thinking about the ever emerging technologies is important.

Finally, in *The privacy badge: a privacy-awareness user interface for small devices* [31] a solution is proposed to give users a feeling of how securely they are behaving in a visual fashion for mobile devices. They do so by displaying a little badge, that shows various privacy leaks and how ”close” they are to the user, meaning how tightly they allow someone to gain information about them. This is particularly helpful to give users a better feeling for the impact of their decisions, which applications they allow to run on their devices.

3.2 Analysis

Following is a critical reflection on the work that has been done so far.

3.2.1 Android Security

The security precautions of the Android platform have been studied very well so far, as shown in: *Android Security Permissions – Can We Trust Them?* [32] which examined whether the Android permission system can be trusted. Orthacker et al. took a close look at a problem that Ambient Dynamix is facing too, which is the spreading off permissions to hide unwanted behavior. Their example is an application that provides GPS information and another one, that offers functionality to post to Twitter. With their combined permissions needs and some Inter-Process communication, it is possible to effectively create a back door by having the two apps run services in the background, that send the GPS information back to a server.

A great introduction to the structure of the Android operating system and the security concept is presented in *Understanding Android Security* [33]. The structure of an Android application is studied in detail as well as the way Android enforces permissions for applications. Furthermore Enck, Ongtang, and McDaniel explain how to secure an application against possible interference with other applications.

In *An Android Security Case Study with Bauhaus* [34] a solid inspection of Android itself has been executed and several inconsistencies between the documentation and system at that time have been revealed.

The system is built on pretty stable ground, meaning that there is a solid background structure due to the Linux kernel it is built upon as well as the user separation concept for apps. Yet the interaction with the end-user proves to be problematic. Especially the all-or-nothing approach when asking for permissions per app leads to a situation in which users prefer to just agree to any condition rather than thinking about what the single permissions actually imply. Also, they are forced into accepting whatever permissions they are asked for, otherwise they could not install the application they want to use.

It is also problematic, that developers sometimes seem to offer a lack of hygiene when it comes to asking for permissions. Some code for testing may have been left in the release .apk file that makes them ask for more permissions than they actually need to run the application, thus leading to further confusion for the end user.

The second source of confusion is the inclusion of advertisements in an application, as the ad framework may require way more permissions than the main function of the application really needs.

What remains a large problem, is the handling of Inter-Process communication. Due to the message oriented nature, many kinds of attacks are possible, that either try to interrupt the function of an application, or abuse the system to gain a wide range of permissions.

3.2.2 OSGi Security

Concerning OSGi, several attempts have been made to improve the security of package deployment. However they are not tightly integrated into the existing solutions yet. So, while there is a theoretical standard, there is no practical implementation of the standard.

When it comes to monitoring the behavior of packages after installation, OSGi based solutions rely on completely unrelated technologies like aspect oriented programming. Therefore an external solution needs to be found in any case, where untrusted code is combined with trusted code into a mixed environment.

3.2.3 Bytecode Manipulation

Bytecode manipulation directly on the device is a rather cumbersome task for the ARM based mobile processors, with run-times of several minutes for relatively small .apk files, as shown in *Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation* [22] below Bartel et al. gave detailed metrics for two devices as shown in the table 3.1. One being a smartphone with a 1.2GHz dual-core processor and 768MB of RAM and the other a tablet with a 1.4Ghz quad-core processor and 1GB of RAM. The following results were obtained by running the algorithm on 130 of the top 500 Android applications at the time of writing. The average size of the applications was 614KB bytecode.

Device	Min. Time (in seconds)	Median Time (in seconds)	Max. Time (in seconds)
smartphone	2.63	120.67	952.64
tablet	2.26	47.52	352.8

Table 3.1: Run-times of Dalvik bytecode manipulation as shown in [22]

The numbers above hold for Dalvik bytecode sizes up to 4000kB.

While it allows for various modifications, including the removal of ad-frameworks or unwanted behavior, on existing binaries, it is a method hardly feasible for technically non sophisticated end-users. But also people with knowledge about the inner workings of common Android apps might refrain from this concept, since it requires one to know exactly what they are doing and to study each application on its own. Due to the complete rebuilding of the .apk file, the signature needs to be recreated, thus requiring the application to be re-installed. This also means, that any update functionality that would otherwise be done via the Google Play Store is completely disabled, since suddenly the signers of the .apk files do not match anymore. In the end, the user would need to manually download each new version of an application, apply the modifications and then install it. So, while this would fix the flaw of having to grant any application all the permissions it requests, it forces the end-user to consider many things that would otherwise just run automatically to keep their software updated and patched in each iteration. This probably leads to a situation where they would rather agree to any permission asked for, rather than removing it.

3.2.4 Lessons Learned from Linux Package Management Solutions

Distributions of the popular Linux operating system have chosen to implement a GPG based solution. As it is an open standard, it can presumably be trusted, also it is known as a versatile standard for encryption and signing of data, with library support for Java, for example Bouncy Castle. [35]

The documentation found in the respective wikis of the openSuSE and Debian lacks details on which algorithms and key strengths are recommended though.

To give a short summary of a commercial approach: Microsoft decided to use digital certificates, like the ones from digicert. Which require the developer to buy a certificate from a trusted root authority that allows to assert their authenticity afterwards. [36]

The big advantage to the GPG approach is, that it is free for anyone to use, but in case of a total compromise of the server, it is equally easy to just replace the whole repository with a malicious one. At least, if a person used the repository for the first time, they would very likely accept the spoofed certificate and download the malicious packages. This situation cannot be circumvented with a public key server, since the authenticity of the public certificate builds the base of the trust in this scenario. There is no other authority that could prove a certain certificate belonging to a repository at a certain location and another one not. While this problem is circumvented in Linux distributions by shipping the installation media with the keys of the base repositories, the approach already fails, whenever any kind of community or self-created repository is added to the installation. Obviously, at this point, the user has to trust whoever created the repository to provide them with benign packages that actually only contain what they promise to.

These results clearly speaks for using a commercial vendor to get a certificate, since there are hardly any situations where such certificates could be spoofed in a manner that no one would notice.

3.2.5 Security from the User Standpoint

When considering the big picture of the articles examined in section 3.1.7, it becomes obvious, that end-user security is rather complex. Users tend to see security as a burden, because it gets into the way of getting work done on their devices. Furthermore depending on the measures taken, they give a false sense of security or lack of security. This is due to users not understanding, what a single security improvement means for the whole of their system.

A larger concern for mobile devices is their lack of resources to perform complex computing tasks, that are required for certain security improving operations. While the architectures often support common hardware security features like ASLR, they are also hindered by the limited nature of mobile devices. Approaches have been made to, at least, give users a sense of how securely they are behaving through visualization, but these don't actually work actively against mistakes that might be made.

To sum it up, despite many varying attempts to make security more convenient for users, it is always perceived as something burdensome that won't be good enough, since the bad guys are always one step ahead. So users aim to keep security concerns away from themselves and prefer to delegate it somehow, in doubt, by simply agreeing to whatever they are asked, just to make their software work.

3.3 Comparison and Summary of Existing Approaches

In this section we will compare some of the solutions discussed in the previous section with regard to the implementation in Ambient Dynamix.

3.3.1 Modifying Binaries on Android

The solution presented in *Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation* [22] was the only attempt for in-vivo bytecode manipulation found. The main problem lies in the resource intense task of actually modifying the binary on the device, along with the inconveniences of the re-signing process. Even the most security aware end-user would probably get unnerved by the long waiting time before the modification is complete. And even then, it remains to be seen, whether the rewritten application is actually still fully functional. So, while the solution offered a good pointer in the sense, that Android byte-code analysis is feasible on-device, it does not seem like a practical solution when working on end-user devices. The idea presented by the authors could be very interesting if it was run as a web service, where the devices could submit .apk files and get the modified binaries back.

3.3.2 Closing Thoughts on Package Management Solutions

On the open source side, there is only one approach established, which is to use GPG for signing and distributing the public keys along with the files. Commercial vendors rely on root certificate authorities to get trustworthy certificates and also have a base of trust without the need of keeping a repository but rather being able to distribute installation packages anywhere or having others redistribute them. Still, as we've seen in section 3.1.6 a secure structure for verifying repositories is only half of the rent, since a sled of other attacks is possible, if the package manager is not prepared for them.

In general, public key cryptography is the only viable solution to verifying the rightfulness of repositories and the packages contained inside.

CHAPTER 4

Methodology

4.1 Design Methods

For the design of the solution, code reuse was always kept in mind. So basically, it was chosen not to simply merge the code directly into the Ambient Dynamix framework, but rather to create libraries that offer functionality a little beyond what was strictly required for the solution. This does not impact performance, since care was taken that the method calls could be performed efficiently without causing additional overhead.

Throughout this work, there is a strong focus on keeping efficiency of operation. Therefore, the code was kept minimal and as readable as possible. Constructs to enable thread safety, for example, were left out in favor of performance, when they were not needed.

Since the code, that was developed for the solution, is meant to improve safety of operation inside Ambient Dynamix, any states that could cause the system to become unstable have been avoided as good as possible. This was tested with test cases as detailed in the subsections of the respective solution.

4.2 Working Environment

This work was created entirely on an openSuSE 12.3 64-bit based machine. Following is a list of all the tools used in the creation and test of the solutions. Detailed version numbers are omitted, since several tools were updated during the course of the work.

- Text Editor: Kate (KDE 4.11)
- Java Version 7u40
- Python Version 2.7.3
- Scrapy Version 0.16.3

- Android Debugging Bridge
- GPG 2.0.19
- Development environment: Eclipse Helios
  - Android Developer Tools
  - Java Developer Tools

4.3 Analysis Methods

Since this work is strongly related to Android and to low-level concepts of the operating system, such as permission management and interaction between applications, approaches for solving the various problems were searched by understanding how Android would normally do things and then recreating them. Due to the fact, that Android and Java are closely related, it seemed like a good idea to start experiments out with Java code and then port them to be executable on Android. This idea only works for higher-level concepts though, as Android has some essential differences to Java, when it comes to security aspects. Another important aspect of the solutions was to find simple solutions, especially with regard to ease of computation and run-time performance. If resources were there in abundance, some more complex approaches could be chosen, but since everything should be running on a smartphone, the idea was to aim for 'good enough' solutions that are simple to perform while providing a high level of security.

4.4 Test Setup

To check the correctness and the performance of the solutions, four Android devices of different kinds were used.

The first device was a Google Nexus 4, updated to Android 4.3 on an AOSP build of the latest version with the build number JSS15J. The processor is a Qualcomm Snapdragon S4 Pro APQ8064 running at 1.5 GHz maximum and 2 GB RAM. This device's merit for testing is a completely unmodified version of Android and its role as a signature device for the Android operating system.

The second device is an Asus TF700T, also known as the Asus Transformer Infinity, running a test build of CyanogenMod 10.2, which means it is basically also running Android 4.3. The processor is a Nvidia Tegra 3 T33 running at 1.6 GHz maximum and 1 GB RAM. This device was chosen to represent a large tablet with a different popular firmware.

The third device is a Samsung Galaxy S3 (i9300), running a test build of CyanogenMod 10.2. The processor is an Exynos 4412 running at 1.4 GHz maximum and 1 GB RAM. This device was chosen to represent another smartphone that is fairly new and popular.

The fourth device is a Google Nexus 7 (tilapia/2012), running the factory image with build number JWR66Y, also Android 4.3. The processor is a Nvidia Tegra 3 T30L running at 1.2 GHz maximum and 1 GB RAM. This device was chosen to represent the popular 7 inch tablet format.

All devices are listed in table 4.1.

