Semdroid

Semantic Android Application Analysis Using Machine Learning

Alexander Oprisnik

Semdroid

Semantic Android Application Analysis Using Machine Learning

by

Alexander Oprisnik

Master's Thesis

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
A-8010 Graz, Austria

January 2014

Assessor: Univ.-Prof. Roderick Bloem, M.Sc. Ph.D.

Advisors: Dipl.-Ing. Dr.techn. Peter Teufl
Dipl.-Ing. Daniel Hein

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am …………………………… ………………………………………………..
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

…………………………… ………………………………………………..
date (signature)

Abstract

Smartphones, tablets, and other mobile devices are becoming an integral part of our daily lives. We entrust these devices with sensitive information about ourselves and we take them everywhere we go. Thus, it is very important to protect this information, to keep the sensitive data secure. The Android platform, which is currently one of the most popular operating systems for mobile devices, already provides several security measurements in order to protect this data and to prevent unauthorized access. However, it is still possible for applications to spy on the user, execute harmful code, or leak sensitive information. Besides detecting such malicious behavior, it is also important to assess the security of benign applications, especially for security-critical scenarios, like for password safes, or for Bring-Your-Own-Device (BYOD) scenarios in corporate environments.

Hence, powerful *analysis tools* are demanded that are able to assess the functionality of applications and to detect suspicious behavior in order to mitigate the risks for users and to improve both the application *quality* and *security*.

In this thesis, we present a new static Android application analysis framework, called *Semdroid*, which employs several different analysis plugins that are capable of assessing an application's functionality. This proposed new framework performs application preprocessing, manages all analysis plugins, and collects the analysis results.
Moreover, we introduce a new static analysis approach, the *Semantic Pattern Analysis*, which is able to accurately determine and pinpoint application functionality. Feature vectors containing analysis-relevant information are extracted from the Android application packages, converted to so-called *Semantic Patterns* and then classified using *machine learning* algorithms. Since the application's components are analyzed separately, the targeted functionality can be accurately pinpointed.

Implemented analysis plugins are able to detect *custom cryptography*, where it is possible to distinguish between *asymmetric-* and *symmetric-key cryptosystems*, and to detect *SMS functionality* included in Android applications. Identifying cryptographic code can help to assess an application's security and can be used as a starting point for subsequent analysis processes. Detecting SMS capabilities helps to identify possible security threats, like SMS spyware, or remote-control-functionality via SMS messages. All plugins have been thoroughly evaluated using both an automated and a manual, empiric evaluation process.

Semdroid can be used on a personal computer, or can be directly deployed onto an Android device for *on-device analysis* of installed applications.

Keywords: Android, application, static analysis, machine learning, Semantic Patterns, malware detection, Support Vector Machine, cryptography detection, SMS-handling detection

Kurzfassung

Smartphones, Tablets, und andere mobile Geräte nehmen eine immer größer werdende Rolle in unserem Alltag ein. Wir vertrauen diesen Geräten unsere privaten Daten an und nehmen sie überall hin mit. Daher ist es sehr wichtig, dass diese Daten geschützt werden. Das Android-Betriebssystem ist zur Zeit eines der populärsten Betriebssysteme für mobile Geräte und stellt schon einige Sicherheitsmaßnahmen zur Verfügung, damit diese Daten sicher bleiben und niemand ohne Erlaubnis darauf zugreifen kann. Jedoch können die Applikationen den Benutzer trotzdem noch auszuspionieren, schadhaften Code ausführen, oder sensitive Informationen durchsickern lassen. Neben Malware-Erkennung ist es außerdem wichtig, die Sicherheit einer gutartigen Anwendung zu verifizieren, speziell in sicherheitskritischen Umgebungen, wie für Password-Safes oder für Bring-Your-Own-Device (BYOD) Umgebungen, wo Mitarbeiter ihre privaten Geräte für Firmenzwecke verwenden können.

Daher werden mächtige Analysewerkzeuge benötigt, die in der Lage sind, die Funktionalität von Applikationen zu bestimmen. Außerdem soll ein verdächtiges Verhalten erkannt werden, um die Risiken für den Benutzer minimieren zu können und die Sicherheit und Qualität von Anwendungen verbessern zu können.

In dieser Arbeit stellen wir *Semdroid* vor, ein Analysewerkzeug für die statische Analyse von Android-Applikationen. Dieses Framework verwendet verschiedene Analysemethoden, welche die Funktionalität von Applikationen bestimmen können. Das Framework bereitet die Applikationen für die Analyse vor, verwaltet alle Analysemethoden und sammelt die Ergebnisse.
Außerdem stellen wir einen neuen Analyseansatz vor, die sogenannte *Semantic Pattern Analyse*, welche in der Lage ist, eine spezifischen Funktionalität in einer Applikation zu lokalisieren. Feature-Vektoren, werden aus der Android Anwendung extrahiert, zu sogenannten *Semantic Patterns* konvertiert und dann mittels *maschinellem Lernen* klassifiziert. Da die verschiendenen Komponenten der Applikation separat analysiert werden, kann die Funktionalität genau lokalisiert werden.

Die implementierten Analysemethoden können kryptografischen Code erkennen und es ist sogar möglich, zwischen symmetrischer und asymmetrischer Kryptografie zu unterscheiden. Eine weitere Analyse ist in der Lage, SMS-Code in Android Applikationen zu detektieren.

Dadurch dass man kryptografischen Code finden kann, kann man die Sicherheit einer Applikation besser abschätzen und den gefundenen Code als Startpunkt für weitere Analysen verwenden. SMS-Code-Erkennung kann verwendet werden um eventuelle Sicherheitsrisiken und Gefahren, wie SMS-Spyware oder Fernsteuerungsprogramme über SMS-Nachrichten aufzudecken. Alle Analysen wurden genauestens evaluiert.

Semdroid kann sowohl auf einem PC als auch auf einem Android-Gerät verwendet werden.

Schlüsselwörter: Android, Applikation, Statische Analyse, Maschinelles Lernen, Semantic Patterns, Malware-Erkennung, Support Vector Machine, Kryptografie-Erkennung

Acknowledgments

First, I would like to thank my advisors, Peter Teufl and Daniel Hein, for their guidance and support over the last couple of months. They patiently answered all of my numerous questions and gave me valuable feedback that allowed me to finish this thesis.

My thanks also go to the entire IAIK team, especially to Roderick Bloem, and to all my colleagues, for having numerous discussions that helped me to steer in the right direction and to finish my studies. Furthermore, I want to thank Keith Andrews for his LaTeX skeleton thesis, which made writing this thesis a little bit easier.

In addition, I would like to thank my friends, who supported me throughout my years of study. Last but definitely not least, I really want to thank my parents, Ingrid and Franz, for their continuous support over the last couple of years. I am very grateful for their endless love and valuable advice. Without them, this work would have never been possible. Thank you.

Alexander Oprisnik

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Overview

Mobile devices are becoming increasingly important for our daily lives. Today, many people use smartphones and tablets on a daily basis, and the number of active users is rapidly growing. One of the operating systems employed on these mobile devices is the Android operating system, which enjoys great popularity. Together with this popularity, it also became an appealing target for attackers. The number of malicious applications is on the rise as more and more applications spy on the user, leak sensitive data, or cause unwanted costs by secretly sending SMS messages to premium rate numbers. Some of these applications even compromise the security of sensitive data unintentionally, for example, by using weak- or no cryptography.

In order to protect this sensitive data, the Android platform employs a permission system[1]. Before an application is installed, the user is notified about the required permissions of the application. Unfortunately, based solely on these permissions, it is not possible to determine whether an application has malicious intentions. For example, if an application requires the permission for reading incoming SMS messages as well as the permission for internet access, it cannot be distinguished whether those permissions are used for malicious activities. The application could listen to incoming SMS messages and then forward them to a remote server without the knowledge of the user.

Furthermore, the requested permissions are not necessarily utilized by the application. According to Felt et al. [15], a third of all Android applications are overprivileged – they request permissions that they do not actually need. Thus, if an application requires the permission to read SMS messages, it might not actually read the messages at all. In addition, some permissions are not very fine-grained. For example, the permission `android.permission.READ_PHONE_STATE` allows, like the name suggests, to read the phone state. Amongst others, this permission is required in order to listen to incoming phone calls and to monitor the phone state. However, with this permission it is also possible to extract the device ID as well as the phone number. Thus, it is not possible to determine which actions are actually performed with the given permission and what sensitive data is actually read by the application. It could be a legitimate usage, which requires reading the phone state, or also an application that spies on the user and transmits the device ID to a given server – or both.

With the release of Android 4.2, Google introduced an improved security system for the Android platform. If the user allows application verification (opt-in), all applications will be analyzed

[1] `http://developer.android.com/guide/topics/security/permissions.html`

before installation[2]. According to Hiroshi Lockheimer[3], Android Vice President of Engineering, a signature of the application is calculated and sent to Google servers for quick identification. This does not only apply to applications installed via Google Play but also to any third-party application installed on the device. With Android 4.3, this app verification mechanism has been moved to the Google Play Services, which allows more frequent updates independent from new Android platform releases[4]. Furthermore, with Android 4.2, Google also changed the way premium SMS messages are handled. If an application wants to send a message to a premium rate number, the user will be notified and asked whether the message should be sent.

Despite these security improvements, Android malware is still a serious threat. According to F-Secure[5], the number of Android threats increased by 49 percent in the first quarter of 2013 compared to the previous quarter. New malware types arise and targeted Android attacks are becoming more popular. F-Secure also states that the number of malware families has doubled over the last year.

Besides detecting malicious applications, it is also important to analyze benign applications, especially applications that have access to security-critical and personal data. Password safes, for example, are used to store login credentials of the user. The user expects, that the password safe securely stores these credentials and that nobody else has access to this data. Unfortunately, this is not always the case; according to Egele et al. [12], 88% of the 11748 applications they have analyzed did not use the cryptographic APIs correctly. So, how can we be sure that we can trust these password safes and messenger applications? How can we be sure that these applications have been implemented correctly, that they store the user's data in a secure way, and that they do not have security flaws?
The answer is simple: *we can't* – at least not without detailed knowledge about the functionality of the application. Hence, sophisticated analysis strategies are demanded that are able to dissect Android applications and to detect certain device functionality, to check whether a given application is secure.

In general, there are two different analysis approaches: *Static* and *dynamic* application analysis. *Static* application analysis relies on analyzing application data without actually executing the application's code. For example, a simple static analysis could look for known suspicious code or search for certain strings and then classify the app based on these results. In contrast to static analysis, *dynamic* application analysis monitors the device state while the application under analysis is executed on a test device or an emulator. Since the application state is monitored throughout execution, it is possible to detect suspicious behavior. For example, a test to detect the SMS-handling capabilities of an application would send an SMS message to the test device. If the application under test reacts to the incoming message, the monitoring system can classify the application accordingly.

Both approaches have their advantages and disadvantages. Suppose an application is able to remote-control the Android device via SMS messages. Using dynamic analysis, this behavior can be very hard to detect; in order to expose this remote-control functionality, a control SMS message has to be sent to the test device in order to trigger this functionality. Since the simulated incoming SMS message has to be an exact replica of a recognized SMS control message and since such a control message could be any arbitrary string, it is almost impossible to check all string combinations and, thus, to detect these remote-controlling capabilities.

[2] `http://source.android.com/devices/tech/security/enhancements42.html`

[3] `http://blogs.computerworld.com/android/21259/android-42-security`

[4] `http://blogs.computerworld.com/android/21259/android-42-security`

[5] `http://www.f-secure.com/en/web/home_global/news-info/product-news-offers/view/story/943273/`

In this case, using static analysis could yield better results, since the entire code, and thus all execution paths can be checked, including the SMS-command-handling functionality.
However, examining all possible execution paths and tracking all variables and registers can be very time-consuming and complex as well. Furthermore, it is possible that a given code segment is never actually called, as for example if a library is included in the application, making "smarter" static analysis approaches very important.

In this thesis, we propose a new static analysis framework, *Semdroid*, as well as an accompanying new static analysis approach, the *Semantic Pattern Analysis*. This approach allows to accurately detect certain application functionality included in Android application packages. Since several application components are analyzed separately (methods, classes, the whole application), it is possible to pinpoint the exact location of the targeted functionality. For example, for an analysis that discerns SMS functionality, the actual Java method responsible for handling the incoming SMS message is given. The *Semdroid* framework itself, which has been implemented in Java, manages all analysis plugins, supplies them with preprocessed application data, and collects all results. These analysis plugins can be based on the Semantic Pattern Analysis, but it is also possible to employ different analysis techniques, based on any other static analysis approach.

How does the Semantic Pattern Analysis work? First, we decide which application components should be analyzed. For instance, it is possible to separately analyze selected methods or classes of a given application, or to analyze the application as a whole. Then, for each of these components, characteristic features, like Dalvik opcodes, method calls, or local variables, are extracted. These feature lists are then converted to so-called *Semantic Patterns* using a previously established Semantic Network. Finally, *machine learning algorithms* are applied on these Semantic Patterns in order to classify and label the corresponding application components according to their functionality.

There are two ways to deploy Semdroid: First, it is possible to analyze applications on a *personal computer* and display the results or output them to a file. Second, a Semdroid Android application is available, which can be directly deployed onto an *Android device*. Installed applications can be dissected and detailed analysis results give the user an insight on what the applications are actually capable of doing. In future work, Semdroid could also be deeply integrated into the Android operating system. Then, Semdroid could be used to assess possible security risks and to take security measures according to the treats – for example, by selectively revoking access to relevant permissions for suspicious applications, or by preventing the installation of such an application.
In theory, a mixed model could also be employed that calculates application fingerprints directly on the Android device and forwards them to a server, which then returns the analysis results to the device. Currently, this third deployment method has not been implemented, but could be addressed in future work.

In this thesis, we are going to present analysis plugins that are capable of detecting *SMS functionality* and *cryptographic code*. The information about included cryptographic implementations might only be of use to IT experts, but the information gathered from other analyses could be of interest to a broader audience. For example, if the user installs a given application that did not mention built-in SMS-handling capabilities, but an analysis concludes that this functionality is included in the application, it helps the user to determine possible security and privacy risks. If the analysis detects that the application is even capable of remote-controlling the device via SMS messages, the user should be very careful and suspicious. If a legitimate SMS application also includes a backdoor that allows to remote-control the device via SMS commands, Semdroid is able to detect both the legitimate SMS handing as well as the remote-control functionality. For the analysis process presented in this thesis, we distinguish between *normal* and *SMS* code. In future work, it could also be possible to further differentiate between different SMS-handling scenarios, like between normal SMS handling and remote-control functionality.

Detecting cryptographic code has been the focus of this thesis. The analysis plugins we have created are able to accurately detect custom cryptographic implementations included in Android applications, and it is even possible to distinguish between symmetric- and asymmetric-key cryptosystems. Going back to the password safes mentioned earlier, this means that we are able to detect custom cryptography utilized by these applications. This information can then be used to assess the security of the application, and thus the security of the login credentials. Container applications[6] are a second use case for cryptography detection. Ideally, these container applications should include a custom cryptographic library in order to be independent of integrated cryptography implementations. Again, detecting such cryptographic functionality can aid a security officer to assess the security of such an application. If custom cryptography implementations are used, it is possible to label them as potential security risks if they are not properly implemented. The security officer can also manually examine the cryptographic code detected by the analysis and evaluate the security of the application.

Due to the flexible framework design, new analysis plugins can easily be developed and integrated into the Semdroid analysis process. In future work, the *Semantic Pattern Analysis* could be used for various analysis tasks, like to determine the application category on Google Play, or to detect UI code, malware, or any other functionality.

1.2 Outline

This section gives an overview of the contents covered in upcoming chapters.
Chapter 2, presents background information on selected topics. First, an overview of the Android platform is given, including details about Android application files, the Android permission system, and the Dalvik Virtual Machine. Furthermore, core Android concepts, like the Intent system and broadcast receivers, are explained. We are also giving a short overview of Android malware, how device are infected, how the malware is activated, and what malicious tasks are performed.
In this thesis, we show how the Semantic Pattern Analysis can be used to detect SMS functionality, as well as cryptographic code included in Android applications. Therefore, we will first explain how SMS messages are handled on the Android platform, followed by a short introduction to cryptography, which explains the basics principles behind symmetric- and asymmetric-key cryptography, as well as hash functions. The focus of the cryptography section lies on the fundamental characteristics of current Java implementations important for code analysis and application classification.
In order to analyze applications, many frameworks, including Semdroid, rely on (partly) decompiling the Android application. Thus, popular decompiling frameworks and their functionality will be presented.
Finally, we will elaborate two core concepts utilized by the proposed new *Semantic Pattern Analysis*, namely machine learning and *Semantic Patterns*, which conclude this chapter.

In Chapter 3, related work is presented. A selection of existing Android application analysis tools is listed and their basic concepts are explained. This chapter is divided into two parts;

[6] Secure enterprise applications used in a Bring-Your-Own-Device environment, where employees can use their private (insecure) devices. Thus, all security features must be implemented and enforced by the enterprise application itself.

first, static analysis frameworks, like the *Semantic Pattern Analysis* are given, followed by several dynamic approaches.

Chapter 4 gives an introduction to *Semdroid* and the ideas behind the proposed new Android application analysis framework. We will elaborate the functionality of this framework and present the analysis workflow. Since Semdroid is a framework suitable for any static analysis, analysis plugins are considered black boxes capable of analyzing given applications. Furthermore, details on the training- and evaluation framework included in Semdroid are given, and different deployment methods are presented.

The next chapter, Chapter 5, delves into the architectural details behind *Semdroid*. It presents the structure of all core components involved in the system and explains how all these components interact with each other.

Then, we will give an overview of the proposed new *Semantic Pattern Analysis* in Chapter 6. First, the analysis workflow is presented, including the component selection, the feature layer generation and the feature selection. Then, the *Semantic Pattern Transformation* and the machine learning process are described, followed by an explanation of the mandatory training process required by the Semantic Pattern Analysis.

After this general overview, Chapter 7 gives architectural details of the *Semantic Pattern Analysis*. This analysis is compatible to Semdroid and can thus be employed within this analysis framework. The main focus of this section lies on the feature extraction process, how features are extracted from Android application packages, and how the resulting feature layers are assembled.

We developed several *Semantic Pattern Analyses* for different tasks, which will be presented in Chapter 8. For each analysis, the thoughts behind the feature selection and combination are elaborated and their intentions are discussed. The first two analysis plugins are able to detect cryptographic code, one for symmetric-key encryption and the second for asymmetric ciphers. Finally, we present an analysis plugin capable of detecting SMS functionality.

Chapter 9 gives detailed evaluation results of the analyses presented in the previous chapter. For each analysis, the evaluation results, including the accuracy, will be presented and their capabilities will be elaborated. In addition, we will present the general performance of the *Semantic Pattern Analysis* for both on-device- and PC-based analysis.

Finally, Chapter 10 concludes this thesis by summarizing *Semdroid* and the *Semantic Pattern Analysis* and its capabilities. Furthermore, we will give an outlook and possible topics that could be addressed in future work.

Chapter 2

Background

This chapter contains background information on selected topics mentioned throughout the thesis. First, details about the Android platform will be discussed in Section 2.1. The Dalvik Virtual Machine and the corresponding Dalvik bytecode of Android applications are presented in Section 2.2. Since Semdroid examines Android application packages, Section 2.3 explains their characteristics and presents their structure and contents.

Since we implemented an analysis process that is able to detect SMS-handling capabilities, Section 2.5 gives details on how SMS messages are handled on Android and how applications can listen to these messages. Some applications allow to remotely control the device via SMS control messages, which can also be utilized for malicious tasks. Thus, Section 2.6, explains the basic concepts behind Android malware.

In order to extract features from Android application packages, we have to dissected these packages. Since this process is related to application decompilation, Section 2.7 gives a short overview of this topic and mentions notable tools.

A second analysis process presented in this thesis is able to detect custom cryptography. Hence, Section 2.8 contains a basic cryptography introduction, which highlights general features and specifications of cryptographic implementations. The focus lies on the characteristics of symmetric- and asymmetric ciphers, as well as of hash functions.

The *Semantic Pattern Analysis* utilizes machine learning as well as Semantic Patterns for the analysis process. Therefore, these two topics are covered in Sections 2.9 and 2.10 respectively.

2.1 The Android Platform

In 2003, Android Inc. has been founded by Andy Rubin. Two years later, in August 2005, Google acquired Android and continued the development of the operating system under the lead of Rubin. The first version has been unveiled in late 2007, as the first product of the Open Handset Alliance[1]. The Android platform is open source and can be found at the Android Open Source Project (AOSP)[2]. The first Android device, the HTC Dream, has been released in October 2008. Five years later, the Android operating system enjoys great popularity as the number of daily activated devices is steadily growing. Since the initial release, updates continuously improved the platform.

[1] http://www.openhandsetalliance.com

[2] http://source.android.com

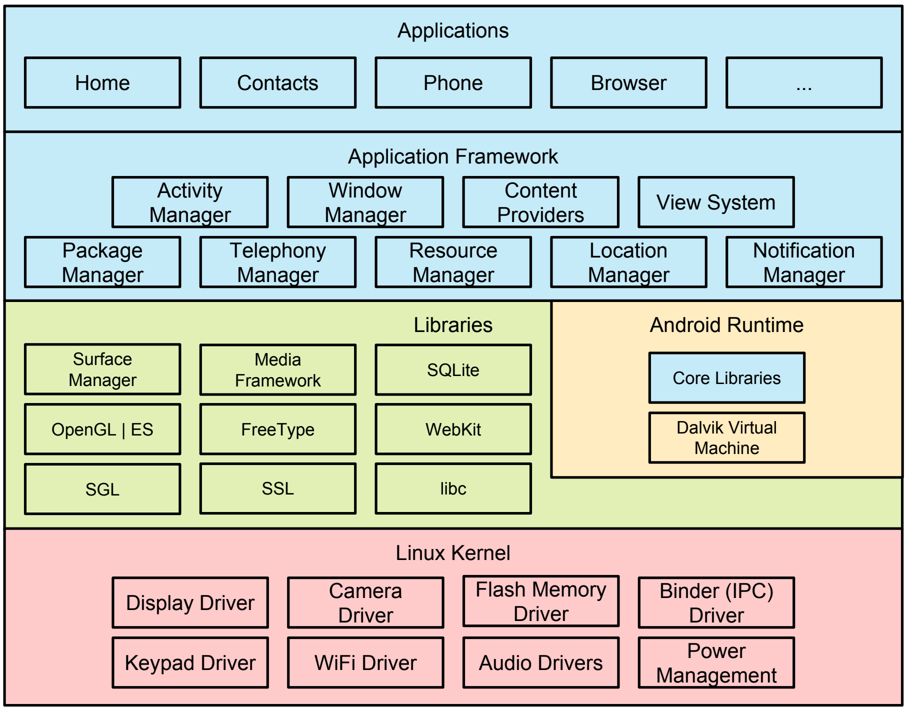


Figure 2.1: Android system architecture. Adapted from http://developer.android.com/images/system-architecture.jpg.

As depicted in Figure 2.1, Android utilizes a Linux kernel for its core functions. On top of this kernel there are several libraries required by the Android operating system. The employed runtime environment is the Dalvik Virtual Machine (Dalvik VM), which executes so-called Dalvik bytecode. Details on this Dalvik VM will be given in Section 2.2.

The application framework holds various managers that can be utilized by Android applications to perform different tasks. Furthermore, a content provider framework allows to provide content across applications, and a view system is employed for the user interface. On top of this application framework, the Android applications are built.

Access to sensitive data, as well as access to certain hardware functionality is restricted by a permission system. Applications have to request access to these system resources, and the user is informed about all required permissions before an application is installed on the device. Section 2.1.3 will take a closer look at this permission system.

According to the Android application fundamentals guide [22], there are four main components Android applications consist of: activities, services, content providers, and broadcast receivers.
Each application can include an arbitrary number and combination of these components and each of these components has different use cases:

- **Activities** are used for components that require a user interface. The guidelines state that for each task, a separate Activity should be used. Since the release Android 3.0, *Fragments* should be employed to encapsulate view components. An activity can include multiple fragments and, depending on several parameters, like the screen size, the display size, or the device orientation, a different layout or fragment combination can be displayed. More information on fragments can be found in the official Android Fragment guide [23].

- **Services** represent background tasks without a user interface. They run in the background without blocking the UI thread. For inter-process communication (IPC), other components can bind to the Service and exchange data.
- **ContentProviders** are, as their name suggests, used to provide content to applications. The *ContentProvider* interface can be used to store and provide data in a unified way. It is possible to make these providers either publicly accessible to other applications installed on the device, or to make their contents private.
- **BroadcastReceivers** are used to listen to events. The developer has to register the receiver with the Android system in order to receive notifications about specific events. The event type the receiver is interested in has to be specified, and a priority can be assigned to the receiver as well. For example, if an incoming SMS message arrives, a system-wide broadcast is sent and all registered broadcast receivers will be notified in order of their specified priority. Since broadcast receivers are an integral part required for handling SMS messages, they will be further discussed in Section 2.1.2.

Intents are used to start activities and services, and to deliver broadcast messages. Section 2.1.1 delves into the details of this intent system. Furthermore, other components include widgets and notifications, which can also be used by the developer. Widgets can be placed on the home screen of the device and they can have various sizes and formats. Notifications are placed in the notification area and can be used to alert the user if important events occurred.
More information on these basic components can be found for example in Meier [33] and Steele and To [50].

2.1.1 The Intent System

The Android operating system utilizes an intent system to deliver messages from one component to another. Intents are also used to broadcast system-wide messages. In addition, intent filters are used to specify the capabilities of components, like which broadcast messages a broadcast receiver would like to listen to.

Intents

According to the official *Intent and Intent Filter guide* [24], intents are messages that activate the three core components – activities, services and broadcast receivers. The *Intent* object holds a given action and can include additional data to operate on. For broadcast messages, the Intent object often holds a description of the event that has occurred and can also include additional information. The main contents of intents are:

- **Action**
  The intent action is a string value representing the action to be performed, or for broadcasts, the action that has occurred. If a phone call should be initiated, `ACTION_CALL` is available that can be set as the intent action. Similarly, other actions are available for various events, like incoming SMS messages.
- **Category**
  This field holds additional information about the category of the component responsible

```
// the parameters are: the context and the activity to be started
Intent intent = new Intent(this , MyActivity.class);

// add extra data to the Bundle
intent.putExtra(MY_STRING, "some data");

// start the activity
startActivity(intent);
```

Listing 2.1: Launching a second activity from a given activity via Intents.

for handling the intent. For example, `CATEGORY_HOME` is used by third party launchers[3] to specify that they can be used as a launcher. Similarly, `CATEGORY_LAUNCHER` has to be set if the activity should be listed in the installed applications list.

- **Component name**
  The component name specifies the component that should handle the intent. For example, if a given activity should be started, the activity name has to be set as component name.

- **Data**
  This field corresponds to the data to be processed. If a contact should be modified, the data field contains the corresponding contact URI as well as the MIME type of the data.

- **Extras**
  The *extras* field can be used to specify additional data. The data structure used for the extras field is a *Bundle*[4], which basically represents a list of key/value pairs. For example, incoming SMS broadcast intents include the sender and the message in the extras bundle. Similarly, if a headphone is plugged or unplugged, a state is supplied in the bundle, which can be used to distinguish between plugging and unplugging.

- **Flags**
  Additional flags can be added to the intent as well. Available flags include options to clear tasks, create a new task or exclude the launched activity from the "recent" list.

More information on the structure of Intents can be found in the Intent API reference[5]. Listing 2.1 gives an example on how to start an activity using intents. The corresponding launch action has to be set, and the component to be started is defined. Additional data is passed to the activity by supplying a Bundle in the extras-field. The launched application can then access the data by calling `getIntent().getStringExtra(MY_STRING).`

Intent filters

Intent filters can be used to specify the capabilities of components. It is possible to create *Intent-Filters* via code, or, the more common way, via XML definitions included in the Android manifest file. Since the application capabilities have to be known before the components are used, intent filters can be defined in the Android manifest, the main configuration file for Android applications (see Section 2.4).

[3] A launcher is the main screen displayed when the device is started. It is used to start other installed applications.

[4] `http://developer.android.com/reference/android/os/Bundle.html`

[5] `http://developer.android.com/reference/android/content/Intent.html`

```
<intent-filter>
    <action android:name="android.intent.action.MAIN" />
    <category android:name="android.intent.category.LAUNCHER" />
</intent-filter>
```

Listing 2.2: Example Intent filter for adding the activity to the application launcher.

The main Activity of an application, which is usually added to the installed applications list and started when the app icon is pressed, must have an intent filter defined that enables these capabilities. Listing 2.2 gives the XML snippet for this filter included in the Android manifest of the application. The intent filter action is set to `MAIN` and the category of the activity has to be `LAUNCHER`.

2.1.2 Broadcast Receivers

The Android operating system offers convenient APIs to listen to broadcast messages. Important system events trigger the broadcast of *Intents* (see Section 2.1.1), containing information about the occurred event. These events include system-boot-complete broadcasts, notifications about incoming SMS messages and phone calls, connectivity changes, battery state updates, and many more. Broadcast receivers have to be registered with the Android system in order to receive events they are interested in by defining an appropriate *IntentFilter*. Meier [33] provides a good resource on how to to implement broadcast receivers.

There are two ways to register broadcast receivers: they can either be declared in the central application configuration file, the `AndroidManifest.xml` (see Section 2.4), or at runtime via Java code.
The first variant is called *static* registration because the receiver is registered automatically when the application is installed and receivers will always be called even if the application itself is not started.
The second variant, *dynamic* registering via `Context.registerReceiver()`[6], requires manual registration and unregistration. Thus, events will only be received if the application (like, for example, a background service) is running. In either case, an *IntentFilter* has to be supplied, which defines the intent actions the receiver requests to listen to (see Section 2.1.1 for more information on Intents). Furthermore, a priority has to be specified for the IntentFilter, which defines the order broadcast receivers will be called with.

Broadcast receivers can also be registered for multiple events. For example, it is possible to implement a receiver that listens to both incoming SMS messages as well as to incoming phone calls.

Furthermore, the Android platforms allows to abort broadcast messages[7] that have been sent through `sendOrderedBroadcast`[8]. Once the broadcast has been aborted, registered receivers with a lower priority than the aborting receiver will not be notified any more. In order to abort broadcast

[6] `http://developer.android.com/reference/android/content/Context.html#registerReceiver(android.content.BroadcastReceiver, android.content.IntentFilter)`

[7] `http://developer.android.com/reference/android/content/BroadcastReceiver.html#abortBroadcast()`

[8] `http://developer.android.com/reference/android/content/Context.html#sendOrderedBroadcast(android.content.Intent, java.lang.String)`

messages, the receiver has to call `abortBroadcast()`[9]. While this is a very useful feature for many use cases, this functionality can also be exploited by malware. For example, incoming SMS messages can be hidden from the user by registering a high-priority SMS broadcast receiver. More information on this topic can be found in Section 2.5.

2.1.3 The Permission System

Android employs a permission system [19] that protects sensitive data and device functionality. Developers have to manually define which permission is required by their applications and the user is notified about required application permissions before the app is installed on the device. For example, if an application wants to listen to incoming SMS messages, the permission `android.permission.RECEIVE_SMS` has to be defined in the Android manifest (see also Section 2.4). Other examples include access to the device state and device IDs, the camera, the external storage, internet access, and much more.
In Section 2.4, an example manifest will be given, which includes the SMS permission (Listing 2.3).

2.2 The Dalvik Virtual Machine

Android applications are written in Java. In addition, they can also include native C/C++ code. The Android platform does not use the classic Java Virtual Machine (JVM) to execute Java bytecode but the proprietary Dalvik Virtual Machine (Dalvik VM). The JVM utilizes `.jar` containers, which include all Java `.class` files, one per Java class. For the Dalvik VM, all `.class` files of an application are transformed to a single Dalvik executable (`.dex` file). This file holds the Dalvik bytecode, which is then executed on the Android device. The *dx* tool, which is included in the Android platform tools, is used for this conversion.

The Dalvik VM is register-based as opposed to the stack-based architecture of the JVM. According to Nolan [39], the JVM consists of four distinct parts: The heap, program counter registers, a method area and the JVM stack. More information on the JVM, the Dalvik VM and their differences can be found in The Android Open Source Project [56] and Nolan [39].

2.2.1 Dalvik Bytecode

The instruction set for the Dalvik VM consists of 218 opcodes. As already stated, the Dalvik VM is register-based. The opcodes can directly operate on these registers. A full list of all opcodes can be found in The Android Open Source Project [56]. There are 31 different opcode formats available, each of them having a unique ID. The Android Open Source Project [58] gives an overview of these instruction formats.

2.2.2 Dalvik Executable Format

When the Android application package (Section 2.3) is created, the Java `.class` files are converted to a single `classes.dex` file, the Dalvik executable. The structure of this file is depicted in Figure 2.2, according to the official Android Dalvik documentation [57] and to Nolan [39].

[9] `http://developer.android.com/reference/android/content/BroadcastReceiver.html#abortBroadcast()`

Dalvik	Java
B	byte
C	char
D	double
F	float
I	int
J	long
S	short
V	void
Z	boolean
Lmy/package/Clazz;	the class my.package.Clazz
[typeDescriptor	array of typeDescriptor

Table 2.1: Dalvik variable string conventions as listed in The Android Open Source Project [57].

The header contains general information about the executable. All strings utilized by the application can be found in the *string_ids* block. For each of these strings, a string element containing the string ID and the actual address of the string in the *data* section is included. Next, the *type_ids* section contains all type definitions. Again, each type has a unique type ID as well as a string ID, which links to the type string, like `Ljava/lang/BigInteger`. Here, we also see the string naming conventions for variables. Table 2.1 gives an overview of these conventions. For more information on the string naming conventions of Dalvik, we refer to The Android Open Source Project [57].

The three blocks *proto_ids*, *field_ids*, and *method_ids* list all method prototype, field, and method definitions. Prototype ID items have four fields: a unique ID, the *shorty_idx*, which is a short-form descriptor for the given prototype, a return type ID, and a parameters offset to the list of parameter types of this method prototype. Field ID items also feature four fields: again, a unique ID, a class string ID, a type ID, and a name string ID. Finally, the method ID item consists of a unique ID, a prototype ID, and a name String ID.

Then, *class_defs* section holds all class definitions, where each class has its own ID, access flags, superclass ID, interface offset, source file ID, annotations offset, class data offset, and static values offset. *Link data* contains data used in statically linked files. The *data* section contains the actual data, where all offsets from previously discussed items lead to. For example, for each class, a *class_data_item* is available, which contains, amongst others, all encoded fields and methods. For each method, the address offset is given, which points to the actual code item of the method. For more detailed information on the Dalvik executable format, we refer to The Android Open Source Project [57] and to Nolan [39].

The Android SDK includes *dexdump*, a tool to print the contents of a Dalvik executable.

2.3 Android Applications

Android applications are implemented in Java and can also include native C/C++ code. In addition, XML can be used to define user interface components and for various other resource types,

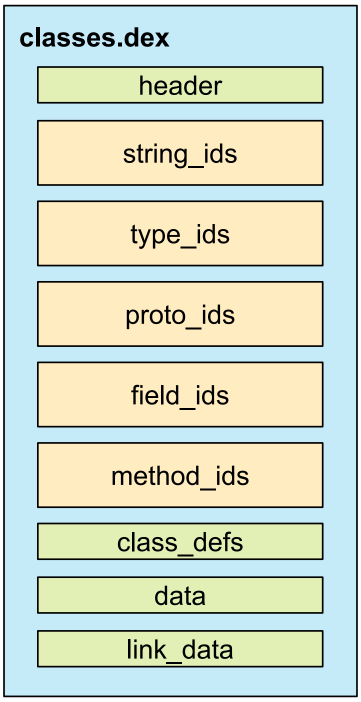


Figure 2.2: Dalvik executable structure (`classes.dex`). Adapted from Nolan [39] and The Android Open Source Project [57].

including strings, dimensions, and many more. The file extension of Android applications is `.apk`, which is short for *Android application package*. This Android application package is a ZIP compressed container holding several files and folders. The files included in this package are signed[10] by the developer using *Jarsigner*. Figure 2.3 shows the contents of such an `.apk` file, which are:

- **AndroidManifest.xml**
  The Android manifest represents the main configuration file for applications. It includes definitions for supported devices, used permissions and lists all application components. More information on the manifest is given in Section 2.4.

- **classes.dex**
  The Dalvik bytecode is stored in the `classes.dex`. As already explained in Section 2.2, Android uses a proprietary Dalvik Virtual Machine and Dalvik bytecode. Details on the structure of the `classes.dex` as well as examples can be found in Section 2.2.2.

- **resources.arsc**
  This file contains the Android resource table, which holds information on all used resources, including their ID.

- **assets**
  Additional files required by the application are placed in this folder. Unlike files in `/res/raw/`, files put in the assets folder will *not* receive an ID.

- **lib**
  This folder contains native (compiled) libraries required by the application. The shared ob-

[10] http://developer.android.com/tools/publishing/app-signing.html

jects (.so file extension) have to be placed in a subfolder named after the target architecture. According to the Android NDK documentation[11], supported folder names are, amongst others, armeabi for ARM processors, armeabi-v7a for ARMv7, x86 for X86 platforms or mips for the MIPS architecture.

- **META_INF**
  This folder contains three files:
  - MANIFEST.MF
    This file contains a list of all files included in the Android application package, as well as their corresponding SHA-1 digests.
  - CERT.SF
    Android application packages are signed using *Jarsigner*. According to the Jarsigner documentation [40], a signature file, called CERT.SF, as well as a signature block file, CERT.RSA are created. The CERT.SF holds the SHA-1 hash value of the MANIFEST.MF file, as well as separate SHA-1 values for all entries included in the MANIFEST.MF file. This signature file is then signed with the credentials of the developer and the resulting signature is stored in CERT.RSA, together with the developer's certificate.
  - CERT.RSA
    This signature block file contains the signature of CERT.SF and the certificate used for the signing process.
- **res**
  All resources required by the Android application are placed in this directory. For example, user interface elements can be designed via XML, as well as animations or graphics. Images and other raw values, like colors, strings, or integers can be defined in resource files as well. More information on Android resources can be found in the official *App Resource* guide [21] and in Meier [33].

2.4 Android Manifest

The Android manifest is an XML file where all application components are defined. First, the package name of the application, as well as the version ID and version string have to be set. In addition, the targeted Android version and supported screen sizes have to be specified. Activities, services, broadcast receivers, and content providers are registered in this file. Intent filters are used to specify the capabilities of these components. Furthermore, all required permissions also have to be defined in the manifest. The developer can also define which hardware- or software features the application demands[12], like a camera, NFC, or Bluetooth. The Android manifest guide [20] gives more information on the structure of the AndroidManifest.xml.

Listing 2.3 shows a small example manifest. In this example, the package name is set to com.oprisnik.smsreceiver. The application requires at least Android *Donut*[13] (API level 4) and the targeted Android version is *Jelly Bean* (API level 18).

Since the application reacts to incoming SMS messages, the permission to receive SMS messages is declared; the uses-permission-tag is used for this purpose. In this case, the demanded permission is called android.permission.RECEIVE_SMS.

[11] http://developer.android.com/tools/sdk/ndk/index.html

[12] http://developer.android.com/guide/topics/manifest/uses-feature-element.html

[13] http://source.android.com/source/build-numbers.html

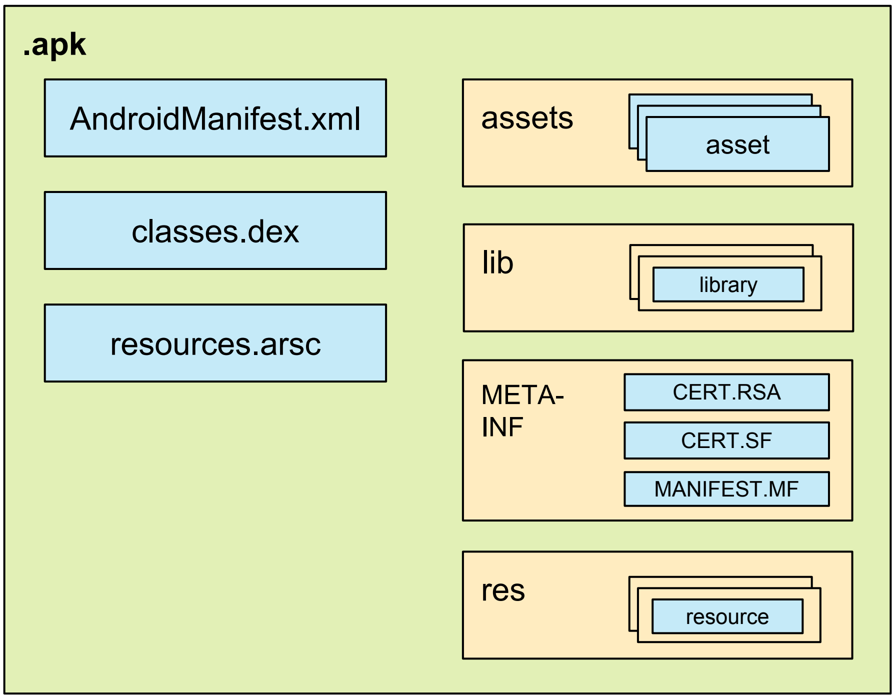


Figure 2.3: Contents of an Android application package file (`.apk`). Files are listed in blue, folders are yellow. The Android application package itself is a ZIP compressed container.

The application itself consists of a single broadcast receiver named *SmsReceiver*, for which an intent filter (Section 2.1.1) is defined. The action `android.provider.Telephony.SMS_RECEIVED` is set for this filter, which registers it to incoming SMS messages. Furthermore, a priority of 1000 is defined for the filter. The next section delves into the details of SMS handling on Android and gives an example implementation for the *SmsReceiver* declared earlier.

2.5 SMS Handling

As stated in Section 2.1.2, broadcast receivers can be used in order to get notified about various events – including incoming SMS messages. The receiver can either be registered statically in the Android manifest, as explained in the previous section, or dynamically via Java code. The priority of the broadcast receiver defines the call order for all registered receivers. Default SMS applications usually have a low priority and, thus, will be called last. For the receiver registered in Listing 2.3 this means that it will be called before default SMS applications, since the *SYSTEM_HIGH_PRIORITY* is 1000 – which has been assigned to the receiver.

Now, we need to implement the broadcast receiver itself. Listing 2.4 shows a very simple implementation for such a receiver. The receiver, called *SmsReceiver*, has to extend the abstract *BroadcastReceiver* class. Since the receiver has been registered to `SMS_RECEIVED` actions, the `onReceive`-method will be called for incoming SMS messages. Information on the incoming SMS message are stored in the bundle of the intent and can be accessed via the `pdus` key. The code in Listing 2.4 parses all SMS messages, which can then be used for further tasks. For example, for each message, the originating address as well as the message body can be extracted and saved in a database.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.oprisnik.smsreceiver"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk android:minSdkVersion="4" android:targetSdkVersion="18" />

    <uses-permission android:name="android.permission.RECEIVE_SMS" />

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >

        <receiver android:name=".SmsReceiver">
            <intent-filter android:priority="1000">
                <action
                    android:name="android.provider.Telephony.SMS_RECEIVED" />
            </intent-filter>
        </receiver>
    </application>
</manifest>
```

Listing 2.3: Simple AndroidManifest.xml for SMS handling. A BroadcastReceiver for SMS events is defined. The receiver priority is set to 1000. The SMS permission has to be requested in order for SMS handling to work.

Since it is possible to abort broadcasts, an SMS receiver with a high priority can abort the broadcast, which will effectively hide the message from all other receivers with a lower priority – including the default SMS application. The message will not be added to the user's inbox and the user will not be notified about this messsage. Uncommenting line 24 in Listing 2.4, `abortBroadcast()`, enables this behavior; receivers with a lower priority will no longer receive the intent and, thus, incoming SMS messages.

With Android 4.4 KitKat, Google changed the way SMS messages are handled on Android. According to Main and Braun [32], a default SMS application has to be set by the user. Only this default application can write to the SMS provider and only the default SMS application will receive the broadcast message with the new action `SMS_DELIVER_ACTION`. Other applications can still listen to incoming SMS messages via the `SMS_RECEIVED_ACTION` – but they *cannot* abort the broadcast any more. All applications will receive the broadcast. Thus, applications cannot abort the broadcast any more. For default SMS applications, it is still possible to hide certain messages from the user, since they can decide not to add incoming messages to the SMS content provider.

In this thesis, applications that hide incoming messages from the user, either by aborting the broadcast, or by manipulating the SMS content provider, are called *"SMS catchers"*. Receivers that just listen to incoming SMS messages without hiding certain messages from the user's inbox are called *"SMS sniffers"*. Both types usually have a high priority set for their broadcast receivers in order to receive all messages.

```
public class SmsReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(final Context context, Intent intent) {
    // get the intent extras
      Bundle bundle = intent.getExtras();

    // parse SMS objects
        Object[] pdus = (Object[]) bundle.get("pdus");
    SmsMessage[] messages  = new SmsMessage[pdus.length];

    for (int i = 0; i < messages.length; i++) {
      messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);

      // Read message address and body
      // messages[i].getOriginatingAddress();
      // messages[i].getMessageBody();
    }

    // do something with the messages
    doSomething(messages);

    // uncomment to abort the broadcast
    // abortBroadcast();
    }
}
```

Listing 2.4: Simple SMS broadcast receiver. Parses incoming SMS messages. If uncommented, `abortBroadcast()` will abort the SMS broadcast.

2.5.1 SMS Sniffers

SMS sniffers listen to incoming SMS messages without aborting the message broadcast. Listing 2.4 can be considered as a sniffer, since the broadcast is not aborted. Depending on the actions performed in `doSomething`, they can be categorized in two sub-groups:
The first sniffer type just records that a message has been received, *without* reading the actual SMS contents or the phone number of the sender. For this type, only the fact that an SMS has arrived is important, not the actual contents of the message. Notification widgets that just display the unread message count would fall in this category.

The second category of SMS sniffers parse and use the message contents. Third-party SMS widgets or applications that sync your messages with other devices can be placed in this category.

The tasks performed by sniffers can either be malicious or benign. Examples for legitimate sniffers have already been given. Malicious sniffers could forward all incoming SMS messages to a third party, for example via the internet or via SMS messages. Furthermore, one could also sniff sensitive data, like transaction authentication numbers (TANs) used by online banking systems. These TANs can then be used to perform unwanted bank transactions. Similarly, all systems that employ two-factor authentication via SMS messages can be exploited as well. Two-factor authentication adds a second layer of security by sending verification PINs to the user, for example, via SMS messages. The user then has to enter this PIN in order to be able to successfuly log in. If the device is infected, an adversary is able to acquire these verification PINs and, thus, is able to log into the user's account (assuming that the attacker is in possession of the user's password as well).

2.5.2 SMS Catchers

SMS catchers behave like sniffers but, in addition, abort the SMS broadcasts. Again, two different catcher types can be distinguished: The first category aborts all SMS broadcasts. The second category only aborts the broadcast if certain criteria are met, like if the incoming message has a special structure or if it originates from a given phone number.

Since the broadcast is cancelled, other broadcast receivers with lower priorities will not receive the message, including most default SMS applications. Thus, the user will not be notified about the incoming message.

With the release of Android 4.4, aborting SMS broadcasts is not possible any more. For SMS catchers this means, that they have to be the default SMS application. If they receive certain SMS messages, they just do not display them to the user (add them to the database) and perform their hidden tasks accordingly. Thus, third party SMS applications set as default SMS applications can still act as SMS catchers. It has to be noted though, that other installed applications that listen for incoming SMS messages will also receive the broadcast.

In general, it is very hard to determine whether an SMS catcher has malicious intentions. There are some use cases where SMS catchers can be used for legitimate operations. Many mobile ticketing providers, for instance, handle the ordering and billing process via SMS messages. SMS catchers can be used to hide the SMS handling from the user, which can improve the usability of an application since the user simply has to press a single button. All the background work consisting of sending corresponding SMS messages to the ticketing providers and handling the SMS responses is done directly by the application. The resulting booking information retrieved via SMS messages can then be directly displayed in the application in a more appealing way.

Blacklisting of unwanted numbers can also be a desired feature of a third-party SMS applica-

```java
public class SmsCommandReceiver extends BroadcastReceiver {

  @Override
  public void onReceive(final Context context, Intent intent) {
    // get the intent extras
    Bundle bundle = intent.getExtras();

    // parse SMS objects
    Object[] pdus = (Object[]) bundle.get("pdus");
    SmsMessage[] messages  = new SmsMessage[pdus.length];

    for (int i = 0; i < messages.length; i++) {
      messages[i] = SmsMessage.createFromPdu((byte[]) pdus[i]);
      doSomething(messages[i].getMessageBody());
    }

    // save messages etc. like a legitimate SMS application
    ...
    }

  public void doSomething(String command) {
    // handle the SMS command
    if(command.equals("MY_COMMAND")) {
      // perform the action according to the command
      performAction();

      // abort the broadcast so that no other SMS receivers get the message
      abortBroadcast();
    }
  }
}
```

Listing 2.5: Simple SMS command broadcast receiver. Parses incoming SMS messages. if the incoming SMS message equals `MY_COMMAND`, a given action is performed.

tion. This case can also be considered as a SMS catcher for messages from blocked numbers.

Remote-control software can include SMS catchers for their tasks as well. By sending command SMS messages to the device, remote control functionality, like acquiring the location of the device or deleting certain data, can be provided. Incoming messages are scanned and if they match a given pattern, a command is be executed and the broadcast is aborted. Listing 2.5 shows a simple implementation of such a command receiver. If the SMS message equals `MY_COMMAND`, a given action, like sending the current device location to a given server, is performed.

Malicious apps can utilize this feature to spy on the user or to perform various other tasks. Sensitive data, the device location and other valuable information can be returned to the adversary. Similarly, security applications can utilize this feature to locate your device or to delete sensitive data in case the device is lost. Thus, this functionality is utilized by both malicious and legitimate applications.

TAN sniffers discussed in the previous section can also be implemented as TAN catchers, where the SMS broadcast containing the TAN is intercepted, forwarded to a third party, and then cancelled. The user will not get a notification about the incoming TAN.

2.6 Malware

Today, several different malware types can be found in Android applications. As already discussed, SMS sniffers and catchers pose a threat. They can be used to spy on the user or to remote-control the device. SMS commands can be sent to the device and actions can be performed accordingly.

Gunasekera [26] gives an overview of Android security and also gives an overview of Android malware. The malware genome project [68] gives detailed insights into several other malware families and their characteristics found in applications from 2010 and 2011. They collected 1260 malware samples, which can be categorized into 49 different malware families. Three key points have been assessed for malicious applications by Zhou and Jiang [68]: malware **installation**, **activation**, and **functionality**. First, the malicious code has to be *installed* on the device. There are different possibilities on how the device can get infected. Common strategies include repackaging, update, or drive-by-downloads. Repackaging means that malicious code is inserted in a (popular) application, repackaged, and distributed. Similarly, the *update* scenario includes an update functionality into the repackaged application. When the application is used, the malicious code will be downloaded and executed. Drive-by-downloads trick the user into downloading malicious applications from the attacker's website or from a fake market – for example via advertisements for their "great", malicious applications.

Next, the malicious code has to be *activated*. Again, there are many possibilities on how this can be achieved, ranging from boot receivers, to SMS and call receivers, listening to battery updates or when the USB cable is plugged in. Of course, it is also possible that the malicious code is executed when the user starts an application or performs certain actions within the malicious application.

Finally, the *functionality* of the malicious code has to be assessed. There are several possibilities for malicious operations that have been found in existing applications. Common intentions of malicious applications can be categorized as follows:

- **Root exploits**
  The application tries to gain root access, which allows the application to access the whole system.
- **Remote control functionality**
  The device can be remote-controlled via SMS messages or the internet.
- **Spyware**
  The application spies on the user and sends sensitive data, like the location, incoming messages, contacts, or calendar appointments to the attacker. As already stated, such functionality can also be desired, as for example for locating a lost device or for parental control.
- **Financial charges**
  The malicious application calls premium rate numbers or sends text messages to premium rate numbers. With the release of Android 4.2, Google added a notification dialog that asks the user if the message really should be sent to the premium rate number[14].

For more detailed information on malware in general as well as on different malware families found in the wild, we refer to Zhou and Jiang [68]. Furthermore, Gunasekera [26] and Castillo [4] provide additional information.

[14] `http://source.android.com/devices/tech/security/enhancements42.html`

2.7 Decompiling Android Applications

It is possible to reconstruct Java code from the Dalvik bytecode. Nolan [39] gives detailed information on the decompilation process, and explains how to implement your own Android decompiler. Most decompilation frameworks first convert the Dalvik executable to a `.jar` file containing all `.class` files. Then, these `.jar` files containing Java bytecode can be decompiled to Java source code using various tools, like *JD-GUI*[15] or *JAD*[16]. Commercial versions like *JEB*[17] are available as well.

A popular tool for converting the Dalvik executable to Java bytecode is *dex2jar* [9]. Alternatively, many other tools like *Dare*[18] or *ded*[19] can be used for this conversion process.
Some frameworks also transform the Dalvik bytecode to a different intermediate language, like for instance *smali* [48]. This disassembler utilizes its own intermediate language and stores them as so-called `.smali` files.

Since the XML files used for the Android manifest as well as for various other purposes, including user interface components, are stored in a binary format, tools like *AXMLPrinter2*[20] or *axml*[21] can be used to transform the binary resources back to a human-readable XML format.

Apktool[22] is a very popular decompiling tool, which aims at simplifying the decompilation process. It performs all of the steps mentioned above – with a single click; the Dalvik bytecode will be converted to a `.jar` file and JD-GUI is then used to display its Java source code. Furthermore, the Android binary XML files will be converted to normal, human-readable XML. In addition, the Dalvik bytecode is converted to *smali*, which can be used for step-by-step debugging.

In order to protect one's intellectual property and to prevent piracy, many applications use code obfuscation to hinder reverse engineering. For Android, the most prominent obfuscation tool is *ProGuard*[23], which ships with the Android SDK. Furthermore, this tool also optimizes the code and shrinks it. For example, methods are inlined, classes are merged, and other code optimization is performed. *DexGuard*[24] is also available, which is a commercial optimizer and obfuscator for Android applications. Compared to ProGuard, it offers advanced functions, like string-, class-, and asset encryption.

2.8 Cryptography

In order to establish a secure connection between two parties, cryptographic mechanisms can be employed in order to encrypt their communication. Figure 2.4 shows how such a system looks in general. The first party encrypts a given plaintext using a cryptograpic key *K1*. The resulting ciphertext can then be transmitted to the second party, even over an insecure channel. An adversary is not able to decrypt the ciphertext without knowing the secret key *K2*. Only the second party, which is in possession of *K2* can recover the original text.

[15] `http://jd.benow.ca/`

[16] `http://en.wikipedia.org/wiki/JAD_(JAva_Decompiler)`

[17] `http://www.android-decompiler.com`

[18] `http://siis.cse.psu.edu/dare/index.html`

[19] `http://siis.cse.psu.edu/ded/index.html`

[20] `https://code.google.com/p/android4me/`

[21] `https://code.google.com/p/axml/`

[22] `https://code.google.com/p/android-apktool/`

[23] `http://developer.android.com/tools/help/proguard.html`

[24] `http://www.saikoa.com/dexguard`

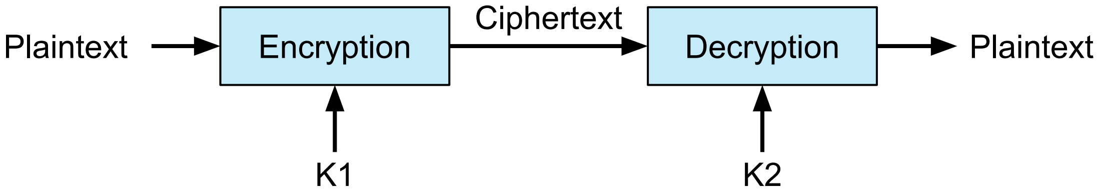


Figure 2.4: Secure communication between two parties. The plaintext is encrypted using a cryptograpic key *K1*. The ciphertext is sent to the second party, which can decrypt the original plaintext using key *K2*.

Encryption mechanisms can be divided into two main categories: *symmetric-* and *asymmetric* encryption mechanisms. For symmetric-key encryption, the same cryptographic key is used for en- and decryption, thus *K1* and *K2* of Figure 2.4 are identical. Asymmetric encryption utilizes two different cryptographic keys, one for encryption (*K1*), and a second key for decryption (*K2*).

Another category of cryptographic functions are hash functions. They are used to map arbitrary input data to a fixed-length output – the so-called hash value. In the following sections, the main characteristics of these two types will be dissected and important algorithms will be presented.

The next sections will give a brief overview of these three categories. Since one goal of this thesis is to detect symmetric-, and asymmetric ciphers, as well as hash functions, we focus on general structural details of these three categories. For more information, we refer to and Trappe and Washington [59], Menezes, Vanstone, and Oorschot [34] and Schneier [45].

2.8.1 Symmetric Cryptography

Symmetric block ciphers use the same key for both en- and decryption. Two of the most common symmetric ciphers are the *Data Encryption Standard* (DES) and the newer *Advanced Encryption Standard* (AES). These ciphers are so-called *block ciphers*. Block ciphers split the input in blocks of a predefined size and encrypt each block separately. There are different modes of operation for block ciphers, since encrypting all blocks separately has some weaknesses and should not be used for all applications. For more information on block cipher modes of operation, we refer to Menezes, Vanstone, and Oorschot [34].

In this thesis, we will focus on AES, since AES is very common and we use different AES implementations as training data later on in this thesis. AES, which is the successor of DES, has been developed by Joan Daemen and Vincent Rijmen under the name *Rijndael*. The official AES specification can be found in National Institute of Standards and Technology [37]. The block size of AES is 128 bits, and it supports key sizes of 128, 192, and 256 bits.

The input, also called the state, can be represented as a 4x4 array of bytes. The algorithm itself utilizes 10, 12, or 14 rounds (depending on the key size), where each round transformation consists of the following four steps:

- **SubBytes**
  The first step, SubBytes, adds non-linearity by transforming the input bytes using an invertible S-Box (lookup table).
- **ShiftRows**
  As its name suggests, ShiftRows is responsible for shifting all rows over different offsets.

- **MixColumns**
  MixColumns transforms all columns of the current input state.

- **AddRoundKey**
  For each round, a round key is calculated and added to the input state.

For more details on these four steps and on the key schedule, we refer to National Institute of Standards and Technology [37] and Stinson [51].

If we take a look at the AES implementation of Bouncy Castle given in Listing 2.6, it can be observed that many mathematical operations are performed. The four steps are not clearly visible, since this code is optimized. The *SubBytes* operation manifests in array lookups for `S[]`, `T0[]` holds precomputed values for the rounds. `KW` represents the working key. The algorithm uses solely basic data types (byte and int) as well as arrays of these types. Many *XOR*, *AND*, and *shift* operations are used. For all Sbox lookups, the corresponding array elements are accessed.

2.8.2 Asymmetric Cryptography

Asymmetric encryption algorithms use different keys for en- and decryption. These keys are also often called public- and private key, since one key can be made publicly available, whereas the other key has to be kept private. Then, everyone can send messages to the party by encrypting the data with the public key. The message can only be decrypted by persons who know the private key.

Similarly, it is also possible to sign content with the private key. Everyone can then validate the signature by using the public key. For large datasets, it is not practicable to sign the entire file (i.e. to encrypt the data with the private key). Thus, the data is first hashed (see Section 2.8.3) and only the resulting hash value is signed with the private key.

One of the most common asymmetric encryption algorithms is *RSA*, named after the three inventors Ron Rivest, Adi Shamir and Leonard Adleman. In this thesis, we use RSA implementations as training data. Thus, we need to take a closer look at this asymmetric encryption mechanism. In order to encrypt a message m with the public key e, the following algorithm is used:

$c \equiv m^e \pmod{n}$

The encrypted message c can then be decrypted by using the private key d and by applying:

$m \equiv c^d \pmod{n}$

The modulus n consists of two large prime numbers p and q:

$n = p * q$

The private keys, as well as p and q are very large numbers (usually with a length of around 1024-8192 bits depending on the required strength). The public key can also be smaller in order to speed up the computations. According to Menezes, Vanstone, and Oorschot [34], $e = 3$ is a very common public key.

More information on RSA can be found, amongst others, in Menezes, Vanstone, and Oorschot [34] or Smart [49].

From an analytical point of view, we see that the algorithm itself is very easy, only one equation has to be solved to en-/decrypt data. Since the keys as well as n are very large numbers, they do not fit into a single integer. Hence, a different approach has to be used in order to perform the calculations on a CPU. For Java, a very common approach is to use the `java.math.BigInteger`