For the tests, the following files were used:

Device	Processor	Clock Speed (GHz)	RAM (GB)	OS Version
Google Nexus 4	Qualcomm Snapdragon S4 Pro APQ8064	1.5	2	4.3 Build JSS15J
Asus TF700T	Nvidia Tegra 3 T33	1.6	1	CM 10.2
Samsung Galaxy S3	Exynos 4412	1.4	1	CM 10.2
Google Nexus 7	Nvidia Tegra 3 T30L	1.2	1	4.3 Build JWR66Y

Table 4.1: Device Specifications

1. org.ambientdynamix.contextplugins.barcode_1.0.0.jar (474kB size) and

   org.ambientdynamix.contextplugins.zephyrhxm_1.0.0.jar (29kB size) from the official Ambient Dynamix repository at the newest version as of the time of this writing found in the repository [37]

2. A self-generated GPG keyring holding a 2048 bit RSA key

3. Detached signatures of the two jars mentioned in point one of this list, generated using the key GPG key, both 475 Bytes in size

4. A permission list, generated from the Android documentation as discussed in the solution section

To get a stable picture, the test files were copied to the internal storage of the device, so a slow micro SD card could not interfere with the measurements.

CHAPTER 5

Evaluation

In this chapter we will discuss the solutions for the various security goals posed by the Ambient Dynamix framework as well as the approaches that were evaluated before the final solution was found.

5.1 Enforcing Permissions on Calling Apps

As stated in section 1.1, the concept Ambient Dynamix introduces to Android, has never been imagined before. Therefore it needs solid workarounds to patch certain holes that arise in the permission security concept. Since Ambient Dynamix requests many permissions from the Android system, it needs to ensure that this power cannot be abused by other applications. These would not ask for any permissions, yet use plugins that make full use of the permissions Ambient Dynamix has. To that extend, it was necessary to find a way to introspect which permissions an application has requested. While this possibility is not directly obvious, Android indeed provides all the tools necessary to do so.

Ambient Dynamix is called by an Intent from another application. The action creates a Java object which, amongst other things, contains the unique id (UID) of the calling application. This is the key to strictly identify a given application. By acquiring the Android PackageManager from the global context, it is possible to gain all the important information of any application that is installed on the system. This allows to gather knowledge about all the permissions an application has requested. The procedure to do that is shown in listing 5.1.

Listing 5.1: Programmatically getting app permissions

```
// Construct a new application for the caller
ApplicationInfo info = null;
// get the PackageManager
PackageManager pm = context.getPackageManager();
// List packages by given id, should only be one
String[] packages = pm.getPackagesForUid(id);
```

```
PackageInfo pkgInfo = null;
try {
// get the ApplicationInfo for the chosen application
  info = pm.getApplicationInfo(packages[0], PackageManager.
     GET_UNINSTALLED_PACKAGES);
// Finally, grab only the PackageInfo we are really interested
    in
// We need to include the GET_PERMISSIONS flag to introspect
    app permissions
  pkgInfo = pm.getPackageInfo(packages[0], PackageManager.
     GET_PERMISSIONS);
} catch (NameNotFoundException e) {
  Log.e(TAG, "Count␣not␣get␣information␣for␣calling␣UID:␣" + e.
     getMessage());
}
return null;
```

Once the information has been gathered, the Ambient Dynamix framework can decide whether the application has acquired sufficient privileges from the system so it could do the operations, if they were part of the calling application's code.

5.1.1 Test Coverage

Since listing 5.1 does not bear any algorithmic difficulty when being run, there were no performance measurements taken. Yet, to assert the correct functionality, the source of the Dynamix Logger sample app, available at [4], was modified. Furthermore, the bar-code scanner plugin and the Wi-Fi scanner plugin were downloaded from the official repository and added to a local repository. That way, it was possible to declare arbitrary permissions for each. In the normal configuration, the bar-code scanner requires access to the camera, while the Wi-Fi scanner obviously needs access to the network configuration.

With the patch applied, the following use-cases could be staged and Ambient Dynamix successfully passed the tests:

- The calling application has all permissions necessary and more.
- The calling application has exactly the necessary permissions.
- An additional permissions, Bluetooth, was unnecessarily declared for the barcode scanner and the calling application did not hold it.
- The calling application lacked all the necessary permissions for the plugins but had others.
- The calling application had no permissions granted.

5.2 Asserting Permissions of Plugins

Before this work on Ambient Dynamix, there already existed a sort of permission system, that had little influence on what was actually happening. The main focus was to inform the user of the plugins expected bahvior. However, these were written by the developer, so for a barcode scanner, they would state that they needed permissions to access the Camera to take a photo of a barcode. But, if the developer was a bit shady, he could simply do whatever he pleased alongside. For example, they could send the image to some remote location without stating it needed network permissions. Since Ambient Dynamix is open by nature, it also aims to allow community repositories. Therefore, it is important to empower the user with security checks on their own device.

In the previous part of the work, it was ensured that a calling app could not escalate their privileges through Ambient Dynamix. In this section of the work we will focus on ensuring that plugins only do, what they ask permissions for, as well as prevent certain calls altogether.

While a cloud based solution seems enticing, it would generate a lot of overhead to upload plugins to the evaluation service before they could be considered for installation on the device. So, while plugins in the official repository can be checked with the library suggested in the solution beforehand, it is necessary to also build this functionality into Ambient Dynamix itself so there is no additional delay before a plugin can be installed and executed.

5.2.1 Mapping Classes and Methods to Permissions

The Android developer documentation [9] is readily available and features an easily parse-able set of HTML pages that contain permissions necessary to use a class or a given method of a class. A good starting point for generating the map is the package index, found in the Android developer documentation. Mapping out the necessary permissions boils down to two scenarios:

1. Permissions that are necessary to instantiate a class

2. Permissions that are necessary to use a certain method of a class

Usually, it is an either-or case. So, either a permission is necessary for the whole of the class, or one can instantiate it and only certain methods require special permissions. If a class needs a permission to be instantiated, it is stated in the introduction section of the respective class. Otherwise, if only a certain method requires a permission to be set, the necessary permission can be found in the detail section of the method.

So, the next step is to examine the page source for both cases. For this work Mozilla Firefox, with the Firebug extension [38] installed, was used. Firebug is very helpful, since it allows to select any DOM element of a given web site and get a neatly formatted view of the corresponding source.

Let us first examine the case, where we need a permission set to instantiate a class. For this example, we will examine the class android.Bluetooth.BluetoothSocket and find, that the class overview contains the following line: ”Note: Requires the BLUETOOTH permission.”. The bluetooth in all capitals is a link to the Manifest.permission class site. So, if we inspect the element, we can see in the page source

'<a href="/reference/android/Manifest.permission.html#BLUETOOTH"> BLUETOOTH</a>', which gives us all we need to know. With this line, we can clearly identify that it is about a permission, since only permissions are linked back to the corresponding site and we also know, that the necessary permission is "Bluetooth".

Now we need to find a way to differentiate between class and method permissions: Google was nice enough to make the documentation easy to parse. Again, utilizing Firebug, we can quickly distinguish that the div we are looking for must have the class "jd-descr" and have a header of type h2, that reads "Class Overview". So, for the second case, where we have a larger class, where only certain Methods require special permissions, a good example can be found in "android.content.Context". The class overview is pretty short and does not state any permissions to be necessary. However, there is a set of methods inside the class, that require various individual permissions: for example, the method "clearWallpaper" requires the permission "SET_WALLPAPER", or the method "getExternalCacheDir" requires the permission "WRITE_EXTERNAL_STORAGE". Obviously, it would be wrong to require these permissions if a developer used the class for some other Method. So, applying the same tactic as before, we can examine the page source again. For the case of the Methods, there is a nice table overview that lists all of them inside the class. And again there is a key sentence to look for: "This method requires the caller to hold the permission "SET_WALLPAPER". As before, the all capitals "SET_WALLPAPER" contains a link back to Manifest.Permission. In order to find the right ones, we again need to examine the DOM tree for the unique features and can see, that the table has the class "jd-linkcol" and the div that holds the actual information about the Method is called "jd-descrdiv".

A custom crawler, using Scrapy, a Python web-crawling framework, was designed to generate a map of classes and methods and the permissions that are required from the Android system in order to use them.

So far it seems that the pages in the documentation are pretty stable, so with new versions of Android coming, the parser should still work. The keyword to look for with XPath was a link to permissions.html which is the overview page that explains all the different predefined permissions of Android.

To use Scrapy, first an itemtype must be defined containing all the fields that are necessary, as shown in listing B.1. For this use-case it consists of three entries. This allows us to store the class with the optional method that requires some permission or several of them.

The places to look for are either the overview of the class, for class-wide permissions, or the detailed description of a given method for a single method or the tables inside the class descriptions. This work could be achieved making extensive use of XPath expressions, an example to extract permissions for a single method is given in the listing 5.2.

Listing 5.2: XPath expressions

```
# extract all divs that contain details on a method

//div[div[@class="jd-details-descr"]//p//code/a[contains(@href,
    "permission.html")]]

# extract the permission necessary to execute a method
```

```
div[@class="jd-details-descr"]//p//code/a[contains(@href,"
    permission.html")]/text()
```

Listing B.2 shows the whole Scrapy spider that was used to extract the permissions according to their methods or classes.

Certain methods may be called with different types of parameters but require the same permissions or the permissions a class might occur twice. To filter out these double results an additional pipeline was employed. The source code of which can be found in listing B.3. It remembers all the items it has seen so far and compares them with each new one, in case it has not yet been seen, it will be added to the final result, else it will be discarded.

Finally, the path to the spider as well as the pipeline must be reflected by the settings as shown in listing B.4.

Scrapy also allows to output the findings in the JSON format, which can be read in Java easily.

This solution allows for a quick and easy way to get a set of all the permissions needed by classes and methods. Once set up, it can be run whenever a new version of Android is released to automatically correct the permissions.

The permission map generated from the Android developer documentation is pretty sizeable with a total of 308 records of both Class and Method permissions combined. This comes down to a set of 277 unique Methods that require permissions and a total of 31 classes that require a certain set of permissions to be instantiated.

Format of the Permission Map

Since Java can natively handle JSON files, the choice to use this type of file was obvious. To make the distinction between method based and class based permissions easier, the package path was separated to end with the class name. An extra field for the method name was added. Each entry is completed by a list of necessary permissions, as of now, there are no methods or classes, that require more than one permission though. An example for a full entry is shown in listing 5.3.

Listing 5.3: JSON permission item

```
{"classname": "android.webkit.WebSettings", "methodname": "
   setBlockNetworkLoads", "permission": ["INTERNET"]}
```

In the currently hypothetical case, that there is more than one permission necessary, the entry would look was shown in listing 5.4.

Listing 5.4: JSON permission item

```
{"classname": "android.nonexistant.class", "methodname": "
   setSomethingNotExisting", "permission": ["INTERNET","
   BLUETOOTH"]}
```

5.2.2 Dangerous Methods and Classes

The guidelines for plugin development state, that a plugin is not allowed to store data on the device. So any output it generates must be returned to the caller, or be lost. Therefore the first set of problematic classes and methods are those in the java.io package, which must all be blocked. The second huge concern is reflection and the utilization of custom classloaders. Both of the structures are necessary under given circumstances on Android, but both can also be avoided:

Let's first discuss the concept of reflection shortly. Reflection under Android is the recommended way of asserting backwards compatibility for an application while allowing the use of newer features on newer versions of Android. An application can use reflection to check if a given class or method is available and if so, use it. Otherwise it can just continue with a baseline of functionality. The methods 'forName' as well as 'loadLibrary' are of concern in this case. Second, we have classloaders. In general, it should not be necessary for a program, the size of a plugin, to need this. However, Android only allows a certain number of classes to be loaded in one classloader. If one needs to go beyond that threshold, they need a second classloader. Again, it gets rather difficult to analyze the happenings inside the code than. Therefore, all types of classloaders shall be prevented from being used.