```
private void encryptBlock(int[][] KW) {
  int r, r0, r1, r2, r3;

  C0 ^= KW[0][0];
  C1 ^= KW[0][1];
  C2 ^= KW[0][2];
  C3 ^= KW[0][3];

  r = 1;

  while (r < ROUNDS - 1) {
    r0 = T0[C0&255] ^ shift(T0[(C1>>8)&255], 24) ^
        shift(T0[(C2>>16)&255],16) ^ shift(T0[(C3>>24)&255],8) ^ KW[r][0];
    r1 = T0[C1&255] ^ shift(T0[(C2>>8)&255], 24) ^ shift(T0[(C3>>16)&255],
        16) ^ shift(T0[(C0>>24)&255], 8) ^ KW[r][1];
    r2 = T0[C2&255] ^ shift(T0[(C3>>8)&255], 24) ^ shift(T0[(C0>>16)&255],
        16) ^ shift(T0[(C1>>24)&255], 8) ^ KW[r][2];
    r3 = T0[C3&255] ^ shift(T0[(C0>>8)&255], 24) ^ shift(T0[(C1>>16)&255],
        16) ^ shift(T0[(C2>>24)&255], 8) ^ KW[r++][3];
    C0 = T0[r0&255] ^ shift(T0[(r1>>8)&255], 24) ^ shift(T0[(r2>>16)&255],
        16) ^ shift(T0[(r3>>24)&255], 8) ^ KW[r][0];
    C1 = T0[r1&255] ^ shift(T0[(r2>>8)&255], 24) ^ shift(T0[(r3>>16)&255],
        16) ^ shift(T0[(r0>>24)&255], 8) ^ KW[r][1];
    C2 = T0[r2&255] ^ shift(T0[(r3>>8)&255], 24) ^ shift(T0[(r0>>16)&255],
        16) ^ shift(T0[(r1>>24)&255], 8) ^ KW[r][2];
    C3 = T0[r3&255] ^ shift(T0[(r0>>8)&255], 24) ^ shift(T0[(r1>>16)&255],
        16) ^ shift(T0[(r2>>24)&255], 8) ^ KW[r++][3];
  }

  r0 = T0[C0&255] ^ shift(T0[(C1>>8)&255], 24) ^ shift(T0[(C2>>16)&255],
      16) ^ shift(T0[(C3>>24)&255], 8) ^ KW[r][0];
  r1 = T0[C1&255] ^ shift(T0[(C2>>8)&255], 24) ^ shift(T0[(C3>>16)&255],
      16) ^ shift(T0[(C0>>24)&255], 8) ^ KW[r][1];
  r2 = T0[C2&255] ^ shift(T0[(C3>>8)&255], 24) ^ shift(T0[(C0>>16)&255],
      16) ^ shift(T0[(C1>>24)&255], 8) ^ KW[r][2];
  r3 = T0[C3&255] ^ shift(T0[(C0>>8)&255], 24) ^ shift(T0[(C1>>16)&255],
      16) ^ shift(T0[(C2>>24)&255], 8) ^ KW[r++][3];

  C0 = (S[r0&255]&255) ^ ((S[(r1>>8)&255]&255)<<8) ^
      ((S[(r2>>16)&255]&255)<<16) ^ (S[(r3>>24)&255]<<24) ^ KW[r][0];
  C1 = (S[r1&255]&255) ^ ((S[(r2>>8)&255]&255)<<8) ^
      ((S[(r3>>16)&255]&255)<<16) ^ (S[(r0>>24)&255]<<24) ^ KW[r][1];
  C2 = (S[r2&255]&255) ^ ((S[(r3>>8)&255]&255)<<8) ^
      ((S[(r0>>16)&255]&255)<<16) ^ (S[(r1>>24)&255]<<24) ^ KW[r][2];
  C3 = (S[r3&255]&255) ^ ((S[(r0>>8)&255]&255)<<8) ^
      ((S[(r1>>16)&255]&255)<<16) ^ (S[(r2>>24)&255]<<24) ^ KW[r][3];
}

private static int shift(int r, int shift) {
  return (r >>> shift) | (r << -shift);
}
```

Listing 2.6: Bouncy Castle AES implementation excerpt for block encryption. Class: `org.bouncycastle.crypto.engines.AESEngine`

```
private BigInteger publicKey = ...;
private BigInteger privateKey = ...;
private BigInteger modulus = ...;

public BigInteger encrypt(BigInteger plaintext) {
  return plaintext.modPow(publicKey, modulus);
}

public BigInteger decrypt(BigInteger ciphertext) {
  return ciphertext.modPow(privateKey, modulus);
}
```

Listing 2.7: Simple RSA Java implementation using `java.math.BigInteger`.

class, which provides a convenient interface to operate on such big numbers. It provides all functionality needed to perform RSA en- and decryption.
A very simple Java implementation is given in Listing 2.7. The public- and private key, as well as the modulus are given. En- and decryption can then be performed by calling the corresponding methods.

Since n is composed of two prime numbers, the *Chinese Remainder Theorem* can be used in order to improve the performance of the algorithm. For more information on the Chinese Remainder Theorem, we refer to Menezes, Vanstone, and Oorschot [34]. Listing 8.1 of Chapter 8 shows, how it can be implemented in Java.

2.8.3 Hash Functions

As stated in Trappe and Washington [59], cryptographic hash functions take a message with an arbitrary length and produce a fixed-length output, the message digest, also called hash value. Mathematically speaking, given a message m, the hash value $h(m)$ is calculated. Hash functions should have the following three properties:

- **Preimage resistance:**
  Given $h(m)$, it is very difficult to find the message m.
- **Second preimage resistance:**
  Given $h(m)$ and m, it is very difficult to find a second m' with $h(m) = h(m')$.
- **Collision resistance:**
  It should be infeasible to find two different messages m and m' with $h(m) = h(m')$.

Currently used hash functions include the *Secure Hash Algorithm 1* (SHA-1) and its successor SHA-2. We do not delve into the technical details of these algorithms. For this thesis, it is only relevant that the structure of these algorithms is similar to the structure of symmetric block ciphers. They also utilize many shift operations and rotations, as well as bitwise operations, including *XOR* and *AND*. For more details, we refer to Smart [49] and National Institute of Standards and Technology [38].

2.9 Machine Learning

In this thesis, machine learning is used to classify data found in Android applications. To be more precice, we employ *supervised learning* algorithms. As stated in Bishop [2], the training data used for supervised learning algorithms consists of input vectors, as well as their corresponding target values. Using this data, a machine learning model can be trained, which can then be used to classify new instances.

Other machine learning types include *unsupervised learning*, where the algorithm tries to discover structures or groups within given data, without having any target values attached to the training vectors, *semi-supervised learning*, which combines both approaches, or *reinforcement learning*, which tries to find actions to take for a given scenario.

The main focus in this thesis is to detect certain application functionality. Thus, we decided to use supervised learning strategies. As training data, we use applications where we manually label the functionality of their components.
A simple example would be an analysis process, where we want to detect malicious applications. The training data comprises malicious applications labeled “malware”, as well as benign applications labeled “normal”. A machine learning model can then be trained using this data. Then, new applications can be classified using the previously created machine learning model.

A second approach, which could also be used for the *Semantic Pattern Analysis* in future work, is *anomaly detection*, also called outlier detection. The goal of this method is to find outliers – instances that are considered to be abnormal. A common method is to calculate the distance to the average normal data. If this distance is too large, the data can be considered as an anomaly. Other methods include probability densities and other statistical approaches, as explained in Witten, Frank, and Hall [60].

Mitchell [35] gives an overview of several machine learning strategies, including decision tree learning, bayesian learning, and neural networks. For this thesis, we mainly use *Support Vector Machines* (SVMs) for machine learning tasks. A Support Vector Machine is a machine learning algorithm for two classes. It is also possible to use them for more than two classes; according to Bishop [2], multiple two-class SVMs can be combined in this case.
Data points are represented as multi-dimensional vectors. The SVM uses a hyperplane to separate the two classes. Depending on the side the new data point is located at, this point can be labeled accordingly. Besides this linear SVM, it is also possible to use different kernels, like polynomials or sigmoids. Furthermore, soft margins have been added in 1995, which allow errors in the training set.
For more details, we refer to Cortes and Vapnik [7].

For estimating the performance of a machine learning model, k-fold cross-validation can be used. The dataset is randomly split into k partitions. Then, for each of these partitions, the model is evaluated: one partition is used as testing data, the other k-1 partitions are used for training. Then, the results are averaged.
More information on this topic can be found in Bishop [2] and Witten, Frank, and Hall [60].

In this thesis, we utilize *Weka* for all machine learning purposes. Weka is a powerful machine learning framework developed by the University of Waikato [27]. It is implemented in Java and provides convenient APIs to perform various machine learning operations. It includes several implementations of common machine learning algorithms and is used for all machine learning tasks in this thesis. Additionally, the Weka Explorer provides a convenient interface for testing machine learning algorithms as well as for visualization purposes. Detailed information can be

found in Hall et al. [27] and Witten, Frank, and Hall [60].
Furthermore, we use LIBSVM [5], a Java Support Vector Machine implementation. This library is compatible with Weka and can be easily integrated.

2.10 Semantic Patterns

The concept of *Semantic Patterns* has been introduced by Peter Teufl [52]. Semantic Patterns can be used to transform arbitrary features, containing both numeric and symbolic features, to vectors suitable for machine learning algorithms.
By applying the *Semantic Pattern Transformation*, the feature vectors are transformed to Semantic Patterns, simple double vectors used as input for machine learning algorithms.

2.10.1 The Problem

Most machine learning algorithms can solely be applied to vectors containing values, meaning that only *distance-based* feature values (numeric values) can be used for the analysis process. If *symbolic* feature values should be added as well, as for example `feature = myString`, it is not possible to simply supply these mixed features to a machine learning model. In order to use combinations of different feature representations, a preprocessing step has to be performed to create valid input data suitable for common machine learning algorithms.

This preprocessing should also include a *normalization* step for the feature values. Suppose we have two separate double arrays as feature values, one containing very large values and the second containing only very small values; both of these arrays should have an impact on the results. Thus, they have to be normalized accordingly. Different value ranges for different features have to be considered and require pre-processing of the data in order to achieve good results.
Furthermore, missing values have to be handled accordingly.

2.10.2 The Semantic Pattern Transformation

In order to be able to combine both symbolic and distance-based feature values, the *Semantic Pattern Transformation* [54] can be used to convert arbitrary feature combinations to vectors. These vectors can then be used as input for machine learning purposes.

The main idea behind Semantic Patterns is to examine the semantic relations between feature values. An associative network, the so-called semantic network, is created, containing nodes for all feature values. Each node of this network represents a feature value. Weighted links are established between these nodes, which represent the relations between these feature values. Since all features of a given instance belong together, the nodes representing these features are linked, if they have not yet been linked for previous instances, or the link weight is adjusted accordingly, if the two nodes are already connected.

Once this semantic network has been established, the Semantic Patterns can be created. For each feature value of a given instance, the representing node of the network is “activated” and *activation spreading* is performed:
First, an initial activation value is set for each node. For the activated node, all weighted links to other nodes are examined. For each of these connected nodes, a new activation value is calculated according to several pre-defined parameters, the weight of the link, as well as the current activation values of the involved nodes. The resulting new activation value is then assigned to the connected node.

Name	Opcodes	Local variables	Method call count
method 1	IF_NE, CMP	SmsMessage, String	24
method 2	IF_EQ, CMP	SmsMessage, int	26
method 3	ADD, CMP	double, int	15
method 4	ADD, IF_EQ	int, String	26

Table 2.2: Example method features.

This process is repeated for all feature values of the given instance. The resulting activation values are then used for creating the *Semantic Pattern*. This pattern contains simple double values, being the activation values of all nodes, which can then be supplied to various machine learning algorithms as input.

Due to the current structure of the Semantic Pattern Transformation, the feature order does not have an impact on the resulting Semantic Pattern. In future work, this topic could be addressed and the feature order could be considered for the transformation process. The implications this order independence has for the proposed new *Semantic Pattern Analysis* will be discussed later on in this thesis.

2.10.3 Example

Here, we give a small example on how the *Semantic Pattern Transformation* works. Since the goal of this thesis is to classify components of Android applications, this example is also based on Android components. To be more precise, we will show, how features found in four different methods of an Android application are transformed to Semantic Patterns.

In order to keep it simple, this example is based on a simplified version of the *Semantic Pattern Transformation*, which omits several optimizations and calculations.

Suppose we want to convert the features of the methods given in Table 2.2 to Semantic Patterns. For each method, five features are given, two opcodes, two local variables and number of method calls invoked by the current method.

For each of these feature values, a node is created in the associative network. Then, links are established between all features of the same method. If two feature values are already linked, the weight of the link is adjusted accordingly. Figure 2.5 shows the final semantic network with all weighted links, indicated by different line strengths.

Then, the spreading activation process is performed. First, each node gets an initial activation value, which is, in this case, 0. The *Semantic Pattern* for *method 1* is then calculated by applying activation spreading to each feature value. For example, for the first feature, *IF_NE*, the corresponding node is activated. For all nodes linked to *IF_NE*, new activation values are calculated depending on the link weight and the current activation values. This process is repeated for all features of *method 1*. Once this process has been completed, the final *Semantic Pattern* contains all activation values of all nodes.

2.10.4 Applications

This section gives a short overview of applications, where Semantic Patterns have already been deployed.

In Teufl [52], the concept of Semantic Patterns has been evaluated on various data. The framework

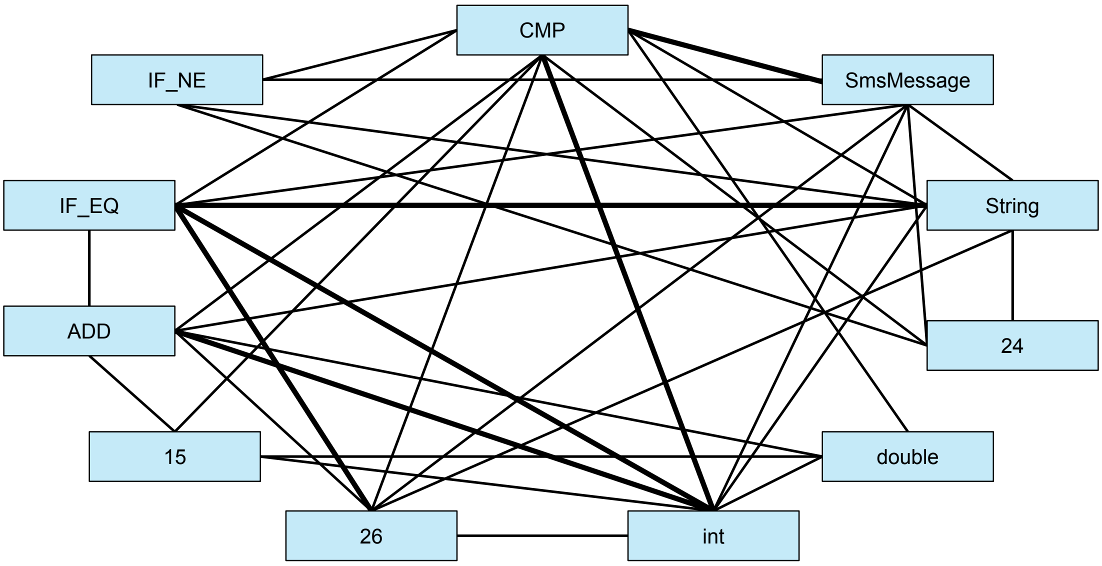


Figure 2.5: Resulting semantic network for the features in Table 2.2. Different line strengths indicate different weights.

has been evaluated with different strategies: Amongst others, both supervised and unsupervised learning strategies have been tested, semantic search has been performed and the influence of missing values has been assessed.

Furthermore, Resource Description Framework (RDF) data analysis has been performed in Teufl and Lackner [53]. The CIA world factbook[25] has been used as RDF data source and the semantic relations between various countries have been evaluated using Semantic Patterns.

In Teufl et al. [55], Semantic Patterns have been used to detect malicious Android application on Google Play. For this analysis, application metadata, i.e., the application description on Google play in combination with the required app permissions, has been used as data source. The Semantic Pattern transformation has been applied on this metadata and various analyses have been conducted, including a semantic search, anomaly detection. Furthermore, the feature relevance and relations between features have been examined.

[25] https://www.cia.gov/library/publications/the-world-factbook/

Chapter 3

Related Work

This chapter presents selected static- and dynamic analysis frameworks for the Android platform. Since the *Semantic Pattern Analysis* presented in this thesis is based on machine learning algorithms, the focus of this chapter will also be on frameworks utilizing various machine learning approaches. First, Section 3.1 presents related static analysis tools. Then, Section 3.2 will cover some dynamic analysis approaches.

3.1 Static Analysis

Static analysis relies solely on examining the application's code without executing it on any device. For Android application packages, this means that the Dalvik bytecode is examined, as well as all other files included in these packages, like the Android manifest, native libraries, or various resource files (see Section 2.3).

One of the first static analysis frameworks has been presented in Schmidt et al. [44]. This framework supports on-device-, collaboration-, or remote analysis. For this analysis, ELF (Executable and Linking Format) objects have been analyzed. They used all system- and function calls invoked by these ELF file (by using the *readelf* command) to classify the executables using machine learning.
The *Semantic Pattern Analysis* proposed in this thesis operates on Android application packages instead of ELF files. Thus, the features used for the analysis process are also different. For the Semantic Pattern Analysis, we utilize various features found in the Dalvik executable as opposed to system calls found in the ELF files.

Shabtai, Fledel, and Elovici [46] showed that machine learning can be used to classify Android applications. They use features found in the Dalvik executable and in various XML files to determine whether a given application is a *game* or a *tool*. They evaluated different feature selection algorithms and different machine learning strategies. In addition, they state that this approach could also assist in detecting malware.

In Sanz et al. [43] automatic application categorization has been performed as well. Instead of two application categories, the following seven categories have been used: Communication, Entertainment, Tools, Multimedia and Video, Productivity, Brain & Puzzle, and Society. Compared to Shabtai, Fledel, and Elovici [46], they used different features, namely the defined strings in the application, required permissions, the rating, the number of ratings, and the size of the application. Various machine learning algorithms, including Bayesian networks, decision trees, and SVMs.

Compared with the Semantic Pattern Analysis, these two frameworks use different feature compositions. The approach of Shabtai et al. also uses features found in various XML files, the approach by Sanz et al. uses the rating and other metadata of a given Application. For our analysis process, we focused solely on the Dalvik executable and the Android manifest. Furthermore, we combine numeric- and symbolic features, including opcode histograms, local variables, or method calls. Depending on the targeted functionality of our analysis plugins, we utilize different feature sets and filtering steps customized specifically for the given analysis. Moreover, these two frameworks operate application-wide. The Semantic Pattern Analysis is designed to pinpoint certain application functionality within applications by separately analyzing their components.

Ghorbanzadeh et al. [18] use the permissions of Android applications to estimate the application categories. Applications from 34 different categories have been used to train a neural network. In addition, they created suspicious applications by tampering with their required permissions, which can then be detected by their neural network.
In theory, we could implement a similar analysis plugin that uses solely the application permissions. These permissions would then be transformed to Semantic Patterns and classified using machine learning algorithms.

In Felt et al. [15], a framework called Stowaway has been presented, which detects overprivileged applications – applications that do not actually require all permissions defined in their Android manifest. They came to the conclusion that about one third of all tested applications are overprivileged. Furthermore, they created a mapping between Android API calls and their required permissions. For Semdroid, this Android permission map is used to obtain component-wise permission requirements. This information can then be used by the analysis plugins, for example for filtering operations or as features.

DroidMat [61] is a static malware detection framework. As features, they use the permissions and the components defined in the Android manifest, information found in intents, API calls and the components these calls are invoked from, and inter-component communication. They use clustering algorithms and the k-nearest neighbor algorithm to classify their data.
This framework uses a clustering approach, as opposed to the classification algorithms used for the Semantic Pattern Analysis, like SVMs.

Besides machine learning, there are also some interesting other approaches. For example, RiskRanker [25] assesses possible security risks in Android applications and categorizes them accordingly. Chin et al. [6] examined inter-app communication and present potential attacks. Another interesting framework is SCanDroid [16], which tries to assess the security of Android applications by examining the data flow.

Androguard [1] is a static analysis tool that can be used for reverse engineering purposes, as well as for malware detection. The framework is written in Python and allows to disassemble and decompile Android application packages. Furthermore, they provide an open source database of known Android malware, as well as a risk indicator for malicious applications. It offers convenient APIs to integrate new static analysis tools.

As demonstrated in Georgiev et al. [17], SSL certificate verification is often performed incorrectly – even in widely spread libraries utilized by many security-critical applications. Man-in-the-Middle (MITM) attacks can be mounted on these systems, which pose dangerous threats to these systems.
Similarly, Fahl et al. [14] presented an analysis tool called *MalloDroid*, which aims at detecting SSL vulnerabilities in Android applications. They used their tool to analyze 13.500 popular free applications on Google Play. In 1.074 of these applications, they found SSL/TLS vulnerabilities that could be exploited by using MITM attacks.

Egele et al. [12] showed that many Android applications do not correctly use the cryptographic

APIs provided by the Android operating system. Their tool, *CryptoLint*, uses static code analysis to track the usage of these cryptographic APIs. A set of rules is then use to detect possible programming mistakes. Using these rules, they analyzed 145.095 applications found on Google Play. Out of these applications, 15.134 utilize the cryptographic APIs provided by the Android platform – but 11.748 of these applications violated at least one CryptoLint rule. Thus, only 1.421 of these application implemented the cryptography correctly.
Compared to the machine learning approach of the Semantic Pattern Analysis, CryptoLint is based on type analysis, super control flow graph extraction, and static slicing techniques in order to determine the parameters the cryptographic APIs have been invoked with. Their framework has been built on top of Androguard.

Judging from the results of the last three frameworks, it seems that many applications do not implement and use cryptography correctly. All of these frameworks checked the cryptographic APIs provided by the Android framework. With the Semantic Pattern Analysis, it is possible to also detect *custom* cryptographic implementations. In future work, this cryptographic code could then be further analyzed using similar static analysis approaches – for example, by adding additional Semdroid analysis plugins.

3.2 Dynamic Analysis

For dynamic application analysis, there are also several frameworks available. Compared to the static analysis approach used for the *Semantic Pattern Analysis*, the applications have to be executed in an actual Android environment while the Android system is monitored closely.

TaintDroid [13] such a dynamic framework, which has been released in 2010 and which utilizes a modified Android distribution for real-time taint tracking. They trace sensitive data throughout the system and perform real-time monitoring of the current device state. For TaintDroid, the Android framework including the Dalvik VM has been instrumented to perform detailed system logging and tracking of sensitive data. A custom Android image has to be built and installed on the device in order for TaintDroid to function.

DroidBox [11] is another popular dynamic analysis framework based on TaintDroid. According to the project page, it allows to monitor the network usage and file read and write operations. In addition, it is possible to analyze information leaks via the network, files and SMS messages, to list broadcast receivers, and to monitor sent SMS messages as well as phone calls. It is possible to monitor cryptographic API calls, to detect circumvented permissions, and to monitor started services and classes loaded through the *DexClassLoader*.

Furthermore, they also offer the DroidBox *APIMonitor*, which directly instruments Android application packages. Thus, these instrumented applications can be executed in a standard Android environment. Monitoring code is directly injected in the Dalvik executable and the built-in Android logging mechanism is used to retrieve the results.

Another dynamic analysis framework is *DroidScope* [64]. This analysis framework is based on a modified Android emulator and allows to monitor information leakage through both Java- and native code.

Dynamic analysis paired with machine learning is utilized by *Andromaly* [47]. Thsi framework closely monitors the current system metrics of the Android device. By using machine learning algorithms, Andromaly decides whether the currently active application is malicious.

Crowdroid [3] utilizes a client-server architecture for malware detection. They employ a behavioral approach. First the Crowdroid Android application monitors all system calls for a given application. Then, these system-call vectors are sent to a server, where clustering algorithms are used to classify the application.

In Zhao et al. [66] and Zhao et al. [67] SVM-based learning algorithms have been used for malware classification. Using this approach, they were able to create a detection system with a high detection rate and a low rate of false positives.

MADAM [10] is another dynamic analysis framework, which monitors the Android system at kernel-level and at user-level. They log system calls, the phone state (idle or not), and the number of SMS messages sent in a given time interval. Again, machine learning alrogithms are used to detect malicious activities.

The power consumption of Android devices has been used for application classification and malware detection in Zefferer et al. [65]. First, the current power consumption has been captured directly on the Android device. Then, the power traces can be classified using machine learning algorithms. With this approach, it is possible to categorize the application. Categories used in the paper are games, internet, idle, malware, music and multimedia and the accuracy lies at roughly 90%. Furthermore, by monitoring the power consumption for a short time after an SMS message is received, it is possible to state whether the application has been handled normally, or if suspicious operations, like handling a SMS command, have been performed.

Chapter 4

Semdroid – An Introduction

This chapter presents the proposed new static Android application analysis framework, called *Semdroid*. Starting with an overview in Section 4.1, the basic analysis workflow is dissected and each step of this process is elaborated in detail. The results of the analysis process are discussed in Section 4.5. Then, details about the deployment of Semdroid are presented in Section 4.8.

Furthermore, Semdroid provides an *evaluation framework* used to determine the performance and accuracy of a given analysis, which is described in Section 4.6. Some analysis processes, for example, based on machine learning algorithms, require a training step, where pre-classified training data is used to create a machine learning model. For this case, Semdroid provides a convenient training interface that aids the developer in creating new training processes. Section 4.7 gives an overview of this interface.

In this chapter, analysis plugins are considered to be black boxes capable of analyzing given applications. No actual analysis approaches are discussed for now. For more information on the proposed new *Semantic Pattern analysis*, see Chapter 6.

4.1 Overview

Semdroid is a static Android application analysis framework capable of detecting application functionality within Android applications and labeling the corresponding components accordingly. This framework, which has been implemented in Java, is very flexible and can be used for various analysis- and classification tasks. Multiple analysis plugins can be employed that analyze *Android application packages* (`.apk` file, see Section 2.3) and generate application analysis reports.
Since Semdroid is not limited to a particular static analysis type, any static analysis approach can be used. For example, we have created a simple analysis plugin that list all methods that invoke certain API calls, as well as sophisticated analysis processes, like the proposed new *Semantic Pattern Analysis*.

Figure 4.1 gives an overview of the application analysis workflow. Suppose a given Android application package should be analyzed. First, an *App object* is created, which represents the data found in the application package and holds all necessary information required by the analysis processes. Basic *global filtering* operations are performed to remove unwanted information, as for example certain common library classes or methods, or irrelevant Dalvik opcodes.

Then, a *test suite* containing multiple analysis plugins is used to classify the application. The App object is the input for each analysis. Each analysis then examines the contents of the App object and produces an *application analysis report* containing all results. The test suite bundles

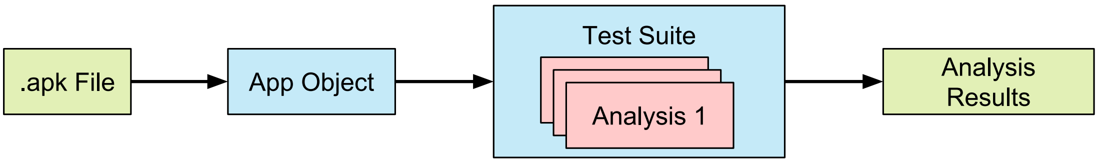


Figure 4.1: Basic Semdroid application analysis. The given Android application package file is parsed, an App object is created, which represents the contents of the `.apk` file and a test suite containing multiple analysis plugins examines the *App object*. Finally, the results are returned.

all results and returns them for further processing. Depending on the deployment method, these results are then post-processed, saved, or displayed.

4.2 Input

The input for the Semdroid framework are Android application packages (`.apk` files). The structure of these files has been explained in Section 2.3. It is also possible to directly supply Dalvik executables, the `classes.dex` files found in Android application packages. Furthermore, `.jar` files can be used as well. They will be converted to Dalvik executables and then processed like any `.dex` file.

Some analysis plugins require additional information included in Android application packages, which is not stored in the `.dex` file. For example, the `AndroidManifest.xml` can be used to determine the required permissions or the defined application components. Hence, for `.apk` files, we also extract the contents of the Android manifest. If the file under analysis is a `.jar` or `.dex` file, this information is not available, which could affect the analysis results for analyses that require this additional information.

4.3 App Object

The *App object* represents the extracted contents of the input file discussed in the previous section. This intermediate representation is a core component of Semdroid since it is used as input for all analysis processes. It holds all necessary information required for application analysis.

Basically, for this representation, the application structure is reconstructed, as all implemented classes and methods are listed, and links between all components are established. This call-graph-like structrue allows to easily trace methods and method calls, or to traverse all classes of the application or all methods of a given class. Furthermore, the Dalvik bytecode is pre-processed; maps and lists are created for used local variables and method calls, the required permissions are recorded and all Dalvik opcodes are listed. Additional data found in the Android manifest (Section 2.4) is included as well. Since `.dex` and `.jar` files do not contain an Android manifest, this additional data cannot be added in this case.

A simplified structure of the App object is shown in Figure 4.2. Some objects have been omitted to keep the figure simple. The App object contains a list of *DexClasses*, one per available class, a Manifest object, which represents the `AndroidManifest.xml`, as well as additional application information. The Manifest object contains all entries according to Section 2.4, which are not shown in Figure 4.2.

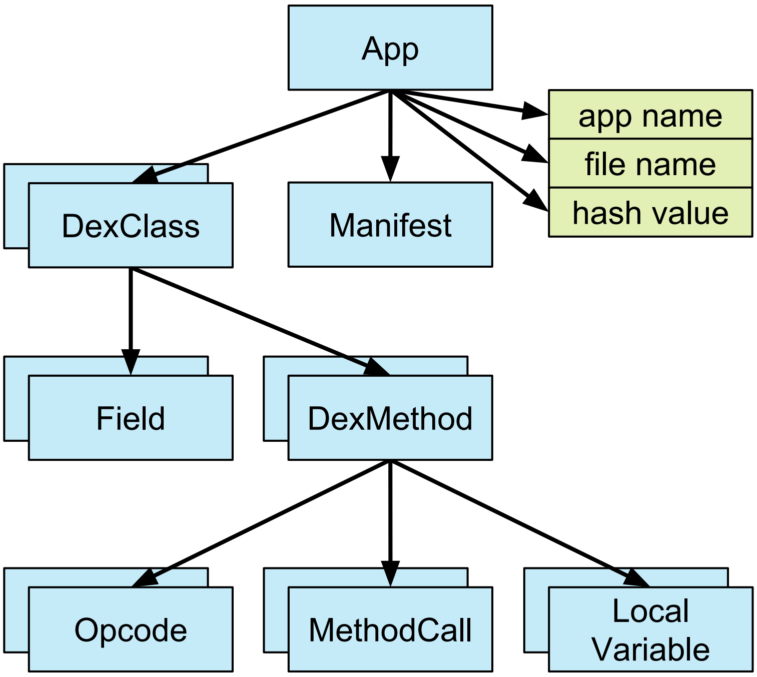


Figure 4.2: Structure of the App object, which represents the `.apk` file. It includes information of the Android manifest and the Dalvik bytecode. All classes and their implemented methods and fields will be parsed, including all operations, method calls, and local variables for each method.

Each DexClass holds, amongst others, all its fields and implemented methods. In addition, a link to the App object is established and additional information is added. For example, the superclass for the given class is determined. If this superclass is implemented by the application, the DexClass reference is added, if it is an external class, the name of this external class is saved. For any given class it is also possible to get the first external superclass. The superclass hierarchy is traversed until a class is found that is not implemented by the application itself. This can be useful for various analyses: For example, for detecting SMS functionality one has to examine classes that extend the `BroadcastReceiver`-class provided by the Android platform.

DexMethods contain the "main features" used by most analysis plugins: a list of all opcodes, method calls, and local variables. For *method calls*, the corresponding DexMethod is linked if the given method is implemented by the application itself. If the method call is an API call to external classes, the full class- and method names are available. The same holds for local variables where the DexClass or the full class name is linked, depending on whether the class is implemented by the application. For each method call, additional information is stored as well, like the Java footprint of the method, input parameters, the return value, or the required permissions for API calls invoked from within the method. *Local variables* include the variable type, name, and signature.

Furthermore, it is possible to query all data, i.e. opcodes, method calls, and local variables, of a given DexMethod, *including the data of nested method calls*. The inclusion depth for the method calls can be specified by the developer. Suppose we have a method A that calls a method B. If we want to acquire all opcodes of method A and specify a method call inclusion depth of 1, all opcodes of both method A and method B will be returned. The opcodes of method B will be inserted at the correct list position, right where method B is called in method A.

Once the App object has been created, it will be handed to the test suite. The original input file will not be touched by any other component. Thus, all information required for analysis must be included in this object.

4.4 Test Suite

All analysis plugins are bundled in a *test suite*. The App object under analysis is handed to this test suite, which then passes it on to all analysis plugins and gathers the analysis results. Once all analyses are finished, the test suite bundles all results and returns a final test suite report. This final report is then be used for further processing, as for displaying it to the user.

Each analysis examines the contents of the App object and, based on this information, creates an application analysis report. Typically, each analysis performs additional filtering steps tailored specifically for the given use case. The application components are then analyzed and labeled accordingly. More details on the analysis results as well as the labeling process can be found in Section 4.5.

Analyses do not have to be based on machine learning, any static analysis can be used to classify applications. They just have to return an application analysis report. For the framework, it does not matter how this report is generated, as long as the analysis can be performed by statically examining the contents of the App object. For example, simple static analysis approaches that just check if certain API calls are performed can be used as well as sophisticated analyses based on complex feature extraction mechanisms or machine learning algorithms like the *Semantic Pattern Analysis* discussed in Chapter 6.

4.5 Results

Each analysis produces an application analysis report. This report contains all analysis results, including labels for all analyzed application components. One advantage of the framework is, that analysis plugins can specify the exact location where they suspect a certain device functionality. For example, an analysis that detects cryptographic code can specify the methods or classes of the application that implement cryptographic functionality.

A labeling concept is employed that allows analyses to tag application components. Currently, these components are the following parts of an App object: the whole *application*, *classes*, and *methods*. Depending on the analysis, different components of the application can be chosen. For example, analysis approaches that determine the application category usually label the whole application, while an analysis that detects cryptographic code could operate on a method- or class-level. It is also possible to combine these component types and label, for example, both selected methods as well as classes.

An analysis can create arbitrary labels with a distinctive name and attach application components to these labels. Furthermore, the labels are also directly attached to the corresponding components. Components are not limited to a single label. Multiple labels can be added to the same component, if desired. The application analysis report produced by the analysis contains all found labels and their attached components. Since a test suite can contain multiple analysis plugin, the global test suite report contains a list of all application analysis reports, one per plugin.

Basically, there are two ways to traverse the resulting datasets: First, it is possible to traverse each analysis report of each analysis, each label of each report, and finally each component attached to the label. Second, the App object itself can be traversed component by component and all attached labels can be displayed.

This component-based labeling process also helps to get a quick and rough estimate of the analysis performance. Judging by the component names, the credibility of the analysis results can be assessed. For example, if an analysis looks for symmetric block ciphers and adds the label *"symmetric block cipher"* to a method called *"com.symmetric.Cipher: void encryptBlock(byte[] block)"*, it is quite likely that the classification could be correct. When creating a new analysis, taking a glance at the component names helps to get a first idea of the analysis performance. However, since these component names do not actually represent their implemented functionality, these results should be used with caution. For accurate results, the actual code of these components has to be manually checked. Furthermore, if obfuscation techniques are used by the application under analysis, method names could be random strings, which renders this method useless.

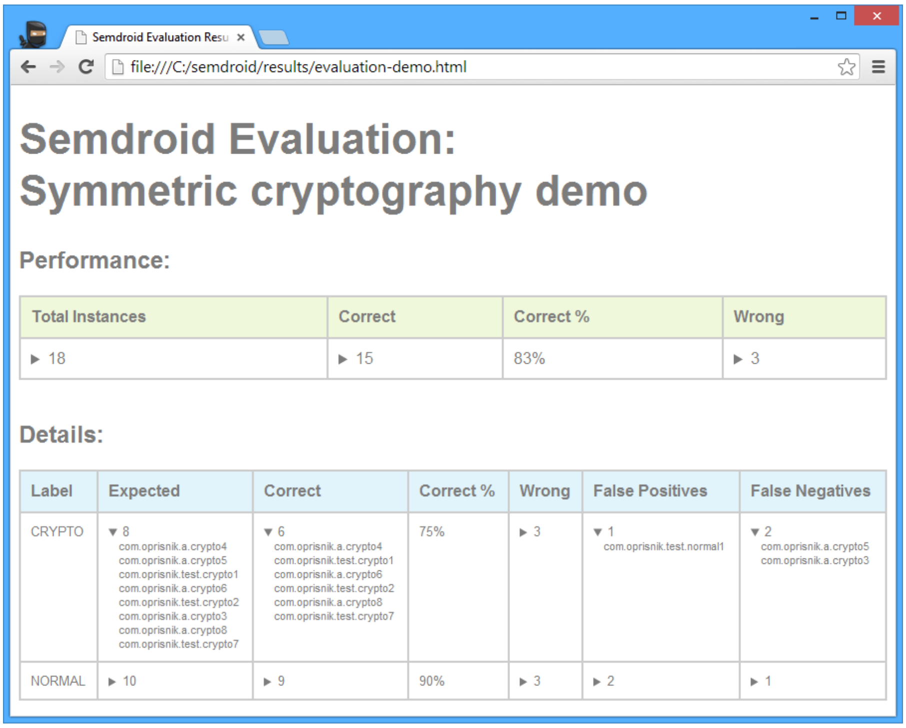
4.6 Evaluation

Semdroid also provides an evaluation framework that can be used to measure the performance of a given analysis. In order to perform such an evaluation, two additional parameters have to be provided: an evaluation dataset and a reference analysis. The evaluation dataset consists of multiple applications (`.apk`, `.dex` or `.jar` files) to be used for the evaluation process. The reference analysis is used to pre-analyze all applications of the evaluation dataset. For each of these applications, a reference application analysis report is produced, which is then compared with the report created by the analysis under evaluation. This reference analysis can be any standard Semdroid analysis. It is also possible to omit the reference analysis and to directly supply a list of reference application analysis reports, one for each application in the evaluation dataset.

The resulting evaluation report contains various performance details: the number of correctly classified components, incorrectly classified components, and the detection rate. Furthermore, additional information is added for each label. The same parameters that are calculated globally are also calculated label-wise. In addition, the number of false positives as well as false negatives is given. For each entry, the affected components are listed: for example, all correctly classified instances are attached to the "correct" section. The report is saved in both HTML and XML format. Figure 4.3 shows such an HTML evaluation report for a demo analysis that detects symmetric cryptography.

4.7 Training

Some analysis techniques, like the proposed *Semantic Pattern analysis* (see Chapter 6), or other techniques based on machine learning, require a training process. Similar to the evaluation process (Section 4.6), a given training dataset (i.e. a folder containing multiple `.apk`,`.dex` and `.jar` files) is used in conjunction with a training analysis to create a new analysis. The training analysis, which is, similar to the reference analysis used for the evaluation process, a standard Semdroid analysis. It is used to add reference labels to the components of all training applications. Based on this pre-labeled App objects, the analysis-specific training process can commence, which has to be implemented by the developer of the analysis. Once this application-specific training step is complete, the new analysis and all required files will be automatically created by the Semdroid training framework. More information on the specific training process for the Semantic Pattern Analysis can be found in (Section 6.8).



Semdroid Evaluation: Symmetric cryptography demo

Performance:

Total Instances	Correct	Correct %	Wrong
▶ 18	▶ 15	83%	▶ 3

Details:

Label	Expected	Correct	Correct %	Wrong	False Positives	False Negatives
CRYPTO	▼ 8 com.oprisnik.a.crypto4 com.oprisnik.a.crypto5 com.oprisnik.test.crypto1 com.oprisnik.a.crypto6 com.oprisnik.test.crypto2 com.oprisnik.a.crypto3 com.oprisnik.a.crypto8 com.oprisnik.test.crypto7	▼ 6 com.oprisnik.a.crypto4 com.oprisnik.test.crypto1 com.oprisnik.a.crypto6 com.oprisnik.test.crypto2 com.oprisnik.a.crypto8 com.oprisnik.test.crypto7	75%	▶ 3	▼ 1 com.oprisnik.test.normal1	▼ 2 com.oprisnik.a.crypto5 com.oprisnik.a.crypto3
NORMAL	▶ 10	▶ 9	90%	▶ 3	▶ 2	▶ 1

Figure 4.3: Evaluation results in HTML format. Detailed global evaluation results as well as label-wise performance information are presented. The analysis under evaluation detects cryptographic code and labels it *CRYPTO*.

4.8 Deployment

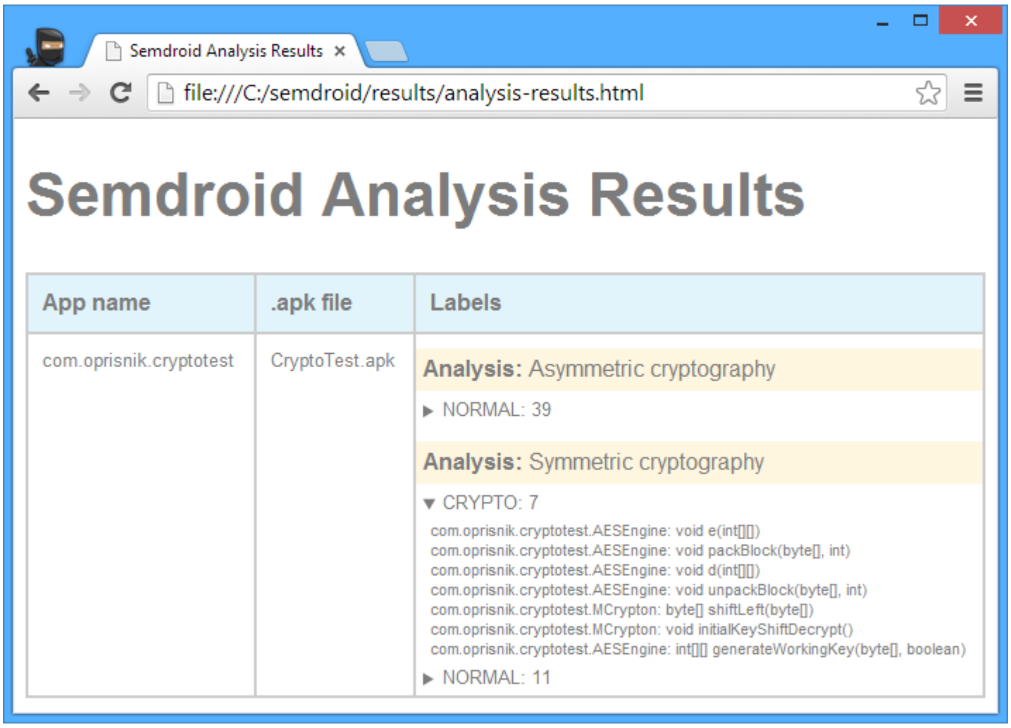
This section presents different possibilities on how Semdroid can be deployed. Currently, there are two possible ways to use Semdroid: on a personal computer or directly on an Android device. Section 4.8.1 give details of the first deployment method and Section 4.8.2 explains on-device analysis.

Additional deployment methods include a hybrid on-device approach and a web service. For the hybrid on-device analysis, fingerprints have to be calculated for a given application, which are then sent to a server. The server analyzes these fingerprints and returns the results back to the device. This hybrid approach has not yet been implemented, but could be addressed in future work. Furthermore, it would be possible to set up a web service that allows to upload and analyze applications and to display the analysis results. This deployment would require a web server that initiates a Semdroid analysis for the uploaded file. The analysis results are already converted HTML files, which could be displayed to the user.

4.8.1 Personal Computer

Semdroid can be used on a personal computer. A command line interface is available that can be used to analyze applications or folders containing multiple applications, to start the training process, and to evaluate a given analysis. More information on the training and evaluation process are given in Sections 4.6 and 4.7.

The analysis results are saved as both XML and HTML files. Figure 4.4 shows the resulting HTML results for two analyses. A list of all analyzed applications is given, and for each analysis,


Semdroid Analysis Results
file:///C:/semdroid/results/analysis-results.html

Semdroid Analysis Results

App name	.apk file	Labels
com.oprisnik.cryptotest	CryptoTest.apk	**Analysis:** Asymmetric cryptography ▸ NORMAL: 39 **Analysis:** Symmetric cryptography ▾ CRYPTO: 7 com.oprisnik.cryptotest.AESEngine: void e(int[][]) com.oprisnik.cryptotest.AESEngine: void packBlock(byte[], int) com.oprisnik.cryptotest.AESEngine: void d(int[][]) com.oprisnik.cryptotest.AESEngine: void unpackBlock(byte[], int) com.oprisnik.cryptotest.MCrypton: byte[] shiftLeft(byte[]) com.oprisnik.cryptotest.MCrypton: void initialKeyShiftDecrypt() com.oprisnik.cryptotest.AESEngine: int[][] generateWorkingKey(byte[], boolean) ▸ NORMAL: 11

Figure 4.4: Analysis results on a personal computer. Two analyses have been performed that are able to detect asymmetric and symmetric cryptography. The symmetric analysis found 7 cryptographic methods and 11 normal ones.

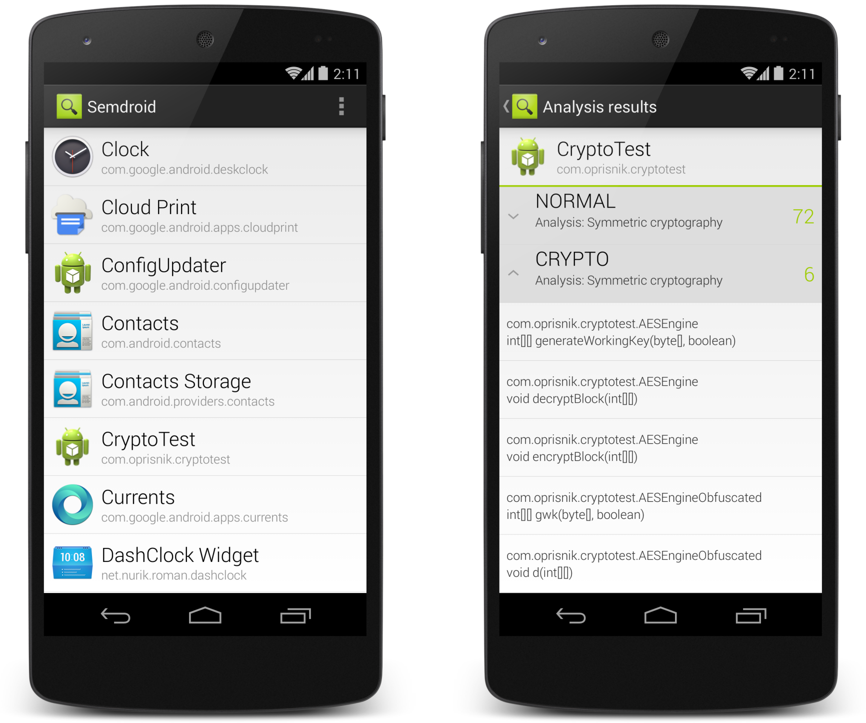
the results are displayed in the “Labels” column. By clicking on a given label, the associated components are listed.

4.8.2 Android Device

A standalone Semdroid Android application is available, which can be installed on any Android device. In order to add a new analysis plugin, the corresponding analysis files have to be placed in a pre-defined folder on the device. The new analysis is then automatically recognized by the Semdroid Android application.

The main screen of Semdroid lists all installed applications, as depicted in Figure 4.5 on the left side. By selecting an application, a dialog window will be shown that asks the user to pick an analysis. The selected analysis will then analyze the Android application package. The right picture in Figure 4.5 shows the analysis results for a given analysis: a list of all labels is displayed and for each label, the associated components are given.