5.2.3 First Attempt: Debugging

Debugging seemed like a great idea for a start. Basically, the Android Debug Bridge (ADB) is based upon JDI, which is the standard Java debugging interface. Yet, it turned out not to be as easy as it seemed for everyday use. The work done by *Android SSL bypass* [39] (Homepage) project was a reasonably good starting point. They had created a solution using ADB to remove SSL from Android internet connections and sniff on the traffic. The beauty of their solution was that it illustrated the use of JDI in combination with ADB and it seemed applicable for a fully on-device solution. At the time of this attempt another application, that claimed to allow on-device debugging of other applications, was also available in the Google Play store. Both of these things combined seemed rather promising, that the solution could work out.

Once development on this approach was started, it turned out to be way less practical than originally thought. First of all, the mentioned app, had the requirement that wireless debugging has been activated on the device. Of course, no technically non-sophisticated user could be asked to do this step themselves. Yet hope remained, that when deploying an app with debugging turned on and running the debugger from the same .apk file in a second process, it would work. This assumption turned out to be wrong though.

Obviously, the concept of apps debugging each other, especially on a stock device, without root access or ADB turned on, was not a use-case for Android. So the idea had to be dropped. It remains though that for a scientific lab environment or for technically sophisticated users, such a solution could be very interesting to remove the need of a computer or laptop to debug and manipulate applications on the device.

5.2.4 Solution: Static Code Analysis

With the restrictions that are imposed on developers by the developer guidelines of Ambient Dynamix, that is, no writing of files and no use of reflection, static code analysis, seems like a feasible solution. The fact, that code for Android is compiled to a different bytecode than plain Java code, renders most supporting libraries and utilities useless. This holds especially for techniques like run-time weaving, where the bytecode would need to be dynamically modified. Due to the completely different virtual machine, Dalvik, on the Android operating system, libraries, that do such things, need to be completely rewritten. Fortunately, an Android version of the asm library, called Asmdex, had already been developed. It might not have the same feature set as asm does, which is mostly due to the way Android works differently from plain Java, but it allows to extensively browse all classes that are used inside an application. *Asmdex Homepage* [40]

Asmdex implements the visitor pattern and callback pattern. It iterates through all the classes in the dex file it is given and whenever it finds something interesting, calls the corresponding method of the subscriber. In this case, we are interested in all classes being visited, so we can later check, if there are class-wide permissions necessary and of course in all methods being visited. These so called Adapters work recursively. First of all the dex is scanned for all classes, then each adapter for a class calls the adapter to examine the methods.

A global list keeps track of each method and class uniquely. So, we can use the information we have gathered before to compare the permissions a plugin states it would need, to what it would actually need, if it was enforced by the Android system. Furthermore, we can also prevent any given set of methods or classes being used. This could also extend to the use of ad networks or anything else a plugin should not be using.

This method is also robust against obfuscation, since at some point, the API calls will have to be disclosed.

How the Solution is Executed

The task of verifying a plugin can be split into several steps, the most basic interaction is shown in figure 5.1. Looking at it in greater detail, first of all, the set of classes and methods as well as permissions need to be read. Then for any given plugin the following steps occur:

1. Once the plugin is downloaded, it is opened, using the jar utility.

2. The classes.dex file is extracted and stored in a temporary location.

3. The asmdex library is used to create a list of all classes and methods that are used in the plugin.

4. Finally the list is compared to the blacklist as well as to the permissions that are necessary and checked if it complies to the permissions it claims to need.

Let us examine the most important pieces of code to run the solution:

First, we need our main logic to get the classes.dex file and start extracting the classes and methods with asmdex. This is done by extracting the .dex file from the .jar file and writing it to

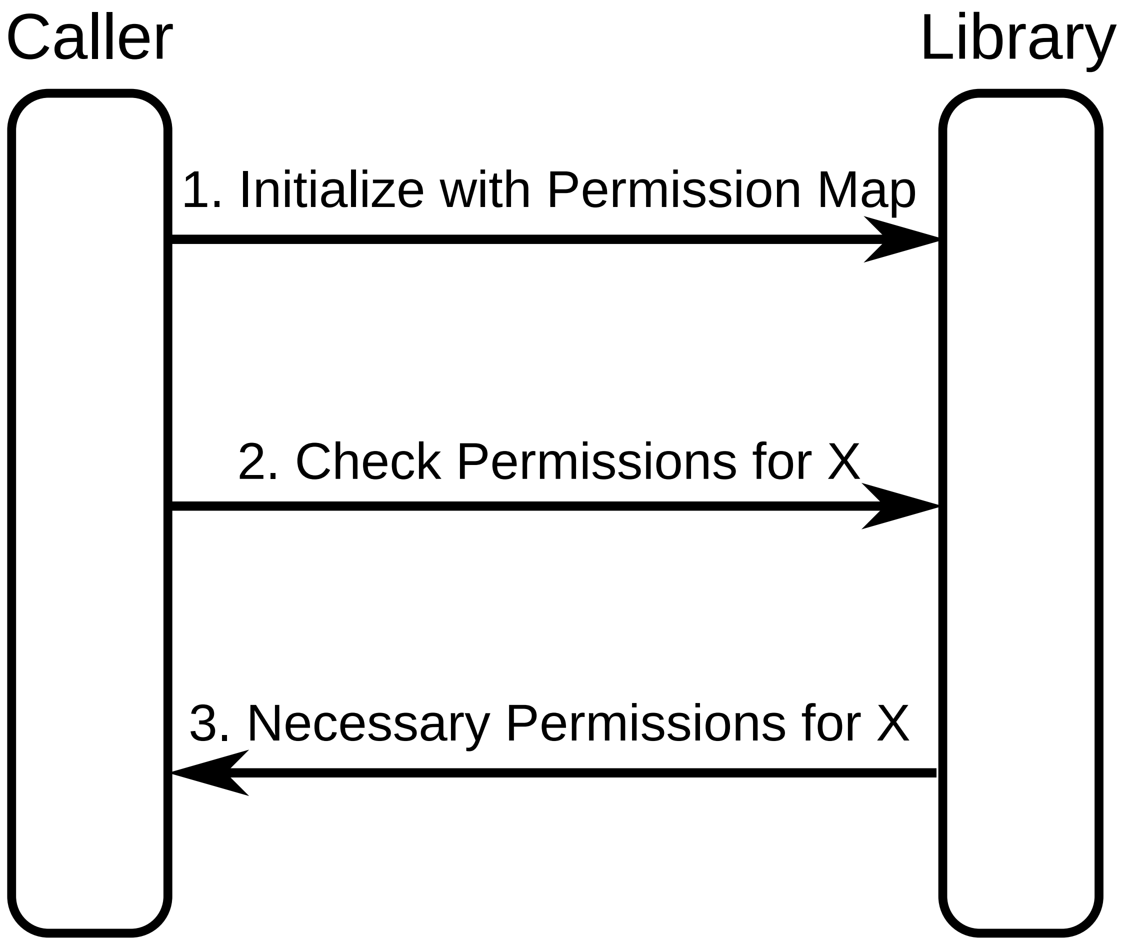
Caller
Library
1. Initialize with Permission Map
2. Check Permissions for X
3. Necessary Permissions for X

Figure 5.1: Interaction between a calling application and the library

a temporary location. We need to remember the necessary permissions in a singleton and finally start the scan. The implementation can be found in listing B.5. The singleton is needed, since asmdex uses the visitor pattern and simply returns single classes and methods depending on where it is inside the code. The most important method of the singleton is the one, that converts the output asmdex generates into something that is formatted in the typical dot separated fashion as shown in listing B.6.

The rest of the implementation is based closely on the examples shown on the website. Here is a short summary of the methods:

ApplicationAdapterAnnotateCalls is an instance of the abstract ApplicationVisitor, which is first called for each new application. It is also important to implement the visitEnd method, to signal when everything has been done for a given application as shown in listing B.7. This hands down the work to the ClassVisitor, called ClassAdapterAnnotateCalls, shown in listing B.8, which hands the work further down to the MethodVisitor, called MethodAdapterAnnotate-Calls shown in listing B.9, wherein the name of the class and the visited method is saved, since any class that does not contain any methods would not be of importance for the check. The addPackage method is the one shown in listing B.6.

In a recent version there was a change to the library, which broke parts of the samples that were available at the time of writing this solution. Most importantly, the so called 'EmptyVisitor' was missing, so it had to be re-implemented like shown in listing B.10.

It remains to compare the classes and methods that were found with the ones in the list, that was generated from the documentation as detailed in section 5.2.1. This way, it can be determined, which permissions are necessary for a given plugin to be executed.

Impact of Changes in the Permission System in Different Versions of Android

So far, the permission system of Android is fully backwards compatible. This means only completely new permissions, for hardware that was not supported in former version of the Android operating system, like NFC, or finer grain permissions like the split up between read and write permissions for external storage were added. Therefore, the new permissions can simply be adopted into Ambient Dynamix the same way they are woven into the existing Android permission system. In order to keep backwards compatibility for older applications, it might be beneficial to forbid the use of newer permissions, like the permission to read external storage, added in API level 16, which does not exist in versions API levels. [9]

For now, it would suffice to stick to permissions that were available in Android as of version 2.x and only use newer permissions, if they are backwards compatible or not relevant for former versions of Android. When new permissions are added, that are neither hardware dependant nor sub-permissions of already existing permissions, it might be better to make Ambient Dynamix aware of the version of Android it is running on and enforce permissions based on the API level as well.

Test Coverage

In order to assert the correctness of the solution, it was tested with the barcode scanner plugin and the zephyrhxm plugin from the official Ambient Dynamix repository. [41]

The following two questions were important:

1. Are all necessary permissions detected as expected?

2. Do the detected permissions match with what was manually declared before?

For the tested plugins, both questions could be verified as correct.

Performance

To assess the performance of the solution, four test runs were made on each device. For each run, the time to load the permission table and then to analyze the two plugins were taken. The table below shows the averages of the four test runs for each device:

The results in table 5.1 show, that while the solution is too slow to be executed on every single plugin, it sure serves the purpose of securing the environment from untrusted plugins. It

Device	Prepare permission map	zephyrhxm	barcode
Nexus 4	159.25	1912.25	25836.5
TF700T	168.5	1434.75	19825.25
i9300	65.75	1091.25	13720.25
Nexus 7	143.25	1717	21745.5

Table 5.1: Average of 4 test runs for permission checks (times in ms)

should also be noted, that the barcode plugin is on the large side of plugins found in the official repository at [37], where the median of the plugin sizes is at 39kB. So it should be expected to have results closer to the run-time of the zephyrhxm plugin rather than the barcode plugin.

Interfaces and Exception Handling

Listing 5.5: Execution of the Permission checking library

```
PermissionChecker c = new PermissionChecker(
    AndroidPermissionMap.json, additionalPermissions.json);
ArrayList<String> result = c.processJar("/Location/of/plugin.
    jar");
```

In order to use this permission checking library, it is necessary to first instantiate it with a permission map, as shown in listing 5.5. The library expects at least the large permission map, like it was generated in section 5.2.1. Optionally, a second permission map may be added for custom checks. In case errors are encountered during the parsing of the permission maps, an IOException or a ParseException may be thrown.

When it is loaded once, it may be used on as many plugins as necessary. The call to the permission checking interface, ”processJar”, is blocking and shall only be used in a single threaded fashion. The argument expected is the path to any .jar file or .apk file, that contains a ”classes.dex” file. In case of errors during the extraction of the classes.dex file, an IOException may be thrown.

Since the library is written entirely in Java, it may just as easily be used in a desktop or server application.

5.2.5 Other Ideas

The following ideas were taken into consideration but turned out not to be viable solutions to the problem, before any development was started. Therefore they are just quickly explained to illustrate how it does not work:

One idea had been to make use of aspect oriented programming, which would have been nice to use, if libraries that allowed run-time weaving were available. However, since Android bytecode is not the same as Java, libraries to do that are lacking. Compile-time weaving is possible but it does not help with pre-compiled plugins, that do not comply to the standards. With run-time weaving, the idea would have been to insert pointcuts on dangerous method calls as well as to methods that require permissions and check during run-time if a plugin is well behaved.

On Android, a developer support functionality, called StrictMode, exists. This setting is meant to allow for quick checking if code is well-behaved. For example, it allows to monitor for writes to the storage. While this is another great solution, when a developer is in full control of their code, it does not help any for a program like Ambient Dynamix, since it can be programmatically turned on and off at any given time. So a malicious plugin would disable StrictMode as the first order of business and no one would ever notice anything it did.

5.3 Authenticating Repositories and Packages

Ambient Dynamix currently is in the early phase. While the software itself is functional, there is lots of room for many more plugins to be written and to be published in various repositories besides the main repository. Besides a growing overhead when more people start developing for Ambient Dynamix to check the sources of plugins and add them to the main repository, there might be cases, where it would not make any sense. For example, a company could base an on-premise solution around Ambient Dynamix and neither want to disclose the sources nor see any use in adding it for the public. Besides the need to manage repositories, there is also the need to verify which software came from whom and to ensure its authenticity as well as decide on how trustworthy it is. For example, a plugin that made it to the main repository should always be trusted and not put through the static code analysis discussed in section 5.2.4.

As of now, the main repository is hosted on a regular web server, without SSL enabled. This means, it could be spoofed easily with a standard man-in-the-middle attack and any plugin could be served instead of the proper one.

In order to save ourselves the trouble of redoing checks for every plugin installed, it is important to properly authenticate the main repository. This way, we will not have to check the permissions of each plugin downloaded from the main repository.

1. Secure the repository with a valid SSL certificate and allow access only through an encrypted connection. This is a first important step against man-in-the-middle attacks.

2. Sign the repository index file with a pre-distributed public key, to ensure authenticity of the repository.

3. Sign each individual package to ensure the download of the file is actually, what is referenced inside the index file.

5.3.1 What Android Provides

Both Java and Android provide ways to sign jars and .apk files respectively. On Android it is even a requirement for .apk files to be digitally signed in order to execute them. While all this is great for standard solutions, it does not help too much in the case of Ambient Dynamix. The .apk file signature is only checked, if an application is installed and ran by Android. Since the plugins are dynamically executed from within Ambient Dynamix, this protection mechanism is bypassed. And even if it was not it does not fully meet the criteria of both authenticating the package but also making sure that it came from someone the software trusts.

Very recently it has been found though, that there is a possibility to craft modified .apk files, that will pass the Android signature check while running modified code. The details can be found in *Uncovering Android Master Key That Makes 99% of Devices Vulnerable* [42]

5.3.2 Public Key Infrastructure

When it comes to securely signing packages, asymmetric cryptography is the way to go. There are open standards readily available, like the chosen PGP standard and libraries to work with them. The solution is based on the Java cryptography library BouncyCastle. Using BouncyCastle it is easy to programmatically generate PGP keys, sign files and verify their signature on the client. Bouncy Castle [35]

So the solution is fairly simple in the end. Per repository a private/public key pair has to be generated, using the DSA algorithm. The public key of which will be distributed with Ambient Dynamix, the private key stays with the rightful owner of the repository, or it can be removed in case of theft or loss of trust in the key holder. With the private key, the repository owner signs the index file. This gives Ambient Dynamix the location of the plugins as well as their required permissions and further information like a description of what it does and the location of the signature. Ambient Dynamix checks the index file's signature and, only if it matches, progresses with adding the repository.

When a user decides to download a plugin, the first thing, after download is completed, will be to check the plugin's signature. Only if the signature that was indicated by the repository index file matches, it will allow the plugin to be installed and run. The decision to create detached signatures has been made to save the file operations needed to get from the signed files to usable indices and plugin bundles. It has also been of importance to keep the solution of a general purpose nature, so no specifics for any kind of file were used, instead it allows to sign any data type that might come along.

Implementation

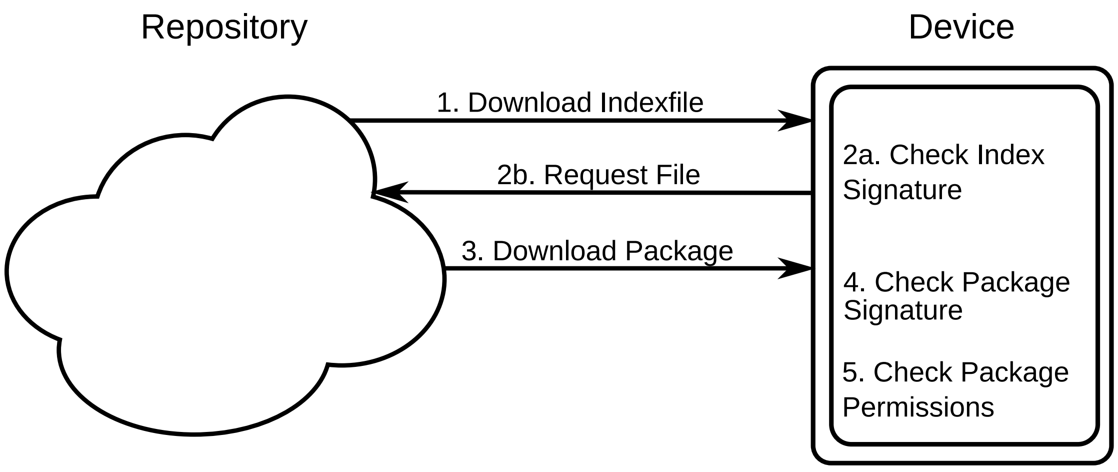
Repository
Device
1. Download Indexfile
2b. Request File
3. Download Package
2a. Check Index Signature
4. Check Package Signature
5. Check Package Permissions

Figure 5.2: Package download process with cryptographic checks

Figure 5.2 shows the standard download process for any package. Of course, the process may halt in case any abnormalities are found during the cryptographic check of a file.

The main complexity of the implementation for this library lay within the use of the available cryptographic library, Bouncy Castle. While the steps to encrypt data from the command line are obvious, it is not that easy to be achieved programmatically. The code of the final implementation was inspired by the Apache Commons OpenGPG project, but had to be improved due to lack of certain functionality.

Before working with the cryptographic library, the system must be made aware of it.

This can be accomplished with the following line of code shown in listing 5.6.

Listing 5.6: Add BouncyCastle as Security Provider

```
Security.addProvider(new BouncyCastleProvider());
```

To work with the keys, a structure to keep PGP keyrings stored is necessary. The code in listing B.11 shows how to do this. Getters and Setters were omitted.

Next we need to import public and private keyring files for use, the code to do this is shown in listing B.12. Once this is done, signature verification can begin. Based on the Apache code, a streaming signature verifier was wrapped to be used with a regular file input, note that we are only verifying detached signatures in this case.

The real work is done inside the streaming signature verifier. This class is first given the signature of the key to check against, as well as the keyring that was created before. Afterwards it is fed with the bytes of the file before the signature check is handed off to the BouncyCastle cryptographic library as shown in listing B.13. This concludes the process of verifying a detached PGP signature of any given file. Based upon similar code, it is also possible to create signatures programmatically.

Test Coverage

In order to assert the correct detection of wrong packages, we need to cover all the possibilities of things going wrong. The following questions were identified:

1. Are matching packages and signatures detected correctly?
2. Is an error shown, when the signature was tampered with, while the package was valid?
3. Is there an error, when the package file was tampered with and the signature was the one of the correct file?
4. Does an error appear, when a package is checked against the signature of another file?

All of the above questions could be answered correctly by the software, again using the two packages, barcode scanner and zephyrhxm from the official Ambient Dynamix repository.

Performance

To assess the performance of the solution, four test runs have been conducted on each device. For each test run the conditions mentioned in the listing in section 5.3.2 were checked, as well as the time it took to prepare the GPG public keyring for use. Table 5.2 shows the averages of the four test runs for each device:

Device	read keyring	barcode correct	zephyr correct	barcode bad-GPG	barcode bad-jar	zeyphr with barcode GPG
Nexus 4	107.75	112.75	59.25	56	82.5	61.25
TF700T	57	137	62.75	64	82.75	75
i9300	67	67.25	47	43	66.75	44.5
Nexus 7	108.5	115.5	85.25	69.75	88	72.5

Table 5.2: Average of 4 test runs for GPG signature checks (all times in ms)

Since the keyring only needs to be loaded once before any checks occur, this solution is clearly feasible to be run during the usual installation process of a plugin.

Interfaces and Exception Handling

Listing 5.7: Execution of the Signature Authentication Library

```
Security.addProvider(new BouncyCastleProvider());
BouncyCastleKeyRing ring = new BouncyCastleKeyRing();
char pass[] = {'t', 'e', 's', 't'};

ring.addSecretKeyRing(new FileInputStream("dummy.skr"), pass);
ring.addPublicKeyRing(new FileInputStream("dummy.pkr"));
```

```
BouncyCastleOpenPgpSigner sign = new BouncyCastleOpenPgpSigner
    ();

sign.detachedSign(new FileInputStream("barcode.jar"), new
    FileOutputStream("barcode.gpg"), "0C95F1DC1CD395A9", ring,
    true);

BouncyCastleOpenPgpSignatureVerifier verifier = new
    BouncyCastleOpenPgpSignatureVerifier();

SignatureStatus s =verifier.verifyDetachedSignature(
     new FileInputStream("barcode.jar"),  // binary input file
     new FileInputStream("barcode.gpg"),
                                                   // inputstream
        for the signature
     ring );

System.out.println(s.isValid());
```

As shown in listing 5.7 the signature authentication library is capable of both generating signatures as well as verifying them. Let us go through the steps in detail. Like mentioned before, it is important, at least on desktop Java, to add BouncyCastle as a securityprovider.

A new BouncyCastleKeyRing needs to be instantiated, before it is filled with secret and public keyrings. For the secret keyring a password must be supplied as a character array as well as the keyring as a FileInputStream. In case of an error, an IOException or GPGException may be thrown.

When the keyring is prepared, it is time to instantiate the BouncyCastleOpenPgpSigner. It contains a method ”detachedSign”, which accepts a FileInputStream for the file to be signed, a FileOutputStream for the signaturefile, the key fingerprint of the private key to be used and the keyring it shall search the private key in. The final boolean parameter allows to decide whether to asciiArmor the output or not. In case of an error, an IOException or GPGException may be thrown.

In the next step, we will examine the verification process of a file and its signature. First, the BouncyCastleOpenPgpSignatureVerifier must be instantiated. The method verifyDetachedSignature allows starting the verification process, the output of which is saved into the SignatureStatus class. Again, it requires a FileInputStream of the file to be checked, the signature to check it against and finally a BouncyCastleKeyRing, that holds the public key to verify the signature. In case of an error, an IOException, UnknownKeyException or GPGException may be thrown.

The results may be obtained by calling the isValid method of the instance of SignatureStatus.

5.4 Orchestration

Before we examine how the improvements benefit the use of Ambient Dynamix on a device, let us understand how Ambient Dynamix handles plugin usage before the security fixes:

We will show the example by the installation process for the barcode scanner plugin:

1. An application requests the barcode scanner plugin.
2. Ambient Dynamix connects to the repository to get the package index.
3. The barcode scanner plugin is obtained.
4. The barcode scanner plugin is installed.
5. The application may use the plugin anytime.