In theory, it is also possible to create and train new analysis plugins directly on the device (Section 4.7). The training process usually requires a rather large training dataset (i.e. many Android applications). Since the performance on a mobile device is considerably lower than on a state-of-the-art personal computer, this training process can be very time consuming. The same applies to the evaluation framework (Section 4.6) where large evaluation datasets can be used to determine the analysis performance. In addition, for the evaluation process, each application has to be analyzed at least twice – once by the reference analysis and once by each analysis under evaluation. Thus, it is not really practicable to create or evaluate analysis plugins directly on the device. For this reason, these two features are currently not accessible from within the Semdroid



2:11

Semdroid

Clock
com.google.android.deskclock

Cloud Print
com.google.android.apps.cloudprint

ConfigUpdater
com.google.android.configupdater

Contacts
com.android.contacts

Contacts Storage
com.android.providers.contacts

CryptoTest
com.oprisnik.cryptotest

Currents
com.google.android.apps.currents

DashClock Widget
net.nurik.roman.dashclock

2:11

Analysis results

CryptoTest
com.oprisnik.cryptotest

NORMAL 72
Analysis: Symmetric cryptography

CRYPTO 6
Analysis: Symmetric cryptography

com.oprisnik.cryptotest.AESEngine
int[][] generateWorkingKey(byte[], boolean)

com.oprisnik.cryptotest.AESEngine
void decryptBlock(int[][])

com.oprisnik.cryptotest.AESEngine
void encryptBlock(int[][])

com.oprisnik.cryptotest.AESEngineObfuscated
int[][] gwk(byte[], boolean)

com.oprisnik.cryptotest.AESEngineObfuscated
void d(int[][])

Figure 4.5: Semdroid Android application. The left image shows a list of all installed applications. By clicking on an application, the analysis process will be initiated and the user is asked to select an analysis to be performed. The right image shows the analysis results.

Android application.

4.9 Semdroid Conclusions

In this chapter, an overview of Semdroid, the proposed new static Android application analysis framework, has been given. First, Android application packages are parsed and application representation objects, the so-called App objects, are created. Global pre-filtering operations are applied to remove unwanted information. Test suites consisting of multiple analysis plugins are then used to analyze the App objects. Analyses can utilize any static analysis approach. Typically, they use the data found in the App object to create an application analysis report. These reports contain all analysis results and links to the corresponding application components. All analysis results are bundled in a test suite report, which is then used for further processing.

Semdroid can be deployed on both personal computers and Android devices. For on-device analysis, the Semdroid Android application is available. It lists all installed applications, which can then be analyzed. Once the analysis is completed, the results will be displayed. On personal computers, a command line interface can be used to analyze given applications. The resulting analysis reports will be saved in both HTML and XML format and contain detailed analysis results. Furthermore, an evaluation framework is available that can be used to determine the performance and analysis accuracy. A training framework, which assists in creating new analysis plugins that require a training process, is available as well.

The next chapter delves into the architectural details of the Semdroid framework.

Chapter 5

The Architecture of Semdroid

In this chapter, we present the *architecture* of Semdroid. The previous chapter outlined the basic functionality of the framework and gave an introduction to the capabilities of the system. Now, an architectural overview of the implemented system will be presented and the main components of Semdroid will be outlined. First, an introduction to all components will be given in Section 5.1. Subsequent chapters are then following up on the details of these components. Finally, Section 5.8 presents the architecture of the Semdroid Android application.

5.1 Component Overview

The basic concepts behind Semdroid have already been described in the previous chapter. An architectural view of the whole system can be found in Figure 5.1. Currently, there are three main tasks that can be performed: application analysis, evaluation, and generating (training) new analysis plugins.

In any case, the first step for each application is to create the *App object* (Section 4.3), which is an intermediate representation of the Android application. The *AppParser* is used to perform this task: It parses the contents of the Android application package and outputs the resulting App object. Section 5.2 gives more details on the architecture of the *AppParser*.

The *TestSuite* component is responsible for analyzing new applications and for assembling the resulting analysis report. It initializes all analysis plugins, calls the *AppParser* to preprocess the Android application packages and then schedules the application analysis process. Similar to that, the *Evaluation* component is used to analyze evaluation applications. But in addition to the analysis plugins under evaluation, a reference analysis is added that creates a reference application analysis report. This reference report is then compared to the reports generated by the analysis plugins and the evaluation results are calculated. For both of these processes, the results are then transformed to various output representations by using different transformers: one for XML, one for HTML, and a third one for direct console output. The *Training* module utilizes a training analysis to pre-classify training applications. Then, the new analysis is trained using this data and the resulting model is saved.
In upcoming sections, these components will be discussed in more detail.

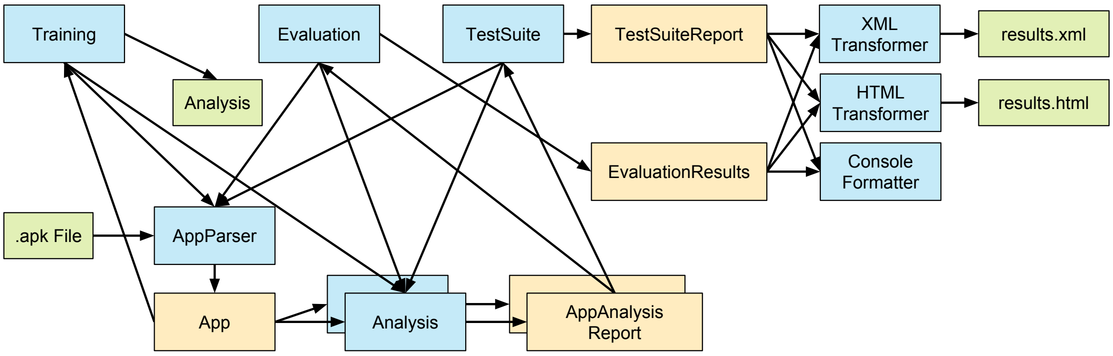


Figure 5.1: Basic Semdroid architecture. The main components used for application analysis as well as for the training- and evaluation process are shown. Input- and output objects are marked green, framework components blue, and intermediate objects have a yellow background.

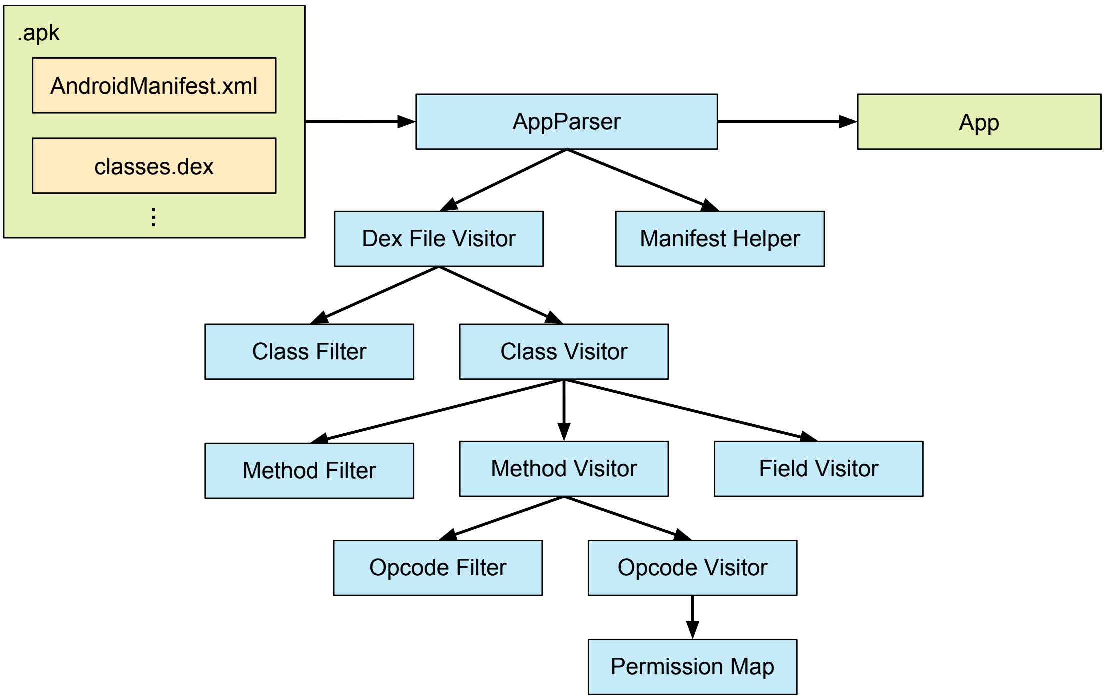


Figure 5.2: Application parsing. The application parser takes an `.apk` file, extracts relevant information, performs global filtering operations, and returns an `App` object.

5.2 App Parsing

The *AppParser* is responsible for extracting information from an Android application package and to bundle this information in an App object. Figure 5.2 shows the architecture behind this process. Section 2.3 outlined the contents of Android application packages. For Semdroid, two files of these packages are currently parsed: the `AndroidManifest.xml` and the `classes.dex`.

First, a manifest helper is used to parse and pre-process the Android manifest, which results in a *Manifest* object holding all of its contents (e.g., permissions, activities, services, broadcast receivers, and their intent filters). Since it is also possible to directly supply `.dex` or `.jar` files instead of APKs, the *Manifest* object cannot be created in these cases.

The second file to be parsed is the `classes.dex`, which contains the Dalvik bytecode. In order to traverse the Dalvik executable, we modified the *dex2jar* [9] library to suit our needs: By using custom visitors for all application components (classes, methods, fields, and opcodes), the application structure is reconstructed and links between these components are established. The resulting structure of the App object has already been described in Section 4.3 and can be seen in Figure 4.2.

Moreover, a pre-filtering step is performed that removes irrelevant data. This pre-filtering step can be adjusted to omit selected opcodes and to filter unnecessary classes, methods, and method calls by setting up white- and blacklists. As depicted in Figure 5.2, a class filter specifies whether a given class should be added. Similar to that, method- and opcode filters decide whether given methods and opcodes should be added to the App object. For instance, this can be used, to ignore common libraries. This pre-filtering process is performed globally, since it influences the structure of the App object handed to all analysis plugins as input. Hence, these filters have to be chosen carefully; all important data required by any analysis has to be included in the App object. Each analysis typically performs additional filtering operations to retrieve analysis-relevant data only.

As explained in Section 2.1.3, the Android operating system employs a permission system to enforce access to sensitive data and device functionality. Since these permissions have to be defined globally for the application, it is not possible to get a component-based mapping of required permissions. Therefore, Felt et al. [15] created a permission map that maps Android API calls to their required permissions. Semdroid utilizes this permission map to determine the required permissions for each component of the application. The opcode visitor depicted in Figure 5.2 has access to this permission map. For each API call, the required permissions are then recorded and attached to the component.

5.3 Analysis

Analyses are the core components of the system that perform the actual analysis operations. Figure 5.3 shows the input and output for a given analysis process. The analysis itself is considered to be a black box that takes as input the App object and outputs a so-called *AppAnalysisReport*. This report contains a list of labels, where each label represents a certain application functionality. Each label contains a list of components that are considered to have the functionality specified by the label. Furthermore, the analysis should also add the label to the application component itself. Internally, additional mappings between labels and components are established that speed up the labeling process. The analysis report also contains additional information, like a link to the application the report belongs to, and the name of the analysis that created the report.

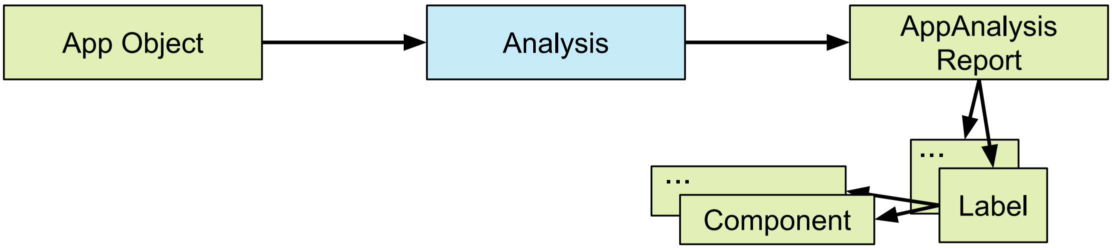


Figure 5.3: Analysis black box. An *App object* is the input for the analysis. The analysis has to produce an *AppAnalysisReport* containing all resulting labels. Each label can have multiple components attached.

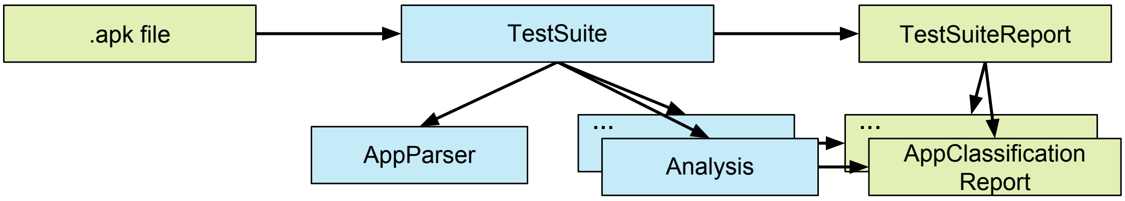


Figure 5.4: Test suite architecture. A given Android application package to be analyzed is passed to the test suite. The *AppParser* creates the App object, which is then handed to all analysis plugins. Each analysis creates an *AppClassificationReport*, which are then added to the final *TestSuiteReport* returned by the test suite.

5.4 Test Suite

The test suite handles the analysis process. It initiates the application parsing process, coordinates all analysis plugins and collects the results, as explained in Section 4.4. In Figure 5.4 a more detailed view of the test suite is presented. A given Android application package (again, either an `.apk`, `.dex`, or `.jar` file) is passed to the test suite. The *AppParser* (Section 5.2) is responsible for creating the App object. Once this object has been created, the test suite initiates all analysis plugins. Each analysis then analyzes the App object and returns the results according to the previous chapter. The test suite collects all results in a *TestSuiteReport*, which is then used for further processing, as for saving it in different formats. Again, additional information, like a link to the analyzed App object and the name of the test suite, is added to the report as well.

5.5 Evaluation

The evaluation framework can be used to measure the performance of an analysis. The structure of this framework is similar to the structure of a test suite and can be found in Figure 5.5. Basically, the evaluation framework extends the test suite framework. In addition to the app parser and a list of analysis plugins under evaluation, a reference analysis is added and for each analysis, an *Evaluator* is added that records the performance of the given analysis.

The reference analysis creates a reference analysis report. The *Evaluator* then compares the report of the analysis under evaluation with this reference report. It records the performance of the analysis both globally and label-wise. The global analysis performance includes the total number of analyzed components, as well as the number of correctly and wrongly classified components. Label-wise, the same parameters are recorded, but in addition, the number of false positives and

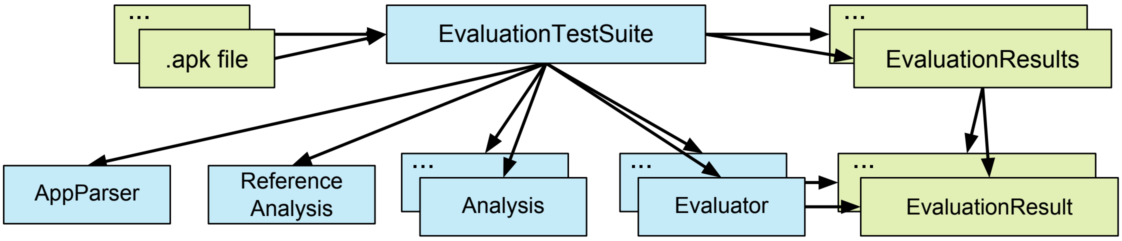


Figure 5.5: Evaluation architecture.

false negatives is calculated as well. In addition, the components belonging to each of these values are also linked. After all training applications have been evaluated, the *Evaluator* returns the final evaluation results for each analysis. Each of these *EvaluationResults* contains the global evaluation results as well as a list of *EvaluationResult* objects, each of which represents one label.

5.6 Result Transformation

Both the test suite results and the evaluation results can be transformed to different formats. As shown in Figure 5.1, there are three different transformations available: XML, HTML, and direct console output. Figures 4.3 and 4.4 show examples for the evaluation- and test suite results in HTML format. The XML formatter converts the results to an XML file that could be used for further processing. The HTML representation is achieveb by applying an Extensible Stylesheet Language Transformation (XSLT) to the XML results. Furthermore, it is possible to directly output the results to the console by using the console formatter.

5.7 Training

The training framework included in Semdroid aids in creating new analysis plugins that require a training process. The basic functionality of this training process can be found in Section 4.7. Figure 5.6 shows the training architecture. As usual, the *AppParser* is used to parse the training applications. The resulting training App objects are then analyzed by a training analysis that directly attaches reference labels to the components of the App objects. These labels represent the desired results the analysis to be trained should ideally produce.

Then, an analysis-specific trainer can be deployed to create the models required for the new analysis. The input for this trainer are the pre-labeled App objects. For example, a trainer for analysis approaches based on machine learning would create machine learning models by using the data and reference labels found in the App objects. Once this analysis-specific training process has been completed, the resulting model files are stored and the training framework generates additional files required for the new analysis to work. Then, the new analysis can be used to analyze new applications.

5.8 On-Device Analysis

The Semdroid Android application consists of two activities: one for listing all installed applications, and one for displaying the analysis results. Once an application as well as an analysis to be

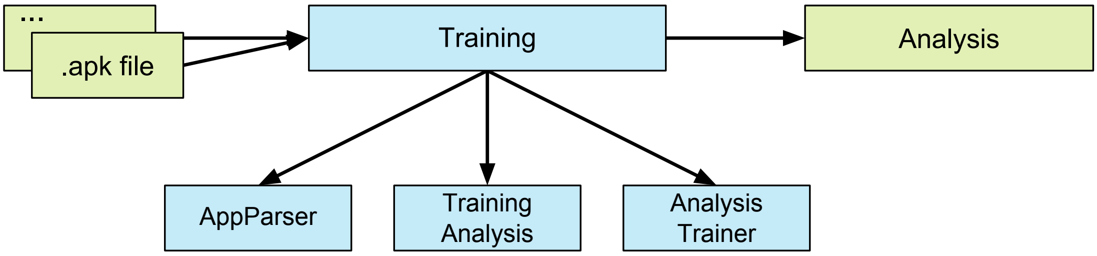


Figure 5.6: Training architecture. Given training applications are parsed and pre-classified. An analysis-specific trainer is then used to create models required by the new analysis.

performed has been selected by the user, the application will be analyzed by using the framework according to Section 5.3. The resulting *TestSuiteReport* is then parsed and displayed by the second activity. General information on the Semdroid Android application can be found in Section 4.8.2.

In addition, an interface is available that allows third party applications to analyze applications and to retrieve the analysis results. An Intent service is available, which can be started by any application that has the corresponding custom Semdroid permission. The starting intent contains information about the application under analysis (i.e., the package name or the path to the Android application package) and the analyses to be performed. The given application will then be analyzed by the Semdroid framework according to these parameters. Once the analysis is finished, a system-wide broadcast is sent containing the analysis results. Any application with the corresponding permissions can implement a broadcast receiver to listen to these analysis results.

One use case for this interface would be a policy manager that analyzes all applications before installing it on the device. According to the analysis results, the policy manager can then prevent or allow the installation of the application.

Chapter 6

The Semantic Pattern Analysis

With Semdroid, a powerful static Android application analysis framework is available. Now we present a new static analysis technique that can be employed within this framework, the *Semantic Pattern Analysis*. There are many different approaches for static application analysis. Since the goal of this thesis is to detect rather general functionality, such as cryptographic code or SMS-handling capabilities, we decided to base this analysis process on machine learning. These rather abstract application capabilities can have multiple different implementations and thus vary from app to app. For example, there are many different encryption algorithms available, which are all based on different concepts. Even for the same encryption algorithm, countless different implementations are possible. Now, if we want to correctly detect all those algorithms and label them, for example, as "cryptography", sophisticated detection algorithms are required that are able to recognize this variety of different, but in a way also similar, implementations. Machine learning algorithms are very well suited for this task and thus have been used as a basis for the *Semantic Pattern Analysis*.

But why *"Semantic Patterns"*? Since we want to combine both numerical and symbolic features, we need to transform the resulting feature set to a simple vector that can be processed by machine learning algorithms. As explained in Section 2.10, the *Semantic Pattern Transformation* can be used to do exactly that – transform arbitrary feature sets to Semantic Patterns, simple vectors suitable as input for machine learning algorithms.

This chapter describes the proposed new *Semantic Pattern Analysis*. Section 6.1 outlines the workflow for application analysis: First, features are extracted from the Android application that characterize the component's functionality. Section 6.5 delves into the details of this feature selection process as different feature types and values as well as filtering methods and feature representations are presented. Then, the features are transformed to Semantic Patterns, which will be explained in Section 6.6. Finally, these patterns are used as input for various machine learning algorithms that perform the final classification step. Section 6.7 gives an overview of this final step.

In order to create a new Semantic Pattern Analysis, a training process has to be performed that creates the models required for the analysis. Section 6.8 will elaborate this process in more detail.

6.1 Analysis Workflow

Figure 6.1 shows the basic analysis workflow. As described in Section 4.3, the Semdroid framework provides the input for the analysis, the App object. Using this object, the following three

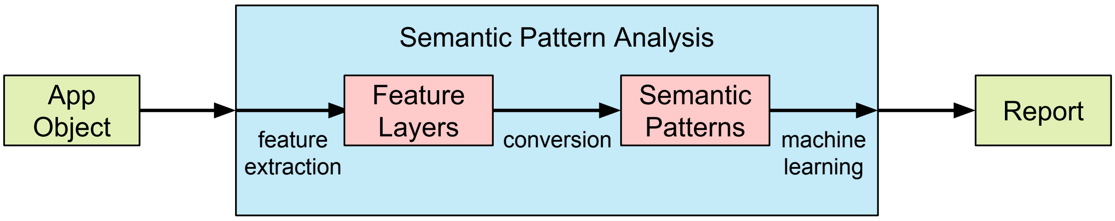


Figure 6.1: Semantic Pattern Analysis. First, feature layers are extracted, which are then converted to Semantic Patterns and classified using machine learning algorithms.

steps are performed: *feature extraction*, *Semantic Pattern Transformation*, and final component *analysis* by applying *machine learning*.

The first step is to create one or multiple *feature layers* containing various characteristics found in the App object. The simplest variant consists of a single feature layer, but advanced analysis processes may also create multiple layers. Each of these layers contains a list of so-called *instances*. An instance represents a single component of the application under analysis and is classified and labeled separately (at least for the single layer case). The three component types currently used are: methods, classes, and the whole app object itself. Section 6.2 gives detailed information on these feature layers. Furthermore, Section 6.3 gives more information on instances and Section 6.4 describes the component filtering process utilized by the *Semantic Pattern Analysis*.

Each of these *instances* contains a list of features that represent the capabilities of the current component. The App object and its components hold many different feature types that can be used, including Dalvik opcodes, method calls, or local variables. Furthermore, it is possible to use different representations for a given feature, and to apply feature filtering mechanisms to retrieve relevant data only. According to the targeted functionality, a suitable feature set has to be chosen. The whole feature selection process with all its facets will be explained in detail in Section 6.5.

Once the feature layers have been created, the *Semantic Pattern Transformation* is used to convert these feature layers to Semantic Patterns. A previously trained Semantic Network is required for this process. This network has to be created by performing a training process described in Section 6.8. For each instance included in the feature layers, one Semantic Pattern is created according to Section 6.6. This resulting Semantic Pattern is a simple vector that is used for further analysis.

The resulting Semantic Patterns are then classified using various *machine learning algorithms*. The machine learning models have to be created in advance by applying a training process on pre-classified data. The training framework included in Semdroid (Section 4.7) can be used to perform this process. For each Semantic Pattern, and thus for each application component under analysis, a label is returned by the machine learning algorithm that describes the capabilities of the current component. Since the component used for creating the Semantic Pattern is directly linked to the pattern, the resulting label can be easily added to this component of the App object (or to multiple components if the source for the Semantic Pattern has been more than one component).

Finally, the application analysis report is created according to the Semdroid labeling concept presented in Section 4.5. The report contains all analyzed components and their labels and will be returned to the Semdroid framework, which performs several post-processing steps.

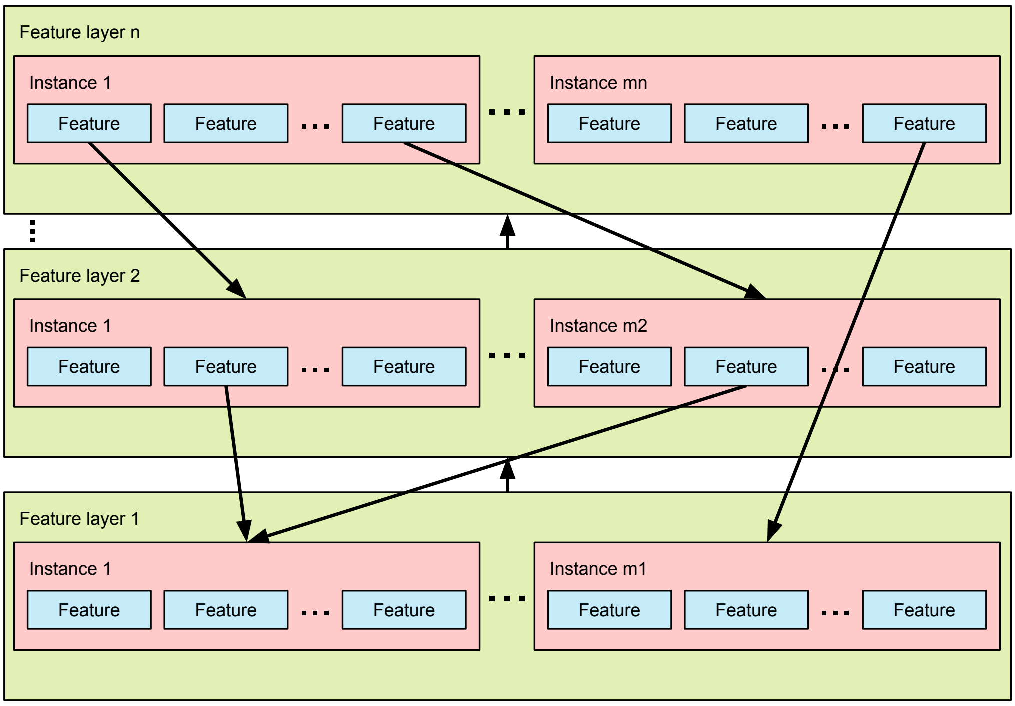


Figure 6.2: General feature layer structure. First, feature layer 1 is created, which holds an arbitrary number of instances. Then, layer 2 is added, which can already access the results (i.e. the instances) of the previous layer.

6.2 Feature Layers

The first step of the *Semantic Pattern Analysis* is to create *feature layers*. A feature layer contains a list of instances, where each instance represents a component of the application under analysis. Figure 6.2 shows the general feature layer structure. Multiple layers are created, each of which contains multiple instances. Instances contain a list of features. These features can be one of the following three types: symbolic, distance-based (numeric), or instance links. The last type can be used to add a whole instance as a feature to another instance, if multiple feature layer are used. This could be used, for example, by advanced machine learning algorithms like deep learning strategies. Multiple layers have a hierarchical structure, where the second layer has access to the elements of the first layer, the third has access to both the second and the first layer and so on. The same holds for the processing order: First, feature layer 1 will be processed, then layer 2 is used, which can already access the results of feature layer 1 (e.g., by using instance-link features).

For now, all implemented *Semantic Pattern Analyses* utilize a single feature layer. For Figure 6.2 this means that only feature layer 1 is used, and all other layers are omitted. Multilayer- and deep learning strategies could be addressed in future work – generating multiple layers as well as establishing instance links is already possible with the current version of Semdroid.

6.3 Instances

Instances represent application components. As already described, each instance contains a list of features. Depending on the intentions of the analysis, different instances and instance combinations are used. The three instance types currently utilized by the Semantic Pattern Analysis correspond to the components used for the labeling process discussed in Section 4.5. They are: method-, class-, and app-instances.

- **Method instances**
  Method instances represent a method of a given DEX class. This instance type is the most fine-grained component currently used for labeling. Thus, it is used to pinpoint certain functionality, like a method that processes SMS messages or implements cryptographic functionality.

- **Class instances**
  This instance type represents a whole DEX class and its contents. Typically, features found in the methods of the class are used in conjunction with additional class-features like its superclass or defined fields. Class instances are more coarse-grained than method instances. One use case is, again, cryptographic code detection. Typically, cryptographic implementations use one distinct class per encryption algorithm or hash function. Thus, class instances offer a suitable representation for these cryptographic implementations. Compared to method instances, which could be used as well, all methods are analyzed together and will be seen as a group. By using method instances, it could be possible that the cryptographic code is not recognized. For example, if the implementation utilizes many nested method calls, the analysis might not detect the correct functionality if the code of these nested method calls is not considered. If both analyses would detect the functionality correctly, method instances would label each method separately, whereas class instances would return one hit per cryptographic class.

- **App instances**
  App instances represent the whole Android application package. This means that a single app instance is used for the current application under analysis, and thus, that the application is labeled as a whole. A use case for app instances would be an analysis that determines the application category (e.g., game, entertainment, or tool).

It is also possible to mix components: One could add an arbitrary combination of method-, class- and app-instances to the same feature layer. Furthermore, different components, like for example whole packages, could be used if the feature generation process creates suitable instances.

For method calls, it is possible to include the features of nested methods as well. As explained in Section 4.3, this can be achieved by specifying a method call inclusion depth. All features of nested methods will be included until the defined inclusion depth is reached. Naturally this only applies to methods implemented by the application directly. Android or Java API calls will be ignored since the call itself can be used as a feature. This inclusion depth mitigates the problem of method instances where outsourcing code to separate methods could have a negative impact on the analysis results. By including the features of nested method calls, certain functionality can still be detected even if the code is split among several methods, or swapped out to static helper methods, which are all called by the "main" method that initiates the computation.
Consider the SMS broadcast receiver given in Listing 2.5 of Chapter 2. An analysis that detects SMS command handling capabilities of an application could check all `onReceive`-methods of broadcast receivers. If only the code of these methods is checked, the SMS handling capabilities of Listing 2.5 would not be detected, since they have been outsourced in the `doSomething` method, which is called from within `onReceive`. If solely the second method, `doSomething`, is analyzed, the analysis might not be able to find suspicious behavior either – because the context is missing.

The method could be required for different tasks not related to SMS handling. Hence, by including the features of nested method calls, the analysis can be able to detect the SMS command handling capabilities since all required features are combined. For the SMS receiver given in Listing 2.5, the features of `doSomething` would be included in the analysis of `onReceive`.
This functionality is not limited to method instances. Any instance type can benefit from this inclusion depth; class instances, for example, can directly include the features found in called methods (e.g., of static helper classes).

6.4 Component Selection

Not all components of the App object have to be used as instances for feature layers. They can be filtered according to multiple criteria. By examining the properties of the components, one can specify whether the current component should be used. The following sections explain the details behind specific component filters.

Depending on the implemented analysis, the filtering process can vary. A basic single feature layer analysis that adds method instances to the feature layer could use the following filter hierarchy: For each class included in the application, a class filter is used to determine whether the current class should be used. Then, if the class is indeed used, a method filter removes irrelevant methods. For each method that has not been filtered, a method instance is generated.

A basic single feature layer analysis that operates on class instances may only apply the class-filtering process and completely omit method filters.

App Filters

Since we only have one App object per application under analysis, app filters are not required. If the App object would be filtered, we would retrieve empty analysis results since no application (i.e. no App object) has been analyzed. Hence, we currently do not employ app filters.

Class Filters

Class filters decide whether a given class should be used. For example, common libraries can be excluded from the analysis by using white- and blacklists. Furthermore, more specific filters can be added as well: For instance, a broadcast receiver is required for SMS handling. Thus, some analysis approaches may filter for broadcast receivers only, and skip all other classes. Furthermore, the intent filter (Section 2.1.1 for the given receiver can be used to gather additional information. For the example of SMS broadcast receivers, it can be checked whether the given receiver listens to SMS actions by checking if the corresponding actions are defined in the intent filter. The analysis could then only analyze receivers that listen specifically to these SMS events. It has to be noted though that SMS receivers can also be registered dynamically. If this is the case, the intent filter defined in the Android manifest might not be complete. For more information on broadcast receiver registration, we refer to Section 2.1.2.

In general, such filtering operations include:

- **Full name**
  The full class name, including the package, is used for filtering.
  Example: `com.my.example.Clazz`

- **Package name**
  The package name or only a part of the package name is considered for filtering.
  Examples: `com.my.example`, `com.my`

- **Super class**
  The superclass of a given class can be used for filtering. For example, only classes that extend a certain class should be analyzed. It is also possible to traverse the superclass hierarchy until an external super class not implemented by the application itself is found. This external superclass is then used to determine whether the original class should be analyzed, for example by checking the full name or the package name of this super class. This can be useful if the analysis is only interested in classes that extend certain Android components.

- **Other information**
  Naturally, all other information found in the class object can be used for filtering as well. Any custom class filter can be implemented and used.

Method Filters

Method filters decide whether a given method should be used to generate features, depending on various parameters, like the method name, the method size (i.e. the number of Dalvik opcodes), or any other information found in the App object. Since most analyses utilize separate class- and method filters, the class-selection has already been performed by the class filter. Thus, method filters can focus on the information found in the actual method and do not necessarily have to check the used class. This also makes filters more flexible, since the same method filter can be used in conjunction with different class filters. For broadcast receivers, one might be interested in analyzing the entry point for a received intent – the `onReceive`-method. In order to allow only such methods, a whitelist can be used. For a real-world scenario, a class filter that only allows classes that extend broadcast receivers and a method filter that only allows methods named `onReceive` could be employed. Depending on the analysis, the class filter of analysis 1 could filter for boot receivers, whereas the class filter utilized by analysis 2 could only allow SMS receivers. In both cases, the same method filter for `onReceive` methods can be used.

Common filtering strategies include:

- **Method name**
  It can be checked whether the method name exactly matches a given string, or if the name contains certain strings. For instance, this is useful if a given class overrides a certain method provided by the Android system. By checking the method name, it is possible to target such specific methods.

- **Method length**
  The number of operations performed by the method can be used as an indicator whether the given method should be used.

- **Other method information**
  Additional information, like the method parameters or the return value, can be considered for the filtering process as well.

6.5 Feature Selection

The feature selection process is an integral part of the Semantic Pattern Analysis that has a huge impact on the accuracy and performance of the analysis process. Previous sections discussed feature layers and their contents, the instances. Now, the actual features that characterize these instances have to be gathered. In this section, we dissect this process and discuss different feature types, values, and representations. Furthermore, filtering and grouping strategies are presented, which help to retrieve relevant data only.

Each analysis utilizes a different feature composition. Finding a proper feature set that yields good analysis results is not an easy task. One has to be familiar with the targeted functionality. By looking at the characteristics of this functionality, suitable features have to be selected. Next, an analysis can be created that uses the desired feature composition. This analysis can then be evaluated and adjustments can be made depending on the evaluation results. This process may have to be repeated several times until the analysis performance is satisfactory.

It has to be noted that the feature order is irrelevant for the analysis process due to the structure of the Semantic Pattern Transformation (see Section 6.6). Looking at opcodes, this means that the order of their execution is irrelevant. This fact has both positive and negative effects. The negative side is that the order of, for example, executed instructions holds valuable information that could lead to more accurate analysis results in some cases. In contrary, one advantage of this order independence is, that code mutations can be easily detected. For the Semantic Pattern Analysis, this is a very welcome behavior, since we also want to detect variations of a given targeted functionality. If some operations are interchanged while the core functionality remains the same, the analysis results could differ if the feature order is considered. By not considering this feature order, the results remain the same and both variant will be detected. This is the desired behavior since the core functionality of the component has not changed.

The basic feature notation used in this thesis is as follows:

$$feature\,type = feature\,value$$

The feature type helps identifying the feature source and is a general descriptor for a feature group. Examples for feature types are: `opcode`, `local variable`, or `method call`. The same feature type can be used multiple times. For instance, if multiple opcodes should be added as separate features, the `opcode` type identifier can be used for all opcodes. Feature values represent the specific feature value of the current feature group: `OP_IF`, `int`, or `com.my.Class: doSomething()` are some examples. Combined, this leads to the following examples:

$$\begin{aligned} opcode &= OP\_IF \\ local\,variable &= int \\ method\,call &= com.my.Class : doSomething() \end{aligned}$$

Note that this is only a human-readable notation introduced for a better understanding. The actual implementation utilizes feature objects that hold this information.
All of the examples above use symbolic feature values (i.e. strings). It is also possible to use distance-based (numeric) feature values, including arrays. Two examples for distance-based feature values are:

$$\begin{aligned} opcode\,count &= 32 \\ opcode\,histogram &= [2, 3, 1, \ldots, 4] \end{aligned}$$

This allows to use different feature representations that have different properties and effects on the analysis performance. Section 6.5.3 gives an overview of these feature representations. Prior to that, Sections 6.5.1 and 6.5.2 present feature types and feature values currently used by analyses. Since it is practical to just use a subset of all features, Section 6.5.4 covers different filtering and grouping strategies.

The final feature selection process for a given instance looks as follows: The developer has to decide, which feature types should be used. Then, each feature of a given feature type to be used, is examined. A feature filter then decides whether it should be added. If it should be added, a corresponding feature identifier and feature value are chosen. Finally, the feature is added to the instance according to the selected feature representation for the given type.

6.5.1 Feature Types

This section describes the different feature types that are currently available for the feature selection process. Depending on the intents of the analysis, the developer has to choose the feature types to be used. Typically, a subset of the types listed below is used. In general, all data found in the App object (see Section 4.3) can be used for feature generation. Commonly used feature types include:

- **Dalvik opcodes**
  The Dalvik opcodes are utilized by almost all analyses. These opcodes have already been described in Section 2.2. They are either used directly, or can be grouped according to pre-defined rules.

- **Method calls**
  Method calls invoked by the component under analysis are also commonly used as features. A distinction between API calls, which are calls to external methods not implemented by the application, and internal method calls implemented by the application has to be made. In general, API calls are more interesting than internal method calls. In many cases, it makes sense to filter for a specific subset of API calls, which are relevant for the targeted functionality. Features of internal method calls can be directly included by inlining these features directly in the calling component itself, as explained in Section 6.3.

- **Local variables**
  Similar to method calls, local variables proved to be good features as well. To be more specific, the type of the local variable holds valuable information. Basically, there are two groups of local variables: First, there are primitive or basic data types (integer, float, double, byte, etc.). Second, there are composite data types, Java class objects. These java classes can again be separated into two groups: internal objects implemented by the application itself and external objects provided by the Android platform (e.g., `java.math.BigInteger` or `android.telephony.SmsMessage`)

- **Fields**
  The defined fields of classes can be used to extract features. Since fields basically are the same as local variables, except their scope, the same rules as for local variables apply.

- **Superclasses**
  The superclass of a given class under analysis can be used as a feature. This superclass can, again, either be provided by the Android framework or implemented by the application.

Since analysis processes are generally more interested in superclasses provided by the Android system, the framework offers the possibility to retrieve this first external superclass. In order to get this class, the superclass hierarchy is traversed until a class not implemented by the application is reached. Some examples for potentially interesting Android classes include: `BroadcastReceiver`, `Activity`, or `Fragment`.

- **Required permissions**
  Thanks to the Android permission map [19], it is possible to determine the required permissions for each component. Basically, a mapping between Android API calls and required permissions is used to generate a permission list for the component.

- **Intent filter parameters**
  For broadcast receivers, activities, or services, the Intent filter parameters (Section 2.1.1) defined in the Android manifest can be used to extract suitable features. Amongst others, these parameters are: the defined actions, categories, data types, or priorities.

- **Other information**
  It is possible to add any information found in the App object. For example, the Android manifest holds valuable information like defined activities, services, or content providers.

Currently, the App object created by the Semdroid framework only parses the Dalvik bytecode and the Android manifest of the Android application package. All other files, like native libraries, included resources, and assets, are not considered yet. In order to be able to add these features, Semdroid would have to parse these components and add them to the App object. Then, they can be accessed like any other application data and thus be used by the feature generators.

6.5.2 Feature Values

Once the features have been selected and filtered, a suitable feature value has to be chosen. The feature value can either be symbolic (i.e. a string) or distance-based (i.e. a double value or array). For each feature type, it is possible to use several different feature values.

By choosing appropriate feature values, an implicit grouping operation can be performed: Suppose we have two class names `com.my.FirstClass` and `com.my.SecondClass` as symbolic feature values. If the full class name is used as feature value, the two values are different and thus treated separately. However, if we just consider the package name of these classes, both feature values would be mapped to `com.my` and thus both classes would be grouped together.

Using feature values containing arbitrary data defined by the developer, such as local variable-, field-, class-, or method names, is not a good idea in most cases since the developer could simply change these names at any time. Since many Android applications use obfuscation techniques (see Section 2.7), these names are often replaced by meaningless strings, which makes them useless for the analysis process. Hence such feature values are usually not used. Class- or method names provided by the Android system (i.e. names of Android classes and API calls), however, are excellent feature values utilized by many analysis processes.
Besides the feature values presented below, it is also possible to use more complex feature values that depend on various criteria and that perform sophisticated grouping operations. In order to use such complex feature values, a custom feature value generator can be implemented.

Opcodes

For opcodes, only one feature value is used, namely the opcode group name. Different opcodes can be grouped together, where each group has a unique name. For example, it would be possible to group all mathematical opcodes to a group called `MATH`. Furthermore, it is possible to ignore certain opcodes. The used grouping mechanism can be defined by the developer of the analysis. Example: `OP_IF`

Method Calls

For method calls a myriad of different representations are possible. For upcoming examples, the following method call is considered:

```
com.my.example.Clazz: public String doSomething(int parameter1)
```

Examples for possible method call feature values are:

- **Full name**
  This feature value uniquely identifies the given method.
  Example: `com.my.example.Clazz: public String doSomething(int parameter1)`
- **Method name**
  Only the method name is used instead of the full name. This means that all methods with the same name, but of different classes will be grouped together.
  Example: `doSomething`
- **Class name**
  By using the class name, all methods from the same class will be grouped together since they share the same feature value. Instead of the class name, other class representations, like for example the superclass, can be used as well.
  Example: `com.my.example.Clazz`
- **Package name**
  Here, the package name or a part of the package name is used as feature value.
  Examples: `com.my.example`, `com.my`
- **Other method information**
  Other information of a given method like the return value, method parameters or method length can be used as well.
  Examples: `String`, `public`, `int`, `123`

Local Variables and Fields

Similar to method calls, different information of local variables can be used for feature values. Since fields and local variables are basically the same except the scope they can be accessed from, they share the same possible feature as well. One additional feature value that can be used For the following examples, “variable” refers to both local variables and fields.

There are two basic groups of variables: primitive/basic data types and composite data types. If the variable is a primitive data type (e.g., `int`, `long`, or `byte`), the name of the type itself is the only used feature representation. For composite data types, there are more options that can be employed. Suppose, we have the following variable:

```
com.my.example.Clazz myVariable
```

Examples for possible feature values are:

- **Class name**
  Example: `com.my.example.Clazz`
- **Package name**
  Again, the package name or only a part of the package name can be used as feature value. All variables of the same package will be grouped together.
  Examples: `com.my.example`, `com.my`
- **Variable name**
  Since the actual name of the variable does not hold valuable information due to obfuscation, using the variable name is only reasonable in special cases.
  Example: `myVariable`
- **Full name**
  Similar to the variable name itself, this feature value should only be used in special cases.
  Example: `com.my.example.Clazz myVariable`
- **Other information**
  It is also possible to use any other information as feature value, like the number of methods included in the class, or the number of opcodes of the class, or the superclass name.
  Example: `123`

Superclasses

The class type of a given superclass can be used as feature value. In general, analyses are more interested in superclasses defined by the Android system rather than superclasses implemented by the application itself. Thus, Semdroid provides the functionality to retrieve this first external superclass. These superclass feature values can also be used for method calls and variables. They are:

- **Superclass name**
  All classes that extend the same superclass will be grouped together.
  Example: `android.app.Activity`
- **Package name**
  By just using the package name, it is possible to group several superclasses together. An example would be to group all Android components or UI elements.
  Examples: `android.app`, `android`

Permissions

For permissions, the full permission name is used by most analysis approaches as feature value. An example for such a permission feature value is `android.permission.SEND_SMS`.

It would also be possible to group permissions together. For example, there are several permissions for SMS handling. One could group all these SMS permissions together to a virtual permission name like for example `SMS_HANDLING`.

Intent Filters

Certain Android components, like broadcast receivers or activities, use intent filters to specify which actions they would like to react to (see Section 2.1.1). The values found in these intent filters can be used as features, like the defined actions, category, mime type, and so on. For example, the main activity of an application must define the action `android.intent.action.MAIN` and the category `android.intent.category.LAUNCHER` in the intent filter.

6.5.3 Feature Representation

For each feature or group of features, a suitable feature representation has to be chosen. The basic feature notation has already been discussed in the introduction of this section. A type identifier as well as a feature value must be specified for each feature to be used. The feature value can also be used for grouping operations as explained in Sections 6.5.2 and 6.5.4. Now, the final feature representation has to be chosen. Different feature types can use different feature representations. The following feature representations are currently used:

- **Basic feature value**
  This representation directly uses a feature type/value pair. The feature value represents the current feature and can be used to perform grouping operations. Multiple features of the same feature type typically share the same type identifier.

  General format:
  $$featureType = featureValue$$
  Example:
  $$opcode = OP\_IF$$

- **Feature type count**
  Instead of directly using the feature value, the number of occurrences of the given feature value is used. Two cases have to be distinguished: it is possible to consider a whole feature category (e.g., all opcodes or method calls), or just a subset. For instance, one could be interested in the opcode count of the current component or just in the number of occurrences of a specific opcode or opcode group. It has to be noted, that the count of occurrences does not necessarily correspond to the number of actual executed operations on the CPU. If loops are used that execute the same code several times, these operations are only counted once since these loops are not resolved in the current version of Semdroid. The same holds for method calls invoked from within a loop. If the features of nested method calls are directly inlined as explained in Section 6.3, these features are also only counted once.

  General format:
  $$featureType = featureValueCount$$

Examples:

$$\begin{aligned} opcodeCount &= 42 \\ OP\_IF &= 11 \\ com.my.Object &= 8 \end{aligned}$$

- **Feature usage**
  This representation states whether a given feature is used (is available). For example, it could be of interest whether a given application requires the SMS permission. This can be realized by defining a feature identifier called `requiresSMSPermission` and by setting the corresponding feature value to `true` or `false`. If a distance-based feature value should be used instead of this symbolic value, it is also possible to replace `true` with `1` and `false` with `0`.
  Another possibility to realize this kind of functionality is to simply use the "basic feature value" representation. For the example of the SMS permission, one could also add the feature $permission = android.provider.Telephony.SMS\_RECEIVED$, if the permission is indeed requested by the application. One difference between these two representations is that if the SMS permission is *not* required by the application, the resulting features differ: If the "basic feature value" representation is used, the SMS permission feature is missing. For the "feature usage" representation, a feature will always be added that explicitly states whether the permission is required.

  General format:

  $$featureType = true|false$$

  Examples:

  $$\begin{aligned} requiresSMSPermission &= true \\ callsAbortBroadcast &= false \\ usesBigInteger &= 1 \end{aligned}$$

- **Histogram**
  This representation counts the number of occurrences of each feature value contained in a pre-defined feature set. The resulting histogram is then used as a distance-based feature value. Again, by choosing appropriate feature values, grouping operations can be performed. In addition it is recommended to normalize the histogram. Especially opcode- or opcode group histograms yield very good results and are utilized by many analyses. It has to be noted, that the feature set used for the histogram as well as the order of this set always has to remain the same. Even if a given feature is not present, the value must be set to 0 and included in the histogram in order to be able to compare two histograms. Similar to the "feature type count" representation, loops are currently not handled; looped features are only counted once.

  General format:

  $$featureType = [fCount1, fCount2, \ldots, fCountN]$$

  Example:

  $$opcodeHistogram = [1, 3, 2, 5, \ldots, 5]$$

It is also possible to combine different feature representations for the same feature type. For example, one could use a histogram representation for basic local variable types, and additionally add selected local variables in a different format, like as "basic feature values".

6.5.4 Feature Filtering and Grouping

Since it is not practical to use all features of a given feature type of an application, filtering strategies have to be employed to constrain these features. Our test showed that choosing appropriate filters and groups improves both the performance and the accuracy of the analysis. If all features are used, a lot of "noise" and unnecessary information is introduced. Many features are not relevant for the current analysis and thus it does not make sense to include them in the analysis process. Using all features also decreases the analysis performance, since all of these features have to be evaluated. Thus, it is wise to just pick certain features or feature types and to ignore the rest.

First, one has to decide which general feature types should be used for the current analysis. This has to be assessed manually by the developer of the analysis and depends vastly on the targeted functionality. In order to select a suitable feature set that yields good analysis results, in-depth knowledge about the characteristics of the functionality to be detected is required.
Suppose an analysis that detects symmetric-key cryptography has to be developed. First, one has to look at the main characteristics of symmetric cryptography. Based on these characteristics, the developer then has to pick suitable features. For this example, the developer of the analysis could decide to use the opcodes and local variables. Furthermore we are only interested in mathematical operations as well as primitive local variable types, like int, double, boolean, etc. Then, the opcodes can be grouped: all non-mathematical opcodes can for example be represented by a group `OTHER`, while mathematical opcodes are used directly. Furthermore, a filter for local variables is applied that only considers primitive local variables and discards all composite data types.

In this example, the two different approaches for eliminating irrelevant information have been combined: feature *filtering* and feature *grouping*. The feature filtering process is used to completely remove irrelevant features. The grouping process is then used for the remaining features to remove irrelevant or too detailed information, by reducing the number of possible feature values and by grouping them together.

These two approaches are completely independent from each other. For example, a filter for method calls can be applied that only adds a method call, if the method is included in a certain set of packages. Independent of this filter, feature value grouping can be performed to group the remaining method calls. The resulting grouped feature values are then used according to the feature representation selected by the developer.

Section 6.5.2 presented an overview of used feature values and explained how they can be used for grouping. But it is also possible to use a more general grouping approach. For example, one could group all basic local variables to a virtual new feature value called `BASIC` and all other types to a feature value called `OTHER`. Furthermore, it is possible to use any arbitrary grouping: For instance, one could group two local variables `com.first.Class1` and `com.second.Class2` to a feature value `firstOrSecondClass`.

The different feature values discussed in Section 6.5.2, which are used to perform grouping operations, can be used for filtering as well: A temporary feature value, which is independent from the actual feature value used for the current feature, can be chosen. This temporary feature value can then be used by the given filter to decide whether the feature should be used. An example for this would be a filter that solely adds local variables of a certain package: The temporary feature value is the package name. If this temporary feature value is then included in a whitelist, the local variable is considered as a feature. The actual feature value can be different from the temporary feature value, like the full class name of the local variable.

Furthermore, the filtering strategies employed for class- and method filtering process (Section 6.4) can also be used in an extended form for feature filtering: Basically, for feature types that have a reference to a class (e.g., the superclass, method calls, local variables, or fields), the same class component filtering methods can be applied. Moreover, since class references could point to external classes not implemented by the application (like an external superclass provided by the Android platform), filters can also consider this information for the selection process. For method calls, the component filtering strategies for methods can be applied. In addition to these component filtering strategies for methods, method calls can also be external API calls provided by the Android framework. Hence, filters can also employ this information for their decision making.

A final example: For each method call, one of the class-filtering strategies mentioned in Section 6.4 can be employed to pre-filter for method calls of certain relevant classes. Then, a second filtering step can be performed that checks if the method name itself contains a certain text.

6.6 Semantic Patterns

Once the feature layers have been generated, the Semantic Pattern transformation is used to convert these layers to Semantic Patterns, value-based vectors, which are used for the final analysis process.

For current analysis plugins, a single feature layer is created. This layer contains a list of instances. A previously created Semantic Pattern network is used to derive Semantic Patterns from these instances. Since instances can contain both symbolic and distance-based values, the Semantic Pattern Transformation is used to create corresponding Semantic Patterns, one for each instance. These Semantic Patterns are simple vectors that can be supplied to machine learning algorithms. For more information on this transformation process, we refer to Section 2.10.

6.7 Machine Learning

The last step of the *Semantic Pattern Analysis* is to analyze the Semantic Patterns. Machine learning algorithms are used to classify these patterns. Then, the component attached to the Semantic Pattern can be labeled according to the results of the machine learning process.

Basic machine learning principles have been presented in Section 6.7. Two different approaches can be used to analyze the Semantic Patterns: *classification* and *anomaly detection*. For classification, we have a pre-defined number of labels, also called classes. For each Semantic Pattern, the most probable class it belongs to is determined. Anomaly detection on the other hand decides whether the Semantic Pattern seems to be “normal”, meaning that the pattern is similar to previously seen “normal” patterns. If the pattern deviate from these “normal” patterns, it can be considered as an *anomaly*. For instance, one could use several benign applications to train *normal* application behavior. Malicious applications that deviate from this normal behavior can then be detected by the machine learning algorithm. The next two sections will give more information on these two approaches.

6.7.1 Classification

In this thesis, we use support vector machines (SVM, see Section 2.9) to classify the Semantic Patterns. Depending on the analysis, different kernels and parameters can be used that are tailored specifically for the given analysis. Furthermore the number of classes can vary from analysis to analysis as well: For some analysis approaches, a binary class, consisting of two possible classes, is required, while others may require multiple classes. If more than two classes are used, a multiclass SVM is required that divides the classification problem into several binary class problems. SVMs have been chosen because first tests showed great classification results compared to other machine learning algorithms. However, it is also possible to specify any other machine learning algorithm supported by the Weka framework.

6.7.2 Anomaly Detection

For anomaly detection, distance-based techniques can be used. These algorithms try to detect outliers. The Semantic Pattern framework can be used to cluster the training data. These clusters can then be used to decide whether a new Semantic Pattern shows normal behavior. If this pattern does not lie within any cluster, it can be considered as an anomaly.

Currently, we do not employ analysis approaches based on anomaly detection. All analysis plugins presented in Chapter 8 are based on classification algorithms. However, in future work, an analysis process based on anomaly detection could be added to the Semdroid framework.

6.8 Training

A training process has to be performed that generates a *Semantic Network* and a *machine learning model* used for the analysis process. The training framework provided by Semdroid (Section 6.8) is used for this task. This framework performs pre-labeling of the training applications by applying a training analysis. The components of supplied App objects then include the desired labels. Then, the feature layers are extracted for each of these objects in the same way as for a normal analysis process. The feature layers for all training applications are then used to create a Semantic Network, which is persisted to a file. The first part of the training process, the creation of the Semantic Pattern network, is now complete.

The second training step creates a machine learning model. Besides creating the Semantic Network, the Semantic Patterns are also created for all instances. For each of these instances, the desired labels are attached to the representing Semantic Pattern. This data is then used to create a machine learning model by performing a standard *supervised learning* process (Section 2.9). Once the training process is complete, the resulting machine learning model is also persisted to a file.

The training process is now complete. The Semdroid framework creates all additional configuration files required for the new analysis. In order to analyze new applications, the Semantic Network and the machine learning model are automatically loaded. Then, the analysis process can be performed according to Section 6.1.

The pre-analysis process is very important in order to create analysis plugins with a great performance. The training data has to be labeled accurately, in the best case without any false positives or false negatives. A training classifier is used to perform this labeling process. As described in Section 6.8, any analysis can be used for this process – even another Semantic Pattern Analysis. However, in order to get accurate training data, a very good but also time-consuming approach is

to manually dissect the training applications, to assess the functionality of the components, and finally to label these components accordingly. The results can be recorded in a text file and then loaded by a training analysis that adds labels to the components according to this text file.
Furthermore, we also implemented a training analysis that labels components based on the folder the application is placed in.

Chapter 7

Semantic Pattern Analysis – Architecture

This chapter presents architectural details of the *Semantic Pattern Analysis*. As explained in the previous chapter, this analysis process can be employed within the Semdroid framework, which provides pre-processed Android application packages and handles the post-processing of the analysis results. Semantic Pattern Analyses try to locate certain application functionality within these pre-processed Android application packages and return detailed analysis results containing all findings. Section 7.1 gives an overview of the main components involved in this process. After that, subsequent chapters delve into the details of these components. Since the Semantic Pattern Analysis relies on a training process, the architecture of the training framework is outlined in Section 7.5.

7.1 Component Overview

Section 5.3 already presented the requirements for any given Semdroid analysis. In short, the input for an analysis is a pre-processed App object, which represents the Android application package under analysis. Based on the information found in this App object, the analysis has to analyze the application components and to return analysis results containing all findings. The blackbox structure for Semdroid analysis plugins has been presented in Figure 5.3.

Now, with the *Semantic Pattern Analysis*, we can take a look inside of one of these black boxes; Figure 7.1 shows the architecture behind this new analysis process. The App object is handed to the feature extractor, the component responsible for assembling vectors suitable as input for machine learning algorithms. Each of these vectors represents a certain application component. In order to create these vectors, a *feature layer generator* first assembles feature layers according to Section 6.2. These layers are then processed by the *Semantic Pattern framework*, which generates the final vectors demanded by the machine learning framework. Then, the machine learning framework labels each of these vectors separately. Label objects that correspond to the labels assigned by the machine learning framework are created and all components are attached to the labels they belong to. The final *AppAnalysisReport* containing all of these labels is then returned to the Semdroid framework.

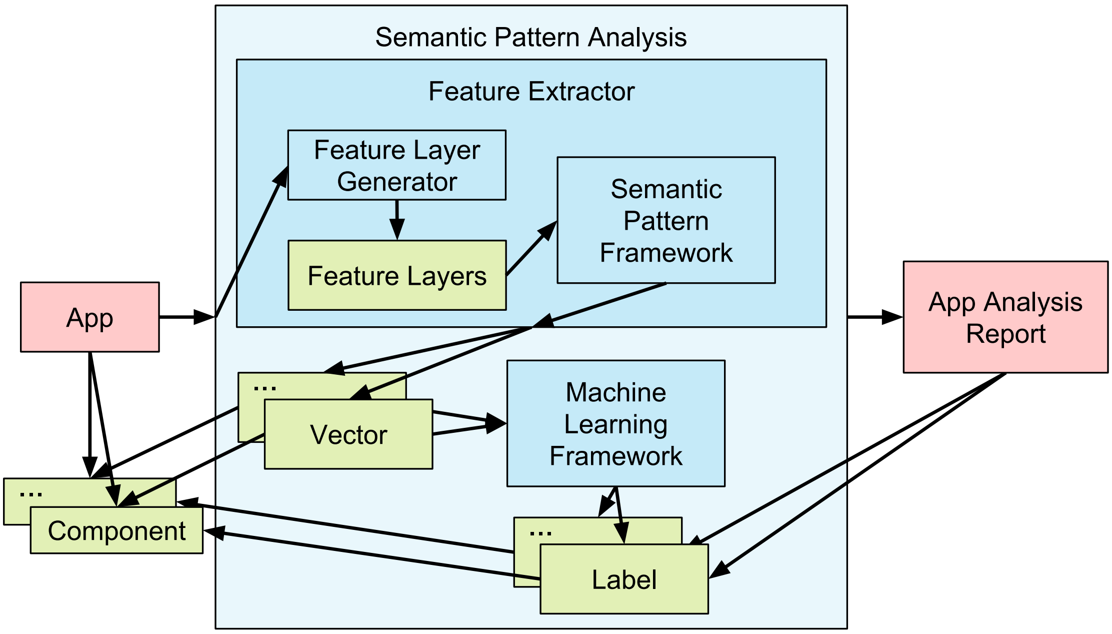


Figure 7.1: Semantic Pattern Analysis architectural overview. The App object (red) is used as input for the analysis. Vectors are extracted and analyzed by the machine learning framework. The resulting *AppAnalysisReport* (red) contains all labels and links to components belonging to the labels.

7.2 Feature Extractor

The feature extractor is responsible for creating vectors consisting of double values, one for each component under analysis. These components are also linked to the vectors in order to be able to distinguish them later on. In theory, any feature extractor that generates such vectors suitable for machine learning algorithms could be used for this process. For the *Semantic Pattern Analysis*, this feature extractor consists of two parts: The first step is to generate feature layers. These layers are then converted to the desired list of vectors (*Semantic Patterns*) by applying the *Semantic Pattern Transformation*. Upcoming sections present the architecture behind these two processes.

7.2.1 Feature Layer Generator

Input:	*App object*
Output:	*Feature layers*

The input for every analysis is the pre-processed App object. The feature layer generator is responsible for extracting features from the application, for grouping these features together in instances, and for further grouping the instances in feature layers. Filtering operations are performed to retrieve analysis relevant data only.

The structure of the *FeatureLayerGenerator* can be found in Figure 7.2. The general feature layer structure has been described in Section 6.2 and is depicted in Figure 6.2. Feature layers contain instances. Different instance generators, managed by the feature layer generator, are used to create these instances. According to certain criteria, the feature layer generator selects an appropriate instance generator for a given component. This instance generator then assembles the final

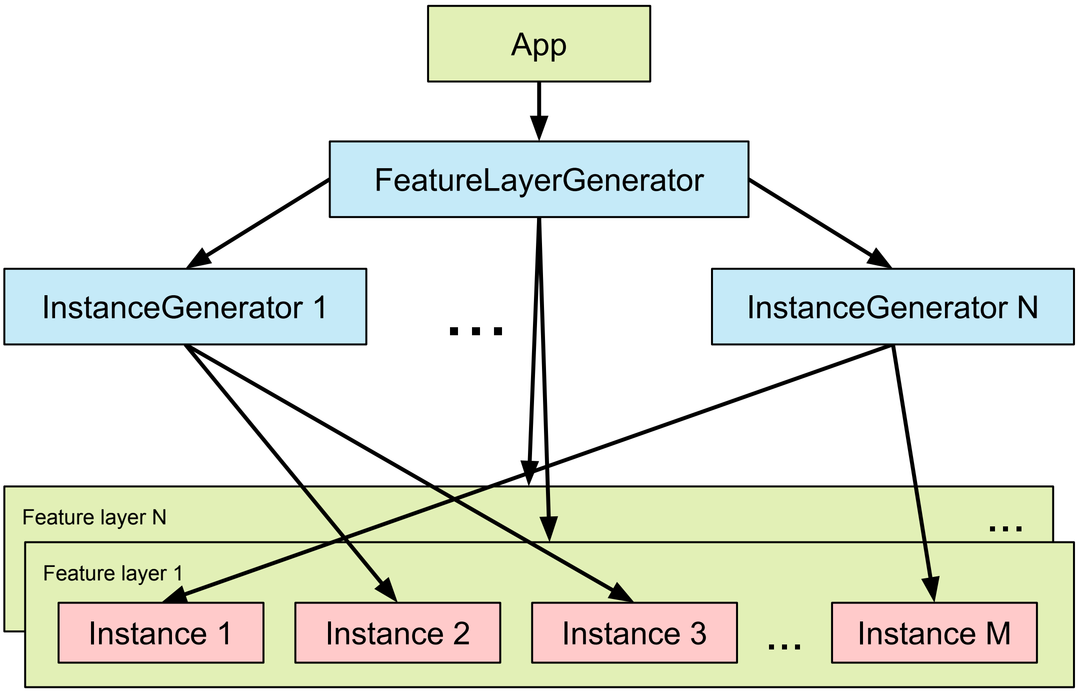


Figure 7.2: Feature layer creation and structure.

instance containing selected features representing the component. For example, the feature layer generator could select different instance generators for different class types. If a class extends an Android activity, a different generator can be used for methods of this class than for methods of a class that, for example, is an Android service. Simple single-layer generators, just use one instance generator for all components of a given type. Furthermore, some feature layer generators employ filtering strategies to filter for relevant components only.

Instance generators take a given component and assemble the instance representing this component. Specialized generators are available for the three main component types: the *AppInstanceGenerator*, the *ClassInstanceGenerator*, and the *MethodInstanceGenerator*. These instance generators are independent of the used feature layer generator, meaning that the same instance generator can be employed by several different feature layer generators (e.g., by a single-layer- as well as a multi-layer generator).

Now, three specific feature layer generators will be highlighted. All of them pursue a single feature layer strategy and are used for three different component types: apps, classes, and methods. Following after these three feature layer generators, more details on instance generators will be given.

App Single Feature Layer

The *AppSingleFeatureLayerGenerator* is the simplest feature layer generator currently employed. It creates a single instance for each application, as depicted in Figure 7.3. The structure is very simple: The *AppInstanceGenerator* takes the App object and creates a single app instance by using various data found in the application, like Dalvik opcodes, utilized local variables, or Android API calls.

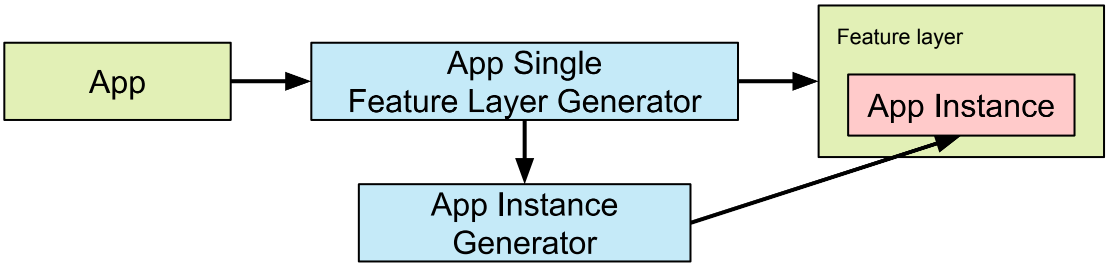


Figure 7.3: App single feature layer generator. A single app instance is generated for each application.

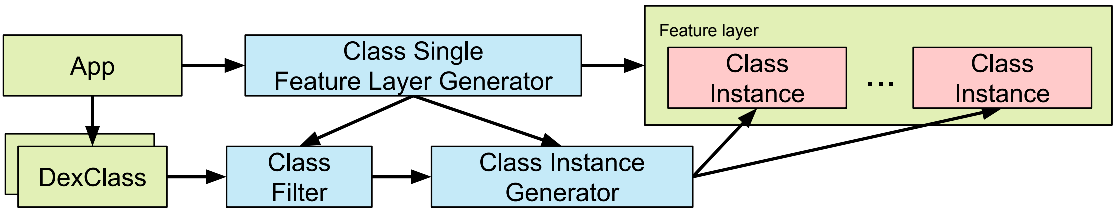


Figure 7.4: Class single feature layer generator. For each class, a class filter decides whether an instance should be added for the given class. The class instance generator is responsible for creating these class instances.

Class Single Feature Layer

Instead of creating a single instance for the whole App object, the *ClassSingleFeatureLayerGenerator* holds several class instances. For each class included in the Android application, a separate class instance is generated. Figure 7.4 depicts the structure of this generator. A *class filter* is used to decide whether a given class should be added. For example, a simple filtering strategy is to remove common libraries not relevant for the analysis. For each remaining class, the *ClassInstanceGenerator* creates a class instance, containing various features representative for the given class. More information on filtering strategies can be found in Section 6.4.

Method Single Feature Layer

Method single feature layer generators create, as their name already suggests, a single feature layer containing multiple method instances. Similar to the previously mentioned class feature layer generator, a class filter is used to remove unwanted classes. Then, all methods of remaining classes are traversed and a method filter performs an additional selection process. Since the filters are independent of the feature layer generator, the same implementation can be used for any feature layer generator. Surviving methods are then passed to the method instance generator, which generates the final instances.

Instance Generation

Instances contain a number of features. The feature selection process has already been elaborated in detail in Section 6.5. The architecture behind this selection process can be found in Figure 7.6. In short, the instance generator first performs a feature filtering process. Application components contain various data, like opcodes or local variables. Feature filters decide whether a given portion

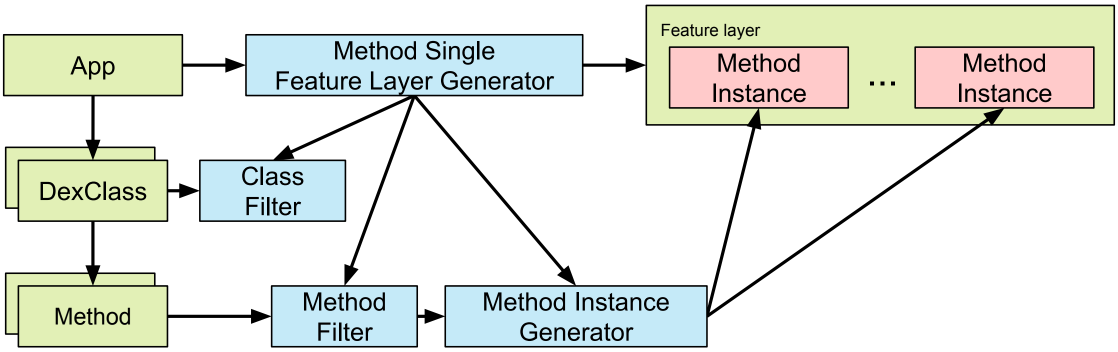


Figure 7.5: Method single feature layer generator. The class- and method filters remove unwanted data. The MethidInstanceGenerator creates instances for the remaining methods, which are added to the feature layer.

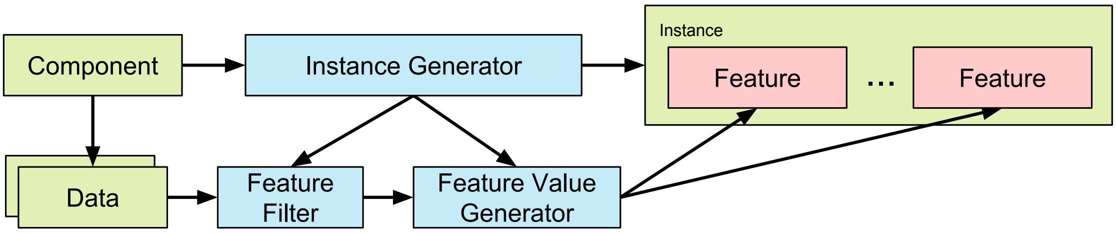


Figure 7.6: General instance generation architecture. A feature of a given feature type is filtered, a feature value is generated and this value is used to generate a feature according to the chosen feature representation.

of this data qualifies for a suitable feature. Data that passed this filtering process is handed to the feature value generator, which generates an appropriate feature value. It is possible to have multiple feature filters and feature value generators. For example, depending on the feature type (e.g., whether the feature is an opcode or a local variable), different filters and value generators can be employed. Then, according to the selected feature representation (see Section 6.5.3), the final data is added to the instance.

The analysis developer can fully configure all parts of this system: The feature filtering process can be set up according to the requirements of the analysis process. For different feature types, different filters, feature value generators, and feature representations can be specified. For example, the developer can decide to add grouped opcodes in the form of an opcode histogram, and to add filtered local variables directly as “basic feature values”.

In general, any instance generator can use any information found in the App object. In practice, the feature scope varies depending on the component type: For example, for a class instance generator, usually the features of the class and of all methods this class implements are used. Method instance generators normally only include features of the method the instance represents. Sometimes it can be useful to also include information of the parent class, like the name of the superclass (e.g., broadcast receiver, activity, or service). App instance generators can include any information of the whole application.

7.2.2 Semantic Pattern Framework

Input: *Feature layers*

Output: *Vectors (Semantic Patterns)*

The machine learning framework demands simple double vectors as input. Hence, we need to convert the feature layers to such vectors – one vector per instance. For the *Semantic Pattern Analysis*, the *Semantic Pattern Transformation* is applied on the feature layers, which does exactly what we need: convert the feature layers to a number of Semantic Patterns, simple double vectors. In order to achieve this task, we had to extend the Semantic Pattern framework developed by Peter Teufl to suit our needs. The architecture of the Semantic Pattern framework can be found in Teufl [52]. We added an interface to communicate with this framework. The extended framework allows us to retrieve the Semantic Patterns and other required data, like the Semantic Networks.

7.3 Machine Learning Framework

Input: *Vectors (Semantic Patterns)*

Output: *Labels*

Now, we need to label the vectors retrieved from the feature extractor. For this task, machine learning algorithms are used. In particular, we utilize the Weka framework (Section 2.9) for this purpose. First, the vectors are converted to Weka-specific instances, which are then separately analyzed by the machine learning model generated by the training process. For each of these Weka instances, and thus for each component under analysis, a label string is returned. These labels are then used by the Semantic Pattern framework to assemble the final analysis results.

7.4 Analysis Results

According to Section 4.5, an *AppAnalysisReport* has to be returned to Semdroid, containing all analysis results. For each unique label string generated by the machine learning framework, a *Label* object is created. The components linked to each vector (e.g., analyzed methods or classes) are then attached to the label they belong to, as depicted in Figure 7.1. Furthermore, the *Label* object is also added to the *component* itself, according to the Semdroid labeling specifications presented in Section 4.5. The resulting application analysis report is then returned to the Semdroid framework.

7.5 Training

As stated in Section 6.8, the Semantic Pattern Analysis requires a training process that generates the *Semantic Network* and the *machine learning model* required to be able to analyze new applications. The Semdroid training framework presented in Section 5.7 is used for this task. It performs the application pre-processing and supplies the pre-labeled App objects. Then, an analysis-specific trainer is called that generates all models required by the analysis, as can be seen in Figure 5.6.

For the Semantic Pattern Analysis, this analysis-specific training module is depicted in Figure 7.7. For all training applications, feature vectors are extracted using the feature extractor

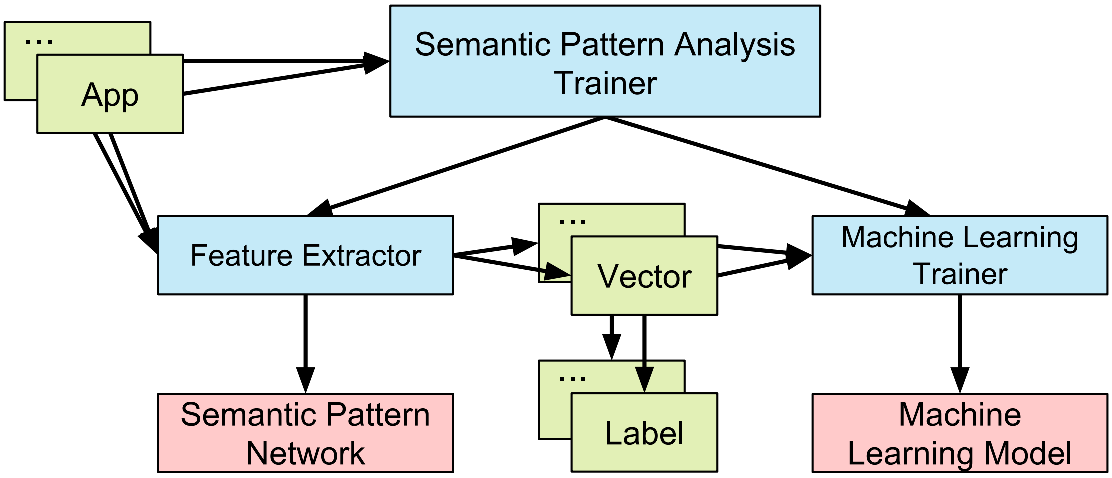


Figure 7.7: Semantic Pattern Analysis training architecture. Vectors are extracted from pre-labeled training App objects. The machine learning training framework creates a machine learning model using the vectors and their corresponding target labels as input. The resulting objects, the Semantic Network and the machine learning model, are highlighted in red.

architecture discussed in Section 7.2. Since all application components have already been pre-labeled by the Semdroid framework, these reference labels are attached to the vectors. These labeled vectors are then handed to the machine learning training framework, which generates a machine learning model using the supplied training data. Again, we use the Weka framework for this training process. The machine learning algorithm and the parameters to be used can be specified by the analysis developer. The resulting machine learning model is then saved together with the *Semantic Network* created by the Semantic Pattern framework.

Moreover, additional debug data can be saved, including an ARFF file that contains all training instances supplied to the machine learning framework, as well as the feature layers extracted from the training applications. The ARFF file can, for example, be used to manually train a machine learning model using the Weka Explorer (Section 2.9).

Chapter 8

Semantic Pattern Analysis – Applications

With the *Semantic Pattern Analysis*, we have a powerful analysis technique that is able to pinpoint specific application functionality.
This chapter gives an overview of specific analysis plugins we have implemented and evaluated. Each of these analysis plugins has been hand-tailored and adjusted in order to achieve good analysis results. First, the targeted scope has been defined, whether the analysis should operate on app-, class-, or method level. Then, we selected a suitable feature set that harmonizes with the targeted functionality. We also tested different feature sets and scopes for the same analysis and picked the best feature combination.

For the training process required by the Semantic Pattern Analysis, we need suitable training applications. These applications have been handpicked, manually analyzed, and the functionality of their components noted in a text file. The training analysis used to pre-label these applications just loads the component names and their desired labels from this text file. After the training process has generated all models required for the new analysis plugin, it is possible to analyze new, unknown applications.

The following analysis plugins have been implemented and evaluated:

- Symmetric cryptography
- Asymmetric cryptography
- SMS handling

The focus of this thesis lies in detecting cryptographic code. Since symmetric ciphers and hash functions have similar structures (e.g., some hash functions are based on symmetric block ciphers), they are grouped together.
Asymmetric cryptography on the other hand does not necessarily have strong similarities to the other two types and is thus targeted separately, by different analysis plugins. It would also be possible to combine all these types in a single analysis that detects all cryptographic structures and assigns different labels according to the type of cryptography. For instance, this analysis could have the following three labels: *SYMMETRIC*, *ASYMMETRIC*, and *NORMAL*. In this thesis, we decided to have separate analysis plugins for the symmetric- and asymmetric case.

Section 8.1 gives an overview of the analysis creation process. Following sections will then delve into the details of each of these analysis plugins, what feature compositions have been used,

what training data has been used, and how the models have been trained.
The evaluation results for these analysis plugins will be presented and discussed in the next chapter, Chapter 9.

8.1 Analysis Creation

This section explains the approach we took to create new analysis plugins.
First, a given target functionality has to be chosen. The developer must have detailed knowledge about this functionality and its characteristics.
Next, the developer has to decide which feature layers should be used. For most cases, a single feature layer should suffice, but a multi-layer approach could yield better result for some cases. For the analysis processes presented in this chapter, we always used a single-layer approach.

Then, it has to be decided which components should be targeted. The three main component types are: methods, classes, and apps. Depending on the targeted functionality, it has to be decided which of these approaches is appropriate. For example, for targeting the application category, it makes sense to work application-wide, whereas for detecting other functionality like cryptographic code, both the class- and method-based approaches could be suitable.
It is also possible to mix these instances. One could use class-instances for class X (e.g., if the class is an activity) and method-instances for class Y – for example, if the class is a broadcast receiver. Once the component type has been selected, appropriate class- and method filters can be set to retrieve relevant data only. For example, a class filter for broadcast receivers could be employed.

Then, the feature set has to be selected according to Section 6.5. These features have to be chosen depending on the characteristics of the targeted functionality. For example, in order to detect cryptographic code, mathematical opcodes could be used. It is possible to combine different feature types, like opcodes and method calls. For each of these feature types, filters can be set, feature values have to be chosen, and a feature representation has to be selected. Once all features have been selected, the machine learning algorithm and its parameters have to be defined.

Finally, we need training applications. Then, we manually examine their functionality and label the application components accordingly. In a text file, we note the components and the desired labels. Once this process has been completed, the new analysis can be trained. The training analysis takes the previously created text file and labels all application components accordingly. All files required for the new analysis will be automatically created and the new analysis can be used.

8.2 Symmetric Cryptography

The first analysis process can detect symmetric ciphers and hash functions. As discussed in Section 2.8, cryptographic operations rely heavily on mathematical functions. Thus, these mathematical operations are an essential part for feature vector generation. All operations are primarily performed on basic data types (byte, int, double, float) and arrays of these basic types.

Hash functions and symmetric block ciphers can have very similar structures, since many hash functions are based on such block ciphers. Even for hash functions that are not based on symmetric block ciphers, the performed operations can be very similar, they usually perform many mathematical operations, like shifts, AND, or XOR operations. This makes it very hard to distinguish

Targeted functionality:	Symmetric cryptography
Targeted components:	Methods
Feature Layer:	Single feature layer
Class filter:	–
Method filter:	More than 30 opcodes
Method call inclusion depth:	2
Features:	Opcodes
Opcode grouping:	Mathematical opcodes separately, others ignored
Opcode feature values:	Opcode group name
Opcode representation:	Histogram
Machine learning algorithm:	SVM

Table 8.1: Analysis configuration: symmetric cryptography detection

between hash functions and symmetric block ciphers. Thus, both of these types are considered to be *symmetric cryptography*.

For symmetric block cipher as well as hash function detection, the usage of an opcode histogram, containing mainly mathematical opcodes, yielded very good results. We also experimented with several other feature sets, which included basic local variables, or method calls, but by simply using the opcode histogram, we achieved the best results.

8.2.1 Analysis Configuration

The configuration for this analysis can be found in Table 8.1. This analysis targets methods implementing symmetric cryptography. A single feature layer is created, which contains method instances. We do not employ a class filter, methods of all classes are used. For methods, we only look at methods with more than 30 opcodes. We chose a value of 30 because the symmetric code we have manually analyzed performed at least 30 operations. The method call inclusion depth (see Section 6.3) is set to 2, meaning that all features of invoked methods, and of methods invoked by these methods, will be added as well.

As features, this analysis process simply utilized grouped opcodes, added as a normalized opcode histogram. All mathematical opcodes are considered as a separate group, like *ADD*, *OR*, *AND*, or *XOR*. All other opcodes will be ignored. A complete list of all opcodes can be found in Table A.1. A *Support Vector Machine* is used for machine learning tasks, since SVMs yielded great results.

8.2.2 Training Data

As training data, we manually labeled six AES methods found in the Bouncy Castle library as cryptographic code: three `encryptBlock` and three `decryptBlock` methods of the three classes *AESEngine*, *AESLiteEngine*, and *AESFastEngine*.

In addition, we picked 100 methods that do not contain cryptographic code and labeled them as "normal" code. These methods have been randomly selected from one of the author's applications. We also manually verified that they do not perform cryptographic operations.

8.3 Asymmetric Cryptography

Now, we want to detect asymmetric cryptography. As stated in Section 2.8.2, some of the most prominent asymmetric encryption algorithms include RSA, ElGamal, and elliptic curve cryptography (ECC). In addition, some signature algorithms, like the Digital Signature Algorithm (DSA), have similar asymmetric properties since they have been built upon asymmetric encryption mechanisms.

In contrast to symmetric cryptography, most asymmetric block ciphers utilize considerably *larger* cryptographic keys. For example, for RSA, key lengths of 2048, 4096, or even 8192 bits are very common. In order to perform mathematical operations on such large numbers, many cryptographic Java libraries utilize the *BigInteger*[1] class, which provides an efficient implementation for arithmetic operations on such large numbers. If this *BigInteger* implementation is not used, one has to manually allocate corresponding arrays of basic data types (i.e. byte arrays) that hold these large numbers. Then, all mathematical operations required by the algorithm have to be correctly implemented. Since this process is quite time-consuming and error-prone, all Java implementations of the RSA algorithm that we manually dissected utilize the more convenient *BigIntger* implementation.

Many asymmetric encryption algorithms can be implemented in just a few lines of code, like the simple RSA example shown in Listing 2.7. The RSA implementation included in Bouncy Castle is given in Listing 8.1. This code actually houses two different implementations of the RSA algorithm:
The basic algorithm, which has already been described in Section 2.8.2, can be seen in line 36. The other branch of the `if`-condition (lines 3 to 33) utilizes the *Chinese Remainder Theorem* to speed up the computation. This theorem requires a little more code though.

For our analysis process, this means that keeping track of the BigInteger implementation could be very useful in order to detect most asymmetric cryptosystems. Thus, we track the usage of classes included in `java.math`, which also contains the BigInteger class, for both method calls and local variables. Since opcodes are always very important, we also include an opcode histogram.

8.3.1 Analysis Configuration

Table 8.2 shows the configuration used for this analysis. Again, a single method instance feature layer is utilized. All methods with more than 5 opcodes are analyzed. Since some asymmetric encryption algorithms, including RSA, can be implemented with just a few operations, we had to lower the number of minimum opcodes compared to the symmetric case.

For this analysis, we utilize the opcodes, as well as certain method calls and local variables. Again, the opcodes will be grouped; mathematical opcodes are added directly and in addition, we also add all *CMP*, *MOV*, and *IF* operations. Table A.1 lists all opcodes and the used opcode group names. The grouped opcodes are then added as a normalized histogram.

As already explained, for method calls this analysis filters for API calls to classes included in `java.math`. For this API calls, we use the "basic feature value" representation (Section 6.5.3) and the full method call name, including the package- and class name, as feature values.

[1] `http://developer.android.com/reference/java/math/BigInteger.html`