Now, that we have gathered three elements to make the use of Ambient Dynamix more secure, let us get an overview of how the solutions interact with Ambient Dynamix. Currently, these libraries are to be implemented into Ambient Dynamix itself, therefore a flow, that excludes possible user interaction is given:

Let us assume, we want to download a plugin from the official Ambient Dynamix repository, let us say the barcode scanner:

1. An application needs the barcode scanner plugin.
2. Ambient Dynamix connects to the repository to check for plugin availability.
3. When the repository index is obtained, its signature is checked against the shipped GPG keys.
    a) The signature check fails, we stop execution of the process.
    b) The signature check succeeds, we move on to the next step.
4. Ambient Dynamix checks whether the calling application has sufficient permissions to use the barcode scanner.
    a) If the permissions are insufficient, we stop here.
    b) If the permissions suffice, we move on.
5. Since the barcode scanner comes from the official repository, we simply download it and check the signature.
    a) The signature check fails, we do not install the plugin.
    b) The signature check succeeds, we continue with the next step.
6. The barcode scanner is installed.
7. Whenever an application wants to make us of the plugin, it first undergoes the same check as before whether it has sufficient permissions to use the barcode scanner, since permissions might have changed during an application update.

Now, we will examine the case, when a community plugin is downloaded. Let us presume it is a kind of location dependent weather plugin:

1. The community repository is added by the user.
    a) If a wrong repository key is shown to the user, the adding will be aborted,
    b) Else the repository is added to the repository list.
2. An application needs the weather plugin.
3. Ambient Dynamix connects to the repository to check for plugin availability.
4. When the repository index is obtained, its signature is checked against the added GPG keys.
    a) The signature check fails, we stop execution of the process.
    b) The signature check succeeds, we move on to the next step.
5. Ambient Dynamix checks whether the calling application has sufficient permissions to use the weather scanner.
    a) If the permissions are insufficient, we stop here.
    b) If the permissions suffice, we move on.
6. Since the weather plugin comes from a community repository, we first download it and check the signature.
    a) The signature check fails, we do not install the plugin.
    b) The signature check succeeds, we continue with the next step.
7. Next, we utilize static code analysis to check if the plugin's behavior fits with the requested permissions.
    a) The static code analysis shows, that there are hidden 'features' inside the plugin, that require extra permissions. In that case, the installation is aborted.
    b) The static code analysis shows, that the plugin needs exactly the permissions it declares or a subset of them, we move on.
8. The weather plugin is installed.
9. Whenever an application wants to make us of the plugin, it first undergoes the same check as before whether it has sufficient permissions to use the weather plugin, since permissions might have changed during an application update.

These are the two use cases, that fully illustrate all the steps the newly implemented solutions will take during the installation of a plugin both from the trusted, official repository as well as from an untrusted community repository.

CHAPTER 6

Critical Reflection

Since this work is split up into three large security targets, we will separately discuss the following important points for each part of the solution:

1. A short recap of the scientific question posed and a brief summary of the solution.

2. Conclusion: This section gives a short summary of the final implementations for the three security goals that were reached.

3. Comparison with Related Work: The proposed solutions of this work are all based on the thought that on mobile devices there is inevitably a trade-off between performance and security. We will see how these solutions compare to work that has been implemented by others.

4. Impact on the User and Developer Environment of Ambient Dynamix: Whenever new security features are implemented, their actions will find some kind of manifestation in the user interface. In the course of this work three crucial security fixes were implemented in Ambient Dynamix. Only the solution to enforcing permissions on calling applications as discussed in section 5.1, has been implemented as a patch to Ambient Dynamix directly. The solutions to assertion of plugin permissions as discussed in section 5.2 and the authentication of repositories as discussed in section 5.3 were implemented as libraries, that could be reused in various applications. In this section, we will discuss the possibilities to implement the solutions into the existing application and the possible consequences for the user experience as well as the developer experience.

5. Open Issues: In this section a short discussion of what could be improved upon the solutions follows.

6.1 Enforcing Permissions on Calling Apps

How can we prevent privilege escalation through Ambient Dynamix by applications that do not request permissions from the Android operating system but use plugins that would require them?

Android provides us with the necessary tools, we just have to perform the right checks in our own code. 5.1

6.1.1 Conclusion

A simple solution that only requires the call of a few API methods in Android is possible. In fact it is very easy to introspect a calling application when intents are used. This way, a fully reliable solution could be found, that could only be circumvented by actually hacking the Android operating system itself.

6.1.2 Comparison with Related Work

The whole concept of Ambient Dynamix would allow creating a bunch of applications that did not require any permissions. But when they are combined with another set of plugins for Ambient Dynamix, it would be possible to run a huge set of operations without the user really understanding that the application could even access them. This concept has been introduced in the presentation [2]. The topic of introspecting the set of permissions of an application calling a method via an intent has not yet been discussed in a scientific fashion. There is only one way to really do this on Android which is pretty stable, since it relies solely on core features of the operating system. It is also a special case, since one could require permissions they have defined themselves. It might be a good idea to add a basic permission to even enable an application to use Ambient Dynamix.

So for the given task, this seems to be the best solution possible.

6.1.3 Impact on the User and Developer Environment of Ambient Dynamix

The assertion, that calling applications have a matching set of permissions to the ones of the plugins, they use, should only concern the developer. It is important to notify developers of software that already uses plugins before releasing the update, to give them a grace period during which they can assert that their application is requesting the right permissions.

If a developer is too slow to release a new build of their application that adheres to the standards, it must be ensured, that the user is informed of the developer's fault. A suggestion needs to be made, to contact the developer and to point out the problem. More difficult than the initial situation is the development and improvement of plugins over time. There are a few cases to be covered here:

- The permissions of the Android security system may change.
- The functionality of the plugin may increase or change, requiring new permissions or different ones.

The first case has already occurred: If one takes a look at the Android developer documentation, and examines the permissions, there are, as of API level 16, two permissions to handle external storage access: "WRITE_EXTERNAL_STORAGE" and "READ_EXTERNAL_STORAGE". While in former times an application could always read the external storage, i.e. the microSD card, this now requires a new permission. While this example is not of relevance for the plugins to be hosted by Ambient Dynamix, since they are not allowed to write anywhere, it clearly shows, that the Android permission system is changing too and the changes might be rather drastic from the point of a single plugin.

The second case is more difficult to handle, especially, when plugin development takes off and more developers jump on board. Most likely, it will be necessary to keep different versions of plugins available and allow software developers to specify the exact version they are developing for. A forced roll-up after a given period of time should be done, to keep the different versions of the various plugins under control.

The most likely solution is to instantiate a sort of information system between plugin- and application developers, to allow them to coordinate their efforts and provide a seamless experience to the user.

6.2 Asserting Permissions of Plugins

How do we prevent plugins, especially from untrusted sources, from exercising unexpected behavior, ones for which they do not have requested permissions?

A robust solution utilizing static code analysis was developed. Since we don't want to allow certain coding techniques, like reflection, this suffices to recreate the permission checks Android would perform, if the code was run as an app.5.2

6.2.1 Conclusion

The chosen approach provides a reasonable trade-off between performance and functionality. By checking all the classes and methods that are used inside a plugin, it is possible to robustly determine if a plugin is requesting exactly the permissions it actually needs. It also allows to block plugins that use methods, like writing arbitrary files to the storage, or using reflection to load additional code that was not visible to the static code analysis.

The performance tests prove, that the method is feasible given the expected average size of a plugin to be used with Ambient Dynamix. Most plugins in the main repository [41] are even smaller than the zephyrhxm plugin. Taking that into account plus the fact, that the check only needs to be done for non-trusted sources and before the plugin is run the first time, it is reasonable to ask a user to wait for a short time, that could even be hidden as part of the installation procedure in most cases.

6.2.2 Comparison with Related Work

This task probably has the strongest tension between tight security and device performance.

Solutions that even go as far as to manipulate the binary on the device have already been proposed, for example in *Improving Privacy on Android Smartphones Through In-Vivo Bytecode*

Instrumentation [22], but as shown by the results of the respective researchers, this is a very slow process, especially for full sized binaries.

In general it is also dangerous to rely on binary modification to enforce desired behavior of an application, as this may corrupt the logic of the application and may fail to prevent undesired behavior. A clever programmer could still use techniques like reflection to circumvent certain methods from being detected. So it would be necessary to rather extensively cut down on the possibilities of the programmer to use advanced programming techniques.

The preferred solution would have been to use debugging techniques directly on the device but Android is greatly hindering such efforts. Since it is designed to be solely a mobile operating system and to rely on computers for any kind of programming tasks. So, it is impossible to find an easy solution for debugging apps directly on the device.

Complex solutions that require rooting or enabling specific settings on device aren't of great help either.

Another beautiful solution would have been to employ an emulator. This would allow extensive behavioral analysis on the various applications, even if they use the native development kit to run C code, or if they make extensive use of reflection. Of course, the emulator would need to be run on a strong server so it could do checks on various plugins, that were previously submitted simultaneously.

6.2.3 Impact on the User and Developer Environment of Ambient Dynamix

While in former times plugins could just claim to need any set of permissions, there is now a strong engine in place to inspect exactly which permissions a plugin needs. While it will not hurt the plugin if it declares more permissions than it actually needs, it can be stopped if it declares too few of them or the wrong ones. When a plugin is accepted for the official repository, it will be checked beforehand, so there cannot be any complications on the user's device.

Once community repositories start to become popular, though the situation may change when suddenly anyone, with benign or malicious intent, can host their own repository and distribute their own plugins. At that point plugin developers must understand the permissions they have to declare.

Users on the other hand, need to be informed about what is happening on on their device, when a new plugin is installed. In the case of a plugin from the official repository, there should not be any problems on the user end.

When an untrusted repository is used, things become a little more difficult: It must be ensured that the plugin does not violate the terms stated by Ambient Dynamix, like no writing of data to any storage. Since the solution developed in this work is freely extensible, this is not a problem to realize. However, the user still needs to be informed of why this plugin will not install and why they cannot do anything to change this. If a plugin needs permissions, that it did not declare, the user interaction is not as clearly defined: The safest way possible would be to inform the user about the problem and allow them to submit a report to the repository owner or plugin producer. Some users may feel restricted by such a behavior. Theese users could be given an option to allow installation on their own terms, clearly warning them about possible consequences in case of malicious plugins.

Finally, the user should be informed about the permissions a plugin needs and what they are capable of doing. This could be done in a way that mimics the original Android application installer. Whichever way is chosen, the user must be made aware of the possibilities a plugin has, without causing paranoia about unwanted behavior.

It might also be interesting to block certain suspicious combinations of permissions for untrusted plugins. For example, if a plugin requests access to contact information and to the internet, it could be intended to leak personal information about the device owner. But as always, it is important to use such automatisms carefully.

All in all, this security extension is probably the most difficult one to handle from the user interface perspective.

6.3 Authenticating Repositories

How can the download process from repositories be secured in a way, so the user can trust, that they are connected to the right repository and get the plugins they requested?

The solution is strongly based on the approach taken by many popular Linux distributions. By utilizing encrypted connections, along with cryptographic signatures of files, it is possible to assert their genuineness. 5.3

6.3.1 Conclusion

The task of implementing package authentication for the repositories was difficult mostly due to the way Android handles cryptography and how to programmatically do the checks that are rather easy to run from a command line. The cryptography library Bouncy Castle, along with the basic work down by the Apache foundation, helped along very well in the creation of a portable solution. Standard ways to use public key cryptography for repository and package authentication have been explored by the various Linux distributions and the obvious winner for this is GPG. It offers an easy way to generate keys for signing repositories and packages and does not require any third party instance to provide additional signing. A basic set of keys can be distributed along with Ambient Dynamix to provide a basis for the chain of trust and could also be extended by further keys for community repositories.

6.3.2 Comparison with Related Work

When looking at ways to authenticate repositories, the open source community, for example OpenSuSE [23] and Debian [24], has already done a very good job in finding ways to do this.

GPG public key cryptography is the best free solution to authenticate repositories to users. Usually it is employed to sign repository index files. It is open for discussion if it is really necessary to provide non cryptographic checksums for packages too, but it sure will not hurt security and the performance impact of creating checksums for small files should also be within limits. However for the implementation it was chosen to keep it simple and lightweight to implement, so only signatures of the files were made. This also suffices to detect bad transmissions from the internet, but mainly serves the purpose of checking authenticity.

6.3.3 Impact on the User and Developer Environment of Ambient Dynamix

Repository authentication is an important feature to make sure a user gets, what they actually expect to receive. When we take a look at the browser world, though, it becomes obvious, that it is not that easy to make users aware of this security feature in a way, they actually understand it.

A good basis for the chain of trust, is to ship Ambient Dynamix with a set of keys for the official repository. So, the user would automatically receive genuine software, as long as they use only the official repository. To further ease the use of other repositories than the official one, it might be a good idea to start a partner program, where trusted community repositories can be established whose keys are either signed by the official Ambient Dynamix key, or are shipped with Ambient Dynamix. This way, a reassuring dialog box could be shown when a user adds a trusted community repository.

When a user decides to add new repositories to their installation, it cannot be avoided to confront the user with the new security feature. If a user still decides to add a completely unknown repository, they still have to be faced with a manual verification screen for the key fingerprint. Probably, only advanced users would ever want to add this kind of repository to Ambient Dynamix though. So it would be reasonable to presume, that they can handle more complex security interaction.

6.4 Reflection on the Architecture of Ambient Dynamix

After the implementation of several security improvements for Ambient Dynamix, it seems worth reflecting upon the initial design and infrastructure of the framework. While Ambient Dynamix is based on a very innovative concept, that attempts to solve problems that plague many smartphones and tablets, especially the one of storage being eaten up by many applications that implement similar functionality, the design left out many security considerations. This led to a situation where security had to be bolted on, rather than being built into the system from the beginning as it should be. If, however, a few design decisions would have been made differently a significant improvement in the security architecture would have been possible.

In the following sections, we will recapitulate the state of how security features have been built into Ambient Dynamix with regard to how the framework could have been implemented differently, to prevent the issues from even arising.

6.4.1 Introspecting Application Permissions

This solution is basically the only one, that relies entirely on standard Android operating system functions. The concept of using the information provided by the Intent and the Android PackageManager cannot be circumvented, unless the whole system has been compromised.

The only improvement that would make sense for this solution would be to require a custom security permission to be set for an application to be allowed to communicate with Ambient Dynamix at all. Other than that, the solution is based entirely on standard functionality of the Android operating system and thus cannot be further improved.

6.4.2 Asserting Permissions of Plugins

The solution found to examination of the true functionality of plugins, is based on static code analysis. While this method gives a good understanding of what the code does, it has its limitations. For example, we are not capable of examining code that was developed with the native development kit, since we can only understand the regular Smali bytecode. Furthermore, there may be ways to sideload new functionality from within the plugin, that is not prevented by the checks that were put in place. These could range from utilizing a solution based on the native development kit, or some methods that seemed harmless but could add up to enabling new functionality.

In order to fix this, it would require a complete redesign, but there would be ways to assert the permissions, without the necessity to do the heavy lifting ourselves: If Ambient Dynamix changed in its self-understanding to behave more like a package manager or app store, it could still serve the purpose, without the worry about security. This could be achieved, if the plugins were turned into full blown Android applications, that would not even need to provide an Activity, but only listen for an Intent from Ambient Dynamix. That way, each application would have to declare its permissions to the Android operating system directly and thereby Android would take care of enforcing them. Furthermore, the plugins would have to assert, that they would only accept an Intent coming from Ambient Dynamix and only report back to the framework. The requirement could be relaxed as well to a concept of creating a common interface for applications to communicate with each other and as a central spot to download apps.

So, Ambient Dynamix would turn into an alternative application store and handle the installation and uninstallation of plugins. Other than that, Ambient Dynamix would only handle the matching of permissions between calling applications and the plugins. This solution would definitely solve the security problem, but it would lead to huge amounts of applications gathering in the application menu, that do not have any user interface components. This may be considered pretty annoying, but it would completely solve the security problem about enforcing the permissions of plugins.

A second approach, would be to provide patches for the Android operating system, that allow the operating system to interact with a package management solution and enforce permissions for light weight applications. These would allow for applications or libraries that do not show up among the usual applications. Since fragmentation is high on the Android platform and older devices are not updated, this solution would only work with very recent devices and probably break backwards compatibility of the Android operating system, as never before.

6.4.3 Package Cryptography

Also, for the topic of cryptographic signatures there might be an easier solution by employing the Android Operating System's functionalities. Continuing the thought from before, that Ambient Dynamix is turned into an application store for light weight apps, Android could do the heavy lifting. Basically, Android employs the same means, that were re-implemented for Ambient Dynamix, namely Public Key Cryptography and only when an .apk file is signed with a proper key, can it be deployed onto a device. This fact could be leveraged to take the check for authenticity of packages off the shoulders of Ambient Dynamix and just let Android do the work. So, for Ambient Dynamix, the only thing that remains to do is to manage the repositories, both official and community based. The task is to verify the repository, or rather the package index of the repository, while Android would take care of verifying the developer certificate.

6.5 Open Issues

In this section we will take a look at the broader field of the given sub problem and see, how things can be improved both with regard to the solutions that were found and in a more general perspective.

6.5.1 Package Management in General

For a mobile operating system it is a pity, that Android did not choose to implement a package manager that resembles those of desktop Linux operating system in general. It would allow to deeply weave a security concept for what Ambient Dynamix aims to do into the operating system and also decrease the individual package size. This would be due to the fact, that there are many common tasks, like scanning QR codes, and not just using the camera, that may be implemented by many applications. If such tasks could just use shared libraries, the size of individual applications would shrink dramatically. Of course, the issue here is, that Android stands for many different devices running different versions of the operating system on a huge variety of hardware. So it would probably end up running relatively slow, since individual performance improvements for single devices could not be applied. Otherwise, it would become very complex to maintain, similar to the present situation concerning the major version updates of Android that take months to ship, if ever, by the device manufacturers.

6.5.2 Inter-Application Communication

Concerning the system of Intents for Inter-Process Communication, it seems like a bad idea, that the calling application does not need to have a strict upper set of permissions from the one it is calling. This lack of control becomes visible especially in the case of Ambient Dynamix. It may also give a false sense of security, since a user is tempted to believe that an application is not really capable of doing much, but combined with the right set of other apps, it could potentially be very harmful.

6.5.3 Checking Program Behavior

The topic of understanding what program code does is difficult to solve comprehensively on current Android devices. A completely stock distribution of Android tries to keep the end user as far away as possible from the underpinnings of the operating system. The Android Debug Bridge does allow for watching debug messages and interacting with the operating system at a lower level, but the powers a root user on a Linux desktop operating system would have are lacking entirely.

Security on Android devices would greatly benefit from behavioral detection mechanisms built into the operating system at a low level. For example it could warn the user if an application accesses contact information and tries to send it via email or SMS a moment later. These kinds of things could theoretically be monitored but it still remains difficult to find out exactly what bad behavior is. With the latest version of Android, version 4.3, some improvements have been made again, by implementing SELinux into the main Android distribution. However only very few devices have been or will be updated to this newest version of Android, leaving vast amounts of users with less security than they should have.

However, the current system of just informing the user on an abstract basis, which permissions an application needs is simply flawed. A common user cannot understand the implication of an application requesting permissions to read contact data while also requesting access to the internet. It is not said, that it makes an application malicious at once, but at least, it could theoretically do malicious things that way. So at least it would help security, if the user could just decide to remove permissions as they please, while being warned that it might ruin certain functionalities of the application.

On the programming level, the Java SecurityManager is clearly missing. Especially in the case of Ambient Dynamix it could have been a great help by enforcing permissions along with the OSGi manager. Yet programmatically there are only voluntary solutions of dropping privileges like the StrictMode, which technically is is only meant for debugging purposes and to check for the correct behavior of an application.

CHAPTER 7

Summary

In this work we have covered three ways to make an Android based middleware solution safer to use. While the concept of a middleware like Ambient Dynamix was not foreseen by the developers of the Android Operating System, it is clearly possible to replicate the most important security features of the Android platform to be imposed upon small application packages run inside a larger application. While there is an obvious performance impact for static code analysis of an untrusted plugin, it is necessary only once when the new piece of code is run on a smartphone and can even be avoided altogether by implementing a strict checking process for the official repository. However, performance measurements have shown, that the expected duration based on the average size of available plugins in the official repository is bearable in the course of the download and installation process. Yet, the solution allows every device to autonomously validate the code of a given application without any complex constructs that would require rooting, access to the original source code or use of debugging techniques, that could be avoided.

Another important topic, the one of privilege escalation, has also been addressed. Android allows for very efficient introspection of the permissions an application has requested. This cannot be circumvented, since it relies solely on the knowledge Android itself has and enforces too. So, while the applications do not contain the code anymore, transparency for the end user is still given, since the applications have to request the permissions upon installation. Otherwise Ambient Dynamix will not allow them to run the plugins that would require them.

Finally, the process of acquiring new plugins from remote sources has been secured properly. While it started out with repositories that were just plain text files residing on http web servers, a solution based on proven open source technology has been found to secure the file downloads. The creation of signatures based on GPG key pairs is in use by several large Linux distributions and provides a simple way to sign packages and check their validity on the client side. Again, the trade-off between security and speed has been kept low, not to harm the user experience but to drastically increase the trust into the legibility of the package source.

So, while there are obvious trade-offs between performance and security on mobile devices, it is possible to design a middleware solution for Android in a way that allows for a satisfying user experience without a complete lack of security.

APPENDIX A

Appendix A

A.1 Performance Tables for Permission Checking

The following time values are all in milliseconds

	1st run	2nd run	3rd run	4th run
Permission map	121	204	198	114
zephyr	1894	1914	1900	1941
barcode	26293	25360	25511	26182

Table A.1: Google Nexus 4

	1st run	2nd run	3rd run	4th run
Permission map	201	133	178	162
zephyr	1318	1561	1571	1289
barcode	20399	20313	20155	18434

Table A.2: Asus TF700T

	1st run	2nd run	3rd run	4th run
Permission map	96	64	43	60
zephyr	1203	934	1142	1086
barcode	14075	13591	13824	13391

Table A.3: Samsung Galaxy S3 i9300

	1st run	2nd run	3rd run	4th run
Permission map	239	140	99	95
zephyr	1793	1586	1781	1708
barcode	22004	21802	21632	21544

Table A.4: Google Nexus 7

A.2 Performance Tables for GPG Signature Verification

	1st run	2nd run	3rd run	4th run
read keyring	147	85	52	147
barcode correct	85	134	152	80
zephyr correct	48	68	74	47
barcode bad-gpg	46	67	68	43
barcode bad-jar	69	94	98	69
zeyphr with barcode gpg	45	81	74	45

Table A.5: Google Nexus 4

	1st run	2nd run	3rd run	4th run
read keyring	112	34	41	41
barcode correct	178	122	147	101
zephyr correct	36	85	70	60
barcode bad-gpg	37	78	86	55
barcode bad-jar	57	98	101	75
zeyphr with barcode gpg	38	94	83	85

Table A.6: Asus TF700T

	1st run	2nd run	3rd run	4th run
read keyring	71	49	90	58
barcode correct	112	47	56	54
zephyr correct	44	42	41	61
barcode bad-gpg	35	65	36	36
barcode bad-jar	62	49	103	53
zeyphr with barcode gpg	38	38	41	61