```java
public BigInteger processBlock(BigInteger input) {
  if (key instanceof RSAPrivateCrtKeyParameters) {
    //
    // we have the extra factors, use the Chinese Remainder Theorem - the author
    // wishes to express his thanks to Dirk Bonekaemper at rtsffm.com for
    // advice regarding the expression of this.
    //
    RSAPrivateCrtKeyParameters crtKey = (RSAPrivateCrtKeyParameters)key;

    BigInteger p = crtKey.getP();
    BigInteger q = crtKey.getQ();
    BigInteger dP = crtKey.getDP();
    BigInteger dQ = crtKey.getDQ();
    BigInteger qInv = crtKey.getQInv();

    BigInteger mP, mQ, h, m;

    // mP = ((input mod p) ^ dP)) mod p
    mP = (input.remainder(p)).modPow(dP, p);

    // mQ = ((input mod q) ^ dQ)) mod q
    mQ = (input.remainder(q)).modPow(dQ, q);

    // h = qInv * (mP - mQ) mod p
    h = mP.subtract(mQ);
    h = h.multiply(qInv);
    h = h.mod(p);               // mod (in Java) returns the positive residual

    // m = h * q + mQ
    m = h.multiply(q);
    m = m.add(mQ);

    return m;
  }
  else {
    return input.modPow(key.getExponent(), key.getModulus());
  }
}
```

Listing 8.1: Bouncy Castle RSA implementation – excerpt from `org.bouncycastle.crypto.engines.RSACoreEngine`.

Targeted functionality:	**Asymmetric cryptography**
Targeted components:	Methods
Feature Layer:	Single feature layer
Class filter:	–
Method filter:	More than 5 opcodes
Method call inclusion depth:	0
Features:	Opcodes, method calls, local variables
Opcode grouping:	Mathematical opcodes, CMP, MOV, IF, others ignored
Opcode feature values:	Opcode group name
Opcode representation:	Histogram
Method call filtering:	API calls to `java.math`
Method call feature values:	Full method call name
Method call representation:	Basic feature value
Local variable filtering:	Classes in package `java.math`
Local variable feature values:	Full variable type class name
Local variable representation:	Basic feature value
Machine learning algorithm:	SVM

Table 8.2: Analysis configuration: asymmetric cryptography detection

Similar to method calls, we also consider local variables as features. Again, we are only interested in classes contained in `java.math`, the features are added using the "basic feature value" representation, and the full class name is used as feature value.

8.3.2 Training Data

For asymmetric training data, we utilize the Bouncy Castle implementations of *RSA* and *ElGamal*. Thus the two `processBlock` methods of the `RSACoreEngine` and `ElGamalEngine` (in the package `org.bouncycastle.crypto.engines`) have been labeled as asymmetric cryptography. In addition, 200 "normal" methods have been randomly selected that do not contain asymmetric cryptography.

8.4 SMS Handling

Android offers a convenient interface to listen to incoming SMS messages (see Section 2.5. Now, we have created an analysis that detects such broadcast receivers that handle incoming SMS messages. They can be categorized as either SMS sniffers or catchers, depending on whether they abort the broadcast. Hence, the main difference between these two types is, that SMS catchers call `abortBroadcast()` in order to cancel the message broadcast. Other than that, the performed operations could be identical. Both types can read and process the contents of the SMS message in the same way. Since the call to `abortBroadcast()` can be detected by a separate static analysis approach that checks all method calls invoked by the given broadcast receiver, we do not directly distinguish between sniffers and catchers. Instead, our analysis simply labels any receiver that handles SMS broadcasts as "SMS receiver".

Furthermore, it has to be noted that it is very hard to detect sniffers that do not read the contents of the incoming message or perform actions with these contents, since any receiver could theoretically be registered to SMS events, even if it performs completely unrelated actions. Thus, we do not consider such receivers for this analysis process.

As shown in Felt et al. [15], a third of all applications is overprivileged. Thus, these applications require permissions that they do not actually require. By using our analysis, we can detect whether actual SMS code is included in the application, and thus, if the application requires the `SMS\_RECEIVED`-permission, which might be requested by the application.

For this analysis process, we do not rely on the permissions requested by the application, since the goal of this analysis is to detect SMS functionality just by analyzing the Dalvik executable. Similarly, we also do not consider receiver information found in the Android manifest. However, for a real-world scenario, the requested permissions as well as information from the Android manifest, like statically defined SMS receivers, could be added as well.

In future work, additional SMS analysis plugins could be developed that are able to detect more specific SMS code, like receivers that allow to remotely control the device via SMS messages, or analysis approaches that are able to detect "legitimate" SMS code used for normal SMS applications.

8.4.1 Analysis Configuration

Figure 8.3 shows the configuration for this SMS analysis plugin. Since we only want to analyze custom broadcast receivers, a class filter for broadcast receivers is deployed. This filter traverses

Targeted functionality:	**SMS handling**
Targeted components:	Methods
Feature Layer:	Single feature layer
Class filter:	First external superclass is `BroadcastReceiver`
Method filter:	`onReceive`-methods
Method call inclusion depth:	1
Features:	Opcodes, method calls, local variables
Opcode grouping:	Mathematical opcodes separately, GET, PUT, CMP, MOV, IF, others ignored
Opcode feature values:	Opcode group name
Opcode representation:	Histogram
Method call filtering:	Android API calls to `android.telephony.*`
Method call feature values:	Package name
Method call representation:	Basic feature value
Local variable filtering:	Local variables from `android.telephony.*`
Local variable feature values:	Class name
Local variable representation:	Basic feature value
Machine learning algorithm:	SVM

Table 8.3: Analysis configuration: detecting SMS functionality.

the class hierarchy until the first external superclass, not implemented by the application itself, is found. If this external superclass is `BroadcastReceiver`, the class is marked for analysis. Then, a method filter that only allows `onReceive`-methods is applied, since this method is the entry point for incoming broadcast messages.

The features extracted from the `onReceive` methods are: opcodes, local variables, and method calls. First, the grouped opcode histogram is added according to the opcode groups given in Table A.1. For local variables and method calls, we consider all data included in the `android.telephony.*` package provided by the Android system. This package includes, amongst others, the `SmsMessage` class utilized by many SMS receivers, as well as other related classes. For both of these feature types, the basic feature value representation is used, the feature values for method calls being the package name, and for local variables the variable type (i.e. the full class name). The method call inclusion depth has been set to 1, so that we also detect SMS code in methods called by `onReceive`, and a SVM is used for machine learning purposes.

8.4.2 Training Data

The training data has been generated as follows: In order to retrieve SMS code, we dissected Android applications that require the `SMS_RECEIVED`-permission. For each broadcast receiver, we checked whether the class name contains “SMS”. Then, we manually checked whether the receiver indeed handles incoming SMS messages.

For “normal” broadcast receivers, we only considered applications that do not request the SMS permission. For each broadcast receiver we then also checked if the class name does not contain “SMS”. If these two criteria have been met, we used the corresponding receiver as “normal” training data. In total, we gathered 34 SMS broadcast receivers and 220 normal receivers, which we used to train and to evaluate the model.

Chapter 9

Evaluation

This chapter presents the evaluation results of the analysis plugins elaborated in the previous chapter. Each of these analysis plugins has been evaluated in terms of accuracy, by performing both an automated evaluation as well as a manual, empirical evaluation.
Section 9.1 gives more details on the used evaluation methodology. The following sections then present the evaluation results for all analysis plugins. We also analyzed the impact of obfuscation and code optimization on the analysis results, which will be presented in Section 9.5.

Finally, the general performance of the Semdroid framework and the Semantic Pattern Analysis is assessed in Section 9.6; the execution times for different analysis plugins are given for both on-device and PC-based analysis.

9.1 Evaluation Process

The evaluation process consists of two parts: automated- and manual evaluation. First, the corresponding analysis will be automatically evaluated using pre-defined evaluation applications. Then, a second evaluation test set is analyzed by the analysis under evaluation and the analysis results are manually checked for irregularities and conspicuous analysis results.

9.1.1 Automated Evaluation

The automated evaluation process looks as follows: First, we manually select and dissect all evaluation applications. The functionality of the components has to be assessed for all applications and a selection of these components has to be labeled according to this functionality. Then, we note the application names, the component names and the corresponding labels in a text file. This text file is then parsed by the evaluation analysis, which labels all application components according to this mapping.

The evaluation process itself is performed by the Semdroid evaluation framework presented in Section 4.6. The framework automatically evaluates the analysis using the given evaluation analysis, which loads the label mapping file created earlier. The resulting HTML evaluation report contains detailed evaluation results for the current analysis, which are then presented in this thesis.

For analysis plugins that utilize *App instances*, a different evaluation analysis is available: Applications used for the evaluation process are placed in a folder named after the desired label. The evaluation analysis then labels the App instances according to the folder the application is placed in.

9.1.2 Manual Evaluation

After the automated evaluation process has been conducted, a second, manual evaluation is performed. A second evaluation dataset is handed to the analysis under evaluation. We then examine the analysis results and present notable findings. In order to assess the functionality of selected components, we use the decompilation tools discussed in Section 2.7, including *dex2jar* and *JD-GUI*.

The applications used for this manual evaluation process have been collected from various categories on Google Play, mostly from the top free applications of the corresponding category. Detailed information on the actual application set used for the evaluation will be given in the corresponding sections for the analysis plugins.

9.2 Symmetric Cryptography

The first analysis, which detects symmetric cryptography, has been automatically evaluated using 1000 evaluation methods that perform both symmetric cryptography and normal operations. Then, a manual evaluation, where we analyzed 98 password safes found on Google Play, is used to further assess the capabilities of this analysis.

9.2.1 Automated Evaluation

First, we manually labeled known cryptographic code, being methods where the actual cryptographic operations are performed. For this evaluation data, *Bouncy Castle*, an open source cryptographic library, has been used. The package `org.bouncycastle.crypto.engines` contains several symmetric block cipher engines. Methods of these engines that perform the encryption and decryption process have been tagged as cryptographic code. In addition, we also labeled parts of the ciphers and methods related to cryptographic round key generation required by many symmetric block ciphers. Furthermore, `org.spongycastle.crypto.digests` contains hash function engines. Since the structure of hash functions and block ciphers can be very similar (some hash functions even are based on block ciphers), methods that perform hash operations have been added as well. In total, we extracted 67 methods from these two packages.

In addition to Bouncy Castle, we manually examined selected Android applications and labeled cryptographic components found in these applications. In particular, we used 24 methods of the Cryptix provider[1] and 14 other cryptographic methods, including custom AES, SHA1, and Bcrypt implementations.

Furthermore, a selection of non-cryptographic methods has been made, including different functionality ranging from activities over services to various other code. We also included 20 “normal” methods that perform asymmetric-key cryptography. Since we just want to detect symmetric block ciphers and hash functions, the analysis plugin should label asymmetric code as “normal”.

The resulting 1000 methods, split into 100 cryptographic methods and 900 normal ones (since custom cryptographic implementations are quite rare), have been used to evaluate the Semantic Pattern Analysis for cryptography detection.

[1] `http://ds0.cc.yamaguchi-u.ac.jp/~joji/doc/cryptix3.2.0/cryptix/provider/cipher/package-summary.html`

Label	Components	Correct	Wrong	Correct [%]
Symmetric cryptography	100	99	1	99%
Normal	900	900	0	100%
Total	1000	999	1	99.9%

Table 9.1: Automated evaluation results for symmetric cryptography detection. 100 cryptographic methods and 900 normal methods have been tested.

Type	Group	Count	Total
Symmetric cryptography	Cryptography	1740	3653
	Non-cryptographic hash functions	46	
	Encoding schemes and data notations	765	
	Compression functions	13	
	Other	240	
	Obfuscated	849	
Normal			55534
Total			59215

Table 9.2: Analysis results for 98 password safes: symmetric cryptography.

The results can be found in Table 9.1. All normal methods have been labeled correctly by the analysis. The cryptographic code detection has been great as well – 99 out of 100 methods have been correctly labeled. The only method not detected is the `setKey` method of the Camellia symmetric block cipher implementation found in Bouncy Castle (`org.bouncycastle.crypto.engines.CamelliaEngine: void setKey(boolean, byte[])`).

9.2.2 Manual Evaluation

We analyzed 98 password safes found on Google Play and then categorized the resulting methods according to their functionality. Since a password safe should include some form of cryptography to protect the user's login credentials, we focused on these applications since they are more likely to include custom cryptography. If no custom cryptography is found, the application could either utilize the built-in Android libraries, or, in the worst case, does not employ cryptography at all.

In total, 59215 methods have been analyzed. Methods with less than 30 opcodes have been omitted, since we assume that symmetric cryptography requires at least 30 operations. It has to be noted that since we analyzed 98 different applications, it is possible that the same methods, or similar methods, can be found in multiple applications. For example, several password safes utilize the Bouncy Castle cryptographic library – either directly, or as a repackaged variant. We did not group these methods together, since they have different package names, or could be from different versions of Bouncy Castle.

The analysis classified 55534 methods as "normal" code, and the remaining 3653 methods as symmetric cryptography. Then, we manually examined these cryptographic methods and categorized them accordingly. Table 9.2 shows the different categories. 849 of these methods have been obfuscated and were not further analyzed. This leaves 2804 remaining methods, which were

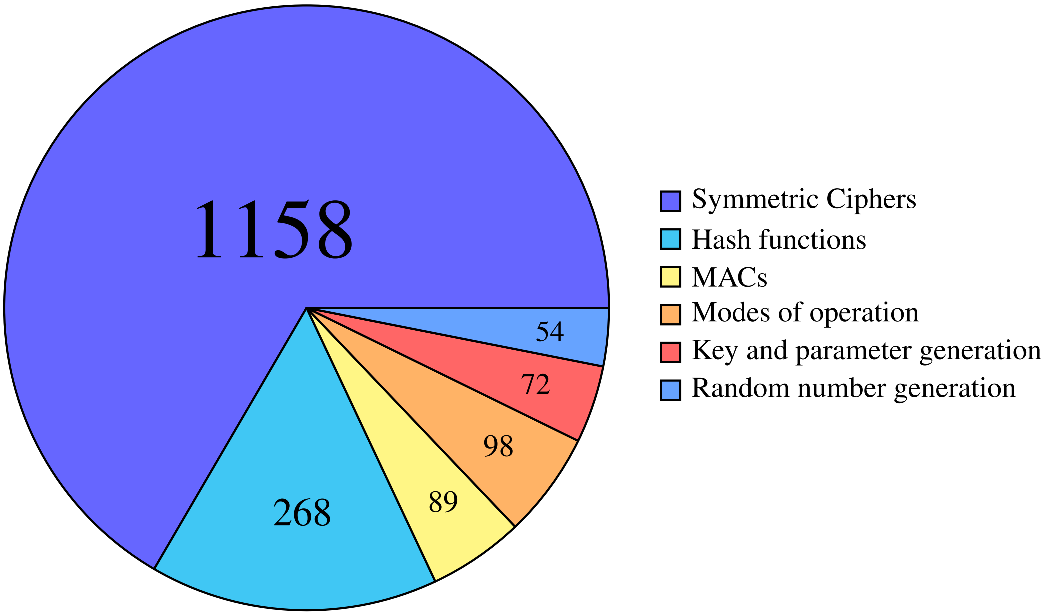


Figure 9.1: Manual evaluation results: Symmetric cryptography categories.

categorized into symmetric cryptography, non-cryptographic hash functions, various encodings, compression functions, and other methods.

Cryptography (1740 methods)

The first subgroup, custom symmetric-key cryptography, accounts for the majority of methods labeled as custom cryptography. We further categorized these 1740 methods according to their functionality. Figure 9.1 shows these subcategories. 1158 methods include code used for symmetric block ciphers. Similarly, 268 methods are used for cryptographic hash functions, and 89 for message authentication codes. Furthermore, code that implements different modes of operation (like CBC) can be found in 98 methods. Some pseudorandom number generators (PRNGs) also perform operations on given data that are similar to hash functions and to cryptographic code in general. Thus, 54 methods that are involved in generating random numbers have been found. Finally, 72 methods contain code used for key- and parameter generation.

- **Symmetric ciphers** (1158 methods)
  The majority of cryptographic methods, 1158, are used to build symmetric ciphers. Since the analysis tries to detect cryptography in general, parts of these ciphers are also labeled as cryptographic code since they are involved in the process.

  - **Bouncy Castle implementations**
    Some applications included the original Bouncy Castle library, or repackaged versions, like *Spongy Castle*[2], in order to avoid name conflicts with the Bouncy Castle implementation included in Android. A few app developers also renamed Bouncy Castle on their own, as, for example, by using the package name `org.bownzycastle.*`. Most of the cryptographic code found in applications originates from Bouncy Castle.

[2] http://rtyley.github.io/spongycastle/

  - **Custom implementations**
    A few implementations included custom cryptography, mostly custom AES implementations. Furthermore, a few applications included cryptographic code found in libraries. One example for such cryptographic code is *UnixCrypt*[3], which is included in some Apache libraries. According to its documentation, this class implements DES. Thus, the analysis has been correct.
  - **Parts of ciphers**
    Many implementations of encryption algorithms are split into several methods. For example, the AES algorithm consists of four parts: *SubBytes*, *ShiftRows*, *MixColumns*, and *AddRowKey*. Many implementations of this algorithm, including Bouncy Castle, have a similar structure where each of these steps is encapsulated within a separate method. Not only does the analysis correctly recognize the main method that performs the encryption and calls all subsequent methods as cryptographic code, but it also tags all parts involved in this encryption process as cryptography.
  - **Round key generation**
    Many ciphers, like AES, require so-called round keys, which have to be derived from the encryption key. Most of these key generation methods are recognized as well.
  - **LFSR**
    Linear feedback shift registers, in short LFSRs, have similar structures to symmetric-key ciphers. Thus, some of them are detected by the analysis. As its name suggests, an LFSR utilizes shift operations and XOR to add the results. The *Grain* stream cipher utilizes an LFSR for its computations. Hence, the analysis detected two LFSR implementations included two Bouncy Castle Grain implementations, namely *Grainv1Engine* and *Grain128Engine*, which can be found in the package `org.spongycastle.crypto.engines`.

- **Hash functions** (268 methods)
  Our analysis found 268 methods required by cryptographic hash functions. The majority of these methods have their roots in Bouncy Castle implementations. In addition, 10 custom SHA implementations (both SHA-0 and SHA-1) have been found, contributing with a total of 20 methods, since different parts of the implementations have been labeled separately.

- **Message Authentication Codes (MACs)** (89 methods)
  In addition, the analysis found 89 methods that implement MACs. All of these methods have been found as parts of Bouncy Castle, in the package `org.bouncycastle.crypto.macs.*` or similar repackaged variants.

- **Modes of operation** (98 methods)
  Various modes of operation, as for example cipher-block chaining (CBC), Galois/Counter Mode (GCM), or several output feedback modes have been found. Again, all of these methods have been included as part of Bouncy Castle, in the package `org.spongycastle.crypto.modes.*`.

- **Key and parameter generation** (72 methods)
  We found two custom password-based key derivation functions: BCrypt [42], which is based on Blowfish, and scrypt [41]. The custom BCrypt implementation has been found in one password safe, whereas the scrypt implementation is included in two other safes, as part of

[3] `http://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/UnixCrypt.html`

the Spongy Castle library.
Furthermore, key generators for DES, IDEA, and LOKI91 have been detected by the analysis, as well as key factories and parameter generators included in the Bouncy Castle library.

- **Random number generation** (54 methods)
  Random number generation is a problem related to cryptography. Random numbers are required for various tasks, including key- and salt generation. Hence, Bouncy Castle includes several pseudorandom number generators (PRNGs), found in package `org.spongycastle.crypto.prng.*`. For instance, the VMPC random number generator [69] included in this package has been detected by the analysis. VMPC itself is a one-way function and a stream cipher. Thus, it has been correctly detected by the analysis.
  In addition to VMPC, `org.spongycastle.util.test.FixedSecureRandom` has been detected as well. As the name states, it creates a fixed secure random sequence used for testing. Finally, the analysis also detected a custom random number generator not included in Bouncy Castle, which implements the R250 random number generation algorithm [29].

Non-Cryptographic Hash Functions (46 methods)

In addition to cryptographic hash functions, there are also some hash applications, where non-cryptographic hash functions can be used, like for hash-based lookup. An example for this would be *MurmurHash3*[4], which has been found in three password safes as part of the Google Guava project[5]. Google Guava includes two implementations of this hash function, for different digest sizes of 32 and 128 bits. Since the structures of these non-cryptographic hash functions can be very similar to cryptographic ones, it makes sense that they are detected by the analysis.

Encoding Schemes and Data Notations (765 methods)

Various encoding schemes and data notations are recognized as cryptographic code. In total, 765 methods belong to this category. The number of methods belonging to this category is quite high because many password safes included several encoding schemes. Figure 9.2 shows the different encoding schemes that we found. The most prominent one is Base64, with 243 methods, followed by UTF-8 and JSON. Since these encodings manipulate given input data and transform it to a different output, it makes sense that the analysis detects these encodings as cryptographic code. The encoding process to a certain format is similar to an "encryption" process (which does not employ cryptographic keys of course and is not secure), while the decoding process has similarities to cryptographic "decryption". In both cases, the given input data is transformed to a different output.

- **Base64**
  Base64 [28] is a binary-to-text encoding scheme utilized by many applications. The Base64 implementation found in Bouncy Castle is shown in Listing 9.1. The Semantic Pattern Analysis labels many of these implementations as cryptographic code. The code looks quite similar to cryptography; it is very mathematics-heavy, it requires many shift operations as well as bitwise AND calculations. In addition, it utilizes an encoding table, similar to the substitution boxes (S-boxes) found in many block ciphers, including AES.

[4] https://code.google.com/p/smhasher/wiki/MurmurHash3

[5] https://code.google.com/p/guava-libraries/

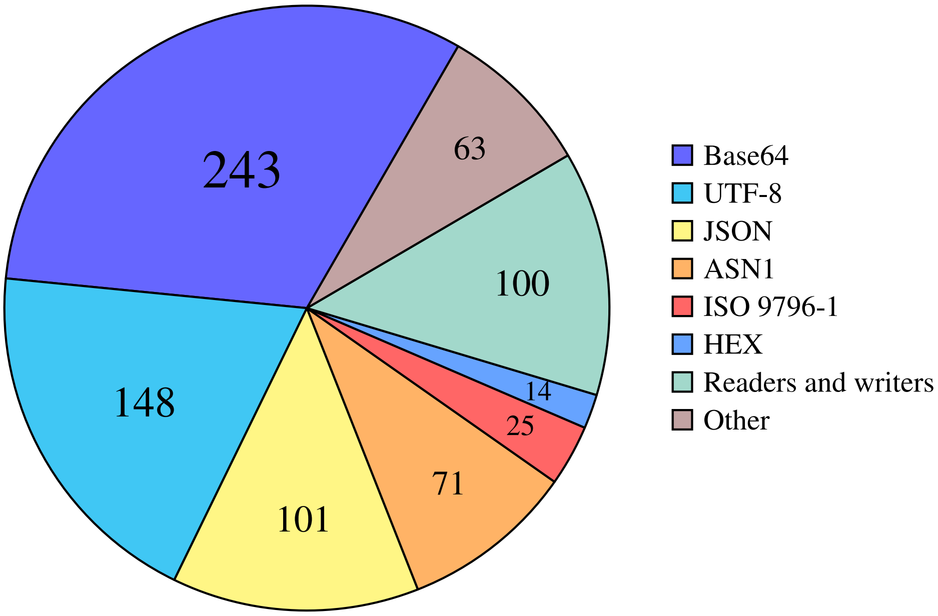


Figure 9.2: Manual evaluation results: Encodings.

- **JSON**
  JSON[6] is short for for JavaScript Object Notation and is used to interchange data between computers. Some methods required for serializing objects, to read and write objects, and to convert data types are recognized as cryptographic code. Since such JSON code is included in 13 different applications, a total of 101 methods are found.

- **ISO 9796-1 Padding**
  The ISO 9796-1 padding is a padding scheme used for asymmetric ciphers. According to Menezes, Vanstone, and Oorschot [34], the message is padded, extended, and redundancy is added. Thus, despite being used for asymmetric cryptography, the padding itself has similarities to symmetric cryptography. The code of this padding can be found in Bouncy Castle, in `org.bouncycastle.crypto.encodings.ISO9796d1Encoding`.

- **Readers and writers**
  Similarly, input stream readers and output stream writers have to transform the data read from and written to their respective streams. One example that has been found in three password safes is `org.spongycastle.bcpg.ArmoredOutputStream`, which performs Base64 encoding and CRC computations.

- **Other encodings**
  Similar to the Base64 encoding, other encoding schemes, like the Hex encoding included in Bouncy Castle, utilize encoding tables and shift operations. The BER and DER encodings, as well as some other rules found in the ASN.1 standard have been labeled as cryptography as well.

[6] `http://www.json.org/`

Compression Functions (13 methods)

Compression algorithms also transform a given input to a different (smaller) output. Thus, it makes sense that these implementations are found by the analysis. In practice, the analysis found a *ZIP* implementation included in one application, and *bzip2* implementations in two other applications. In total, 13 methods have been detected that are required by these three implementations.

Others (240 methods)

Finally, the analysis detected 240 other methods not necessarily related to the previous categories. For example, checksums have been detected, as well as other methods containing many mathematical operations similar to symmetric cryptosystems.

- **Checksums**
  Similar to non-cryptographic hash functions, some checksums also have comparable structures. To be specific, our analysis found an implementation of Adler-32, which is one of these checksum algorithms [8].

- **False positives**
  Finally, the analysis also incorrectly classified a few methods, mainly concerned with many mathematical functions, as for example the method *findCornerFromCenter* found in the *MonochromeRectangleDetector*[7], which tries to find a corner of a barcode. Overall, it has to be said that such false positives are very rare.

9.2.3 Conclusions

Summarizing these results, it is safe to say that the performance of this analysis is very good. Actual cryptographic code used for symmetric-key cryptosystems is found with a very high detection rate. Code for asymmetric-key cryptosystems is correctly classified as "normal" code. Since it is hard to strictly define "cryptographic code", the analysis also finds similar code not necessarily used in the context of cryptography. All other methods tagged as cryptographic code, like the various encodings or compression functions, perform mathematical operations directly on the given input data, very similar to encryption algorithms. These components might not necessarily offer strong cryptography or use cryptographic keys for the transformation process, but the characteristics are still similar.

In general, this analysis can be characterized to detect components that apply mathematical operations on a given input, which transform it to a different output. These operations are similar to the operations used by symmetric block ciphers and hash functions.

It also has to be noted that many applications did not include any custom cryptography at all, even security-related applications, like password safes or mobile banking applications. By manually examining some of these applications, it can be seen that many of them utilize the built-in cryptographic libraries provided by the Android system. Many utilize the standard Java cryptographic API to retrieve cipher- or hash instances.

[7] https://code.google.com/p/zxing/source/browse/trunk/core/src/com/google/zxing/common/detector/MonochromeRectangleDetector.java?r=1003

```java
/**
 * encode the input data producing a base 64 output stream.
 *
 * @return the number of bytes produced.
 */
public int encode(byte[] data, int off, int length, OutputStream out)
  throws IOException {
  int modulus = length % 3;
  int dataLength = (length - modulus);
  int a1, a2, a3;

  for (int i = off; i < off + dataLength; i += 3) {
    a1 = data[i] & 0xff;
    a2 = data[i + 1] & 0xff;
    a3 = data[i + 2] & 0xff;

    out.write(encodingTable[(a1 >>> 2) & 0x3f]);
    out.write(encodingTable[((a1 << 4) | (a2 >>> 4)) & 0x3f]);
    out.write(encodingTable[((a2 << 2) | (a3 >>> 6)) & 0x3f]);
    out.write(encodingTable[a3 & 0x3f]);
  }

  /*
   * process the tail end.
   */
  int    b1, b2, b3;
  int    d1, d2;

  switch (modulus) {
  case 0:        /* nothing left to do */
    break;
  case 1:
    d1 = data[off + dataLength] & 0xff;
    b1 = (d1 >>> 2) & 0x3f;
    b2 = (d1 << 4) & 0x3f;

    out.write(encodingTable[b1]);
    out.write(encodingTable[b2]);
    out.write(padding);
    out.write(padding);
    break;
  case 2:
    d1 = data[off + dataLength] & 0xff;
    d2 = data[off + dataLength + 1] & 0xff;

    b1 = (d1 >>> 2) & 0x3f;
    b2 = ((d1 << 4) | (d2 >>> 4)) & 0x3f;
    b3 = (d2 << 2) & 0x3f;

    out.write(encodingTable[b1]);
    out.write(encodingTable[b2]);
    out.write(encodingTable[b3]);
    out.write(padding);
    break;
  }

  return (dataLength / 3) * 4 + ((modulus == 0) ? 0 : 4);
}
```

Listing 9.1: Bouncy Castle Base64 encoding – excerpt from `org.bouncycastle.util.encoders.Base64Encoder`.

Label	Components	Correct	Wrong	Correct [%]
Asymmetric cryptography	20	20	0	100%
Normal	980	980	0	100%
Total	1000	1000	0	100%

Table 9.3: Automated evaluation results for asymmetric cryptography detection. 20 methods implementing asymmetric cryptography and 900 normal methods have been tested.

Type	Count
Asymmetric cryptography	512
Normal	2415001
Total	2415513

Table 9.4: Analysis results for 98 password safes: asymmetric cryptography.

9.3 Asymmetric Cryptograpy

Compared to the symmetric case, it is much harder to find suitable asymmetric evaluation data. Since asymmetric-key ciphers, like RSA, can be easily implemented with a single line, or with a couple of lines, we do not have many different implementation possibilities (at least not many that make sense and that can be found in production code).
Similarly, we did not find many different custom implementations in our evaluation applications.

9.3.1 Automated Evaluation

Again, we labeled methods implementing asymmetric cryptography as “asymmetric cryptography”, and other methods as “normal” code. Signature algorithms based on asymmetric cryptography, like DSA, have been considered as asymmetric code as well. Since we did not find many implementations of asymmetric-key ciphers, the evaluation test set is smaller than for the symmetric case. In total, we used 20 methods implementing asymmetric cryptography and 980 randomly selected “normal” methods. These “normal” methods also include symmetric cryptography and hash functions, which should ideally not be detected as asymmetric cryptography.

The evaluation results can be found in Table 9.3. All methods have been correctly classified; it did not label any symmetric