Table A.7: Samsung Galaxy S3 i9300

	1st run	2nd run	3rd run	4th run
read keyring	256	78	42	58
barcode correct	132	81	66	183
zephyr correct	87	69	93	92
barcode bad-gpg	59	65	68	87
barcode bad-jar	77	85	87	103
zeyphr with barcode gpg	62	70	69	89

Table A.8: Google Nexus 7

APPENDIX B

Appendix B

B.1 Additional Source Code Listings

B.1.1 Mapping Classes and Methods to Permissions

Listing B.1: Scrapy itemtype

```
from scrapy.item import Item, Field

class MethodItem(Item):
    classname = Field()
    methodname = Field()
    permission = Field()
```

Listing B.2: Scrapy spider to extract permissions

```
from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors.sgml import
    SgmlLinkExtractor
from scrapy.selector import HtmlXPathSelector
from scrapy.item import Item

from tutorial.items import MethodItem

import string

class ReferenceSpider(CrawlSpider):

    name = 'reference'
    start_urls = ['file:///usr/share/android-sdk/docs/reference
        /packages.html']
```

```
#We want to parse all html files

    rules = (Rule(SgmlLinkExtractor(allow=['.*/reference/.*.
       html']),callback='parse_permission',follow='true'),)

#callback method to be used when an item is found

    def parse_permission(self, response):
        hxs = HtmlXPathSelector(response)

#using XPath we extract the divs, overview and tables inside
   the html file that somewhere contain a hyperlink to
   permission.html

        divs = hxs.select('//div[div[@class="jd-details-descr
            "]//p//code/a[contains(@href,"permission.html")]]')

        overview = hxs.select('//div[@class="jd-descr"␣and␣./h2
            [contains(.,"Class␣Overview")]]/p//code/a[contains(
            @href,"permission.html")]')

        table = hxs.select('//td[@class="jd-linkcol"␣and␣./div[
            @class="jd-descrdiv"]//p/code/a[contains(@href,"
            permission.html")]]')

        permissions = []

#We are only looking for a specific type of table, the one that
    holds a short overview of all the methods and constants in
   a class

        for index, tab in enumerate(table):
    tabshort = tab.select('nobr//span[@class="sympad"]//a/text
       ()').extract()
    if tabshort:
    permission = MethodItem()

    permission['classname'] = string.replace(response.url[
       string.index(response.url,'reference')+10:string.index(
       response.url,'.html')],'/','.')

    permission['methodname'] = tabshort[0]
    permission['permission'] = tab.select( wouldiv[@class="jd-
        descrdiv"]//p//code/a[contains(@href,"permission.html")
        ]/text()').extract()

␣␣␣␣permissions.append(permission)

#Some␣classes␣require␣a␣certain␣permission␣to␣be␣instantiated,␣
   we␣grab␣them␣here

␣␣␣␣for␣index,␣over␣in␣enumerate(overview):
␣␣␣␣permission␣=␣MethodItem()
␣␣␣␣permission['classname']␣=␣string.replace(response.url[
    string.index(response.url,'reference')+10:string.index(
    response.url,'.html')],'/','.')
␣␣␣␣permission['methodname']␣=␣""
␣␣␣␣permission['permission']␣=␣over.select('./text()').extract
    ()
␣␣␣␣permissions.append(permission)

#Most␣of␣the␣time,␣it␣is␣about␣a␣certain␣method␣that␣requires␣a
   ␣permission,␣so␣we␣only␣require␣it,␣it␣said␣method␣is␣used
␣␣␣␣␣␣␣␣for␣index,␣div␣in␣enumerate(divs):
␣␣␣␣perm␣=␣div.select(␣wouldiv[@class="jd-details-descr"]//p//
    code/a[contains(@href,"permission.html")]/text()').extract()
    divshort = div.select('h4[@class="jd-details-title"]//span[
        @class="sympad"]/text()').extract()
    if perm and divshort:
    permission = MethodItem()
    permission['classname'] = string.replace(response.url[
        string.index(response.url,'reference')+10:string.index(
        response.url,'.html')],'/','.')
    permission['methodname'] = divshort[0]
    permission['permission'] = [perm[0]]
    permissions.append(permission)

        return permissions
```

Listing B.3: Pipeline to filter doubles

```
from scrapy import signals
from scrapy.exceptions import DropItem

class DuplicatePipeline(object):
    def __init__(self):
        self.ids_seen = set()

    def process_item(self, item, spider):
        if (item['classname'] + '.' + item['methodname']) in
            self.ids_seen:
             raise DropItem("Duplicate␣item␣found:␣%s" % item)
        else:
             self.ids_seen.add(item['classname'] + '.' + item['
                methodname'])
             return item
```

Listing B.4: Scrapy settings

```
BOT_NAME = 'permission'

SPIDER_MODULES = ['permission.spiders']
NEWSPIDER_MODULE = 'permission.spiders'
ITEM_PIPELINES = [
    'permission.pipelines.DuplicatePipeline',
]
```

B.1.2 Static Code Analysis

Listing B.5: Main method

```
// get classes.dex

JarFile jar = new java.util.jar.JarFile(jarLocation);
ZipEntry dexfile = jar.getEntry("classes.dex");
File inFile = File.createTempFile("dex",null);

java.io.InputStream is = jar.getInputStream(dexfile); // get
    the input stream
java.io.FileOutputStream fos = new java.io.FileOutputStream(
    inFile);
while (is.available() > 0) {  // write contents of 'is' to 'fos
    '
  fos.write(is.read());
}
jar.close();
fos.close();
is.close();
```

```
// get classes from classes.dex

Trackkeeper k = Trackkeeper.getInstance();
int api = Opcodes.ASM4;
ApplicationReader ar = new ApplicationReader(api, inFile);
ApplicationWriter aw = new ApplicationWriter();
ApplicationVisitor aa = new ApplicationAdapterAnnotateCalls(api
    , aw);
ar.accept(aa, 0);
```

Listing B.6: Converter

```
public void addPackage(String classname) {
  if(classname.startsWith("["))
    classname = classname.substring(1);
  if(classname.startsWith("L"))
    classname = classname.substring(1);
  if(classname.startsWith("org") || classname.startsWith("com")
     )
    classname = classname.substring(classname.indexOf('/')+1);
  String methodname = classname.replace('/', '.');
  classname = classname.substring(0, classname.indexOf('.'));
  classname = classname.replace('/', '.');
  String packname = classname.substring(0, classname.
     lastIndexOf('.'));
  if(!classes.contains(classname))
    classes.add(classname);
  if(!packs.contains(packname))
    packs.add(packname);
  if(!methods.contains(methodname))
    methods.add(methodname);
}
```

Listing B.7: ApplicationAdapterAnnotateCalls

```
@Override
public ClassVisitor visitClass(int access, String name, String
    [] signature, String superName, String [] interfaces) {
  ClassVisitor cv = av.visitClass(access, name, signature,
     superName, interfaces);
  ClassAdapterAnnotateCalls ca = new ClassAdapterAnnotateCalls(
     api, cv);
  return ca;
 }
@Override
```

```
public void visitEnd() {
  av.visitEnd();
 }
```

Listing B.8: ClassAdapterAnnotateCalls

```
@Override
public MethodVisitor visitMethod(int access, String name,
   String desc, String[] signature, String[] exceptions) {
  MethodVisitor mv = cv.visitMethod(access, name, desc,
     signature, exceptions);
  MethodAdapterAnnotateCalls ma = new
     MethodAdapterAnnotateCalls(api, mv);
  return ma;
  }
```

Listing B.9: MethodAdapterAnnotateCalls

```
public void visitMethodInsn(int opcode, java.lang.String owner,
    java.lang.String name, java.lang.String desc, int[]
   arguments) {
  mv.visitMethodInsn(opcode, owner, name, desc, arguments);
  k.addPackage(owner.substring(0, owner.length()-1) + "." +
     name);
}
```

Listing B.10: EmptyVisitor

```
class EmptyVisitor extends ClassVisitor {

AnnotationVisitor av = new AnnotationVisitor(Opcodes.ASM4) {

@Override
public AnnotationVisitor visitAnnotation(String name, String
   desc) {
  return this;
}

@Override
public AnnotationVisitor visitArray(String name) {
  return this;
}
};

public EmptyVisitor() {
  super(Opcodes.ASM4);
}

@Override
public AnnotationVisitor visitAnnotation(String desc, boolean
    visible) {
  return av;
}

public FieldVisitor visitField(int access, String name, String
    desc, String signature, Object value) {
  return new FieldVisitor(Opcodes.ASM4) {

@Override
    public AnnotationVisitor visitAnnotation(String desc,
        boolean visible) {
      return av;
    }
  };
}

public MethodVisitor visitMethod(int access, String name,
    String desc, String signature, String[] exceptions) {
  return new MethodVisitor(Opcodes.ASM4) {

@Override
    public AnnotationVisitor visitAnnotationDefault() {
      return av;
    }

@Override
    public AnnotationVisitor visitAnnotation(String desc,
        boolean visible) {
      return av;
    }

    @Override
    public AnnotationVisitor visitParameterAnnotation( int
        parameter, String desc, boolean visible) {
      return av;
    }
  };
}
}
```

B.1.3 Authentication Repositories and Packages

Listing B.11: Java GPG Keyring class

```
public class BouncyCastleKeyRing implements KeyRing
{
    private final Map<Long, PGPSecretKey> pgpSec = new HashMap<
       Long, PGPSecretKey>();

    private char[] password;

    private final Map<Long, PGPPublicKey> pgpPub = new HashMap<
       Long, PGPPublicKey>();

    private static final long MASK = 0xFFFFFFFFL;

    public BouncyCastleKeyRing()
    {
    }

    public BouncyCastleKeyRing( InputStream secretKeyRingStream
       , InputStream publicKeyRingStream, char[] password )
        throws IOException, PGPException
    {
        addSecretKeyRing( secretKeyRingStream, password );

        addPublicKeyRing( publicKeyRingStream );
    }

    public void addPublicKeyRing( InputStream
       publicKeyRingStream )
        throws IOException, PGPException
    {
        PGPObjectFactory pgpFact = new PGPObjectFactory(
           PGPUtil.getDecoderStream( publicKeyRingStream ) );
        Object obj;

        while ( ( obj = pgpFact.nextObject() ) != null )
        {
            if ( !( obj instanceof PGPPublicKeyRing ) )
            {
                throw new PGPException( obj.getClass().getName
                   () + "␣found␣where␣PGPPublicKeyRing␣expected
                   " );
            }

            PGPPublicKeyRing pgpPublic = (PGPPublicKeyRing) obj
                ;
            long key = pgpPublic.getPublicKey().getKeyID() &
               MASK;

            pgpPub.put( key, pgpPublic.getPublicKey() );
        }
    }

    public void addSecretKeyRing( InputStream
       secretKeyRingStream, char[] password )
        throws IOException, PGPException
    {
        PGPObjectFactory pgpFact = new PGPObjectFactory(
           PGPUtil.getDecoderStream( secretKeyRingStream ) );
        Object obj;

        while ( ( obj = pgpFact.nextObject() ) != null )
        {
            if ( !( obj instanceof PGPSecretKeyRing ) )
            {
                throw new PGPException( obj.getClass().getName
                   () + "␣found␣where␣PGPSecretKeyRing␣expected
                   " );
            }

            PGPSecretKeyRing pgpSecret = (PGPSecretKeyRing) obj
                ;
            long key = pgpSecret.getSecretKey().getKeyID() &
               MASK;

            pgpSec.put( key, pgpSecret.getSecretKey() );
        }

        this.password = password;
    }
}
```

Listing B.12: Signature verifier wrapper

```
public class BouncyCastleOpenPgpSignatureVerifier implements
    OpenPgpSignatureVerifier
```

```
{
    private static final int BUFFER_SIZE = 1024;

    public SignatureStatus verifyDetachedSignature( InputStream
        data, InputStream signature, KeyRing keyRing )
        throws OpenPgpException, UnknownKeyException,
           IOException
    {
        OpenPgpStreamingSignatureVerifier verifier =
            new BouncyCastleOpenPgpStreamingSignatureVerifier(
               signature, keyRing );

        byte[] buf = new byte[BUFFER_SIZE];

        int len;
        do
        {
            len = data.read( buf );
            if ( len > 0 )
            {
                verifier.update( buf, 0, len );
            }
        }
        while ( len >= 0 );

        return verifier.finish();
    }
}
```

Listing B.13: PGP Signature Verifier

```
public class BouncyCastleOpenPgpStreamingSignatureVerifier
    implements OpenPgpStreamingSignatureVerifier
{
    private PGPSignature sig;

    public BouncyCastleOpenPgpStreamingSignatureVerifier(
       InputStream signature, KeyRing keyRing )
        throws OpenPgpException, IOException
    {
        init( signature, keyRing );
    }

    private void init( InputStream signature, KeyRing keyRing )
    throws OpenPgpException, IOException
{
    Security.addProvider( new BouncyCastleProvider() );

    try
    {
        signature = PGPUtil.getDecoderStream( signature );

        PGPPublicKey key = null;
        while ( key == null && signature.available() > 0 )
        {
            PGPObjectFactory pgpFact = new PGPObjectFactory
               ( signature );

            PGPSignatureList p3;

            Object o = pgpFact.nextObject();
            if ( o == null )
            {
                break;
            }

            if ( o instanceof PGPCompressedData )
            {
                PGPCompressedData c1 = (PGPCompressedData)
                   o;

                pgpFact = new PGPObjectFactory( c1.
                   getDataStream() );

                p3 = (PGPSignatureList) pgpFact.nextObject
                   ();
            }
            else
            {
                p3 = (PGPSignatureList) o;
            }

            for ( int i = 0; i < p3.size(); i++ )
            {
                sig = p3.get( i );
                key = keyRing.getPublicKey( sig.getKeyID()
                   );
```

```
                    if ( key != null )
                    {
                        break;
                    }
                }

            }

            if ( key == null )
            {
                throw new UnknownKeyException( "Unable␣to␣find␣
                    key␣with␣key␣ID␣'"
                     + Long.toHexString( sig.getKeyID() ).
                        toUpperCase() + "'␣in␣public␣key␣ring" )
                        ;
            }

            sig.initVerify( key, "BC" );
        }
        catch ( NoSuchProviderException e )
        {
            throw new OpenPgpException(
                                        "Unable␣to␣find␣the␣
                                           correct␣provider␣for
                                           ␣PGP␣-␣check␣that␣
                                           the␣Bouncy␣Castle␣
                                           provider␣is␣
                                           correctly␣installed"
                                           ,
                                        e );
        }
        catch ( PGPException e )
        {
            throw new OpenPgpException( "Error␣calculating␣
                detached␣signature", e );
        }
    }

    public void update( byte[] buf )
        throws OpenPgpException
    {
        update( buf, 0, buf.length );
    }
    public void update( byte[] buf, int offset, int length )
        throws OpenPgpException
    {
        try
        {
            sig.update( buf, offset, length );
        }
        catch ( SignatureException e )
        {
            throw new OpenPgpException( "Error␣calculating␣
               detached␣signature", e );
        }
    }

    public SignatureStatus finish()
        throws OpenPgpException, IOException
    {
        try
        {
            if ( sig.verify() )
            {
                return SignatureStatus.VALID_UNTRUSTED;
            }
            else
            {
                return SignatureStatus.INVALID;
            }
        }
        catch ( PGPException e )
        {
            throw new OpenPgpException( "Error␣calculating␣
               detached␣signature", e );
        }
        catch ( SignatureException e )
        {
            throw new OpenPgpException( "Error␣calculating␣
               detached␣signature", e );
        }
    }
}
```

Bibliography

[1] Darren Carlson and Andreas Schrader. “Adaptive OSGi-Based Context Modeling for Android.” In: *MobiQuitous*. Ed. by Alessandro Puiatti and Tao Gu. Vol. 104. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer, 2011, pp. 209–213. ISBN: 978-3-642-30972-4. URL: `http://dblp.uni-trier.de/db/conf/mobiquitous/mobiquitous2011.html#CarlsonS11`.

[2] Giovanni Russello. *Android Security*. last accessed 09/09/2013. URL: `http://www.cs.auckland.ac.nz/compsci725s2c/lectures/Russello-Smartphone-Security.pdf`.

[3] Arno Becker and Marcus Pant. *Android: Grundlagen und Programmierung*. Heidelberg: dpunkt, 2009. ISBN: 978-3-89864-574-4.

[4] Darren Carlson. *Ambient Dynamix Homepage*. last accessed 09/09/2013. URL: `http://ambientDynamix.org/`.

[5] OSGi Alliance. *The OSGi Architecture*. last accessed 09/09/2013. URL: `http://www.OSGi.org/Technology/WhatIsOSGi`.

[6] Darren Carlson. *The Architecture of Ambient Dynamix*. last accessed 09/09/2013. URL: `http://ambientDynamix.org/documentation/Dynamix-overview`.

[7] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2010.

[8] Arjen K Lenstra and Eric R Verheul. “Selecting cryptographic key sizes”. In: *Journal of cryptology* 14.4 (2001), pp. 255–293.

[9] Google. *Android Developer Documentation*. last accessed 09/09/2013. URL: `https://developer.android.com/guide/components/index.html`.

[10] Oracle. *Java Developer Documentation*. http://docs.oracle.com/javase/7/docs/ last accessed 09/09/2013.

[11] Emily Bridges, Dhriti Kishore, and Joseph Pedo. “Android App Security Analysis”. In: *Governor’s School Of Engineering And Technology Research Journal* (2012).

[12] Adrienne Porter Felt et al. “Android permissions demystified”. In: *Proceedings of the 18th ACM conference on Computer and communications security*. CCS ’11. Chicago, Illinois, USA: ACM, 2011, pp. 627–638. ISBN: 978-1-4503-0948-6. DOI: `10.1145/2046707.2046779`. URL: `http://doi.acm.org/10.1145/2046707.2046779`.

[13] T. Vidas, N. Christin, and L. Cranor. "Curbing Android permission creep". In: *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*. 2011.

[14] Wook Shin et al. "Towards formal analysis of the permission-based security model for android". In: *Wireless and Mobile Communications, 2009. ICWMC'09. Fifth International Conference on*. IEEE. 2009, pp. 87–92.

[15] Sebastian Höbarth and Rene Mayrhofer. "A framework for on-device privilege escalation exploit execution on Android". In: *Proceedings of IWSSI/SPMU (June 2011)* (2011).

[16] Dragos Sbırlea et al. "Automatic Detection of Inter-application Permission Leaks in Android Applications". In: *Technical Report TR13-02* ().

[17] Christopher Mann and Artem Starostin. "A framework for static detection of privacy leaks in Android applications". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing*. ACM. 2012, pp. 1457–1462.

[18] Erika Chin et al. "Analyzing inter-application communication in Android". In: *Proceedings of the 9th international conference on Mobile systems, applications, and services*. ACM. 2011, pp. 239–252.

[19] Jesse Burns. *Developing secure mobile applications for Android*. ISEC Partners, 2008.

[20] Pierre Parrend and Stephane Frenot. "Supporting the secure deployment of OSGi bundles". In: *World of Wireless, Mobile and Multimedia Networks, 2007. WoWMoM 2007. IEEE International Symposium on a*. IEEE. 2007, pp. 1–6.

[21] Phu H Phung and David Sands. "Security policy enforcement in the OSGi framework using aspect-oriented programming". In: *Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International*. IEEE. 2008, pp. 1076–1082.

[22] Alexandre Bartel et al. "Improving Privacy on Android Smartphones Through In-Vivo Bytecode Instrumentation". In: *CoRR* abs/1208.4536 (2012).

[23] OpenSuSE Wiki. *Secure Installation Sources*. last accessed 09/09/2013. URL: `http://old-en.opensuse.org/Secure\_Installation\_Sources`.

[24] Debian Wiki. *Setting up signed Apt Repository with Reprepro*. last accessed 09/09/2013. URL: `http://wiki.debian.org/SettingUpSignedAptRepositoryWithReprepro`.

[25] Sascha Fahl et al. "Why Eve and Mallory love Android: An analysis of Android SSL (in) security". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 50–61.

[26] Justin Cappos et al. "Package management security". In: *University of Arizona Technical Report* (2008), pp. 08–02.

[27] Sean W Smith. "Humans in the loop: Human-computer interaction and security". In: *Security & Privacy, IEEE* 1.3 (2003), pp. 75–79.

[28] Paul Dourish et al. "Security in the wild: user strategies for managing security as an everyday, practical problem". In: *Personal and Ubiquitous Computing* 8.6 (2004), pp. 391–401.

[29] Anthony I. Wasserman. "Software Engineering Issues for Mobile Application Development". In: FoSER '10 (2010), pp. 397–400. DOI: 10.1145/1882362.1882443. URL: http://doi.acm.org/10.1145/1882362.1882443.

[30] Jon Oberheide and Farnam Jahanian. "When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments". In: *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*. ACM. 2010, pp. 43–48.

[31] Martin Gisch, Alexander De Luca, and Markus Blanchebarbe. "The privacy badge: a privacy-awareness user interface for small devices". In: *Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology*. ACM. 2007, pp. 583–586.

[32] Clemens Orthacker et al. "Android Security Permissions – Can We Trust Them?" In: *Security and Privacy in Mobile Information and Communication Systems*. Ed. by Ramjee Prasad et al. Vol. 94. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. Springer Berlin Heidelberg, 2012, pp. 40–51. ISBN: 978-3-642-30243-5. DOI: 10.1007/978-3-642-30244-2_4. URL: http://dx.doi.org/10.1007/978-3-642-30244-2_4.

[33] W. Enck, M. Ongtang, and P. McDaniel. "Understanding Android Security". In: *Security Privacy, IEEE* 7.1 (2009), pp. 50–57. ISSN: 1540-7993. DOI: 10.1109/MSP.2009.26.

[34] B.J. Berger, M. Bunke, and K. Sohr. "An Android Security Case Study with Bauhaus". In: *Reverse Engineering (WCRE), 2011 18th Working Conference on*. 2011, pp. 179–183. DOI: 10.1109/WCRE.2011.29.

[35] The Legion of the Bouncy Castle. *Bouncy Castle Homepage*. last accessed 09/09/2013. URL: http://www.bouncycastle.org/java.html.

[36] Digicert. *Code Signing with Microsoft Authenticode*. last accessed 09/09/2013. URL: https://www.digicert.com/code-signing/microsoft-authenticode.htm.

[37] Darren Carlson. *Ambient Dynamix Main repository version 9*. last accessed 09/09/2013. URL: http://repo.ambientDynamix.org/Dynamix/context_plugins/live9/.

[38] FirebugWorkingGroup. *Firebug*. last accessed 09/09/2013. URL: https://addons.mozilla.org/en-US/firefox/addon/firebug/?src=search.

[39] iSEC Partners. *Android SSL bypass*. last accessed 09/09/2013. URL: https://github.com/iSECPartners/android-SSL-bypass.

[40] OW2 Consortium. *Asmdex Homepage*. last accessed 09/09/2013. URL: http://asm.ow2.org/asmdex-index.html.

[41] Darren Carlson. *Ambient Dynamix Main Repository*. last accessed 09/09/2013. URL: http://repo.ambientDynamix.org/Dynamix/context_plugins/live8/.

[42] Bluebox. *Uncovering Android Master Key That Makes 99% of Devices Vulnerable*. last accessed 09/09/2013. URL: `http://bluebox.com/corporate-blog/bluebox-uncovers-android-master-key/`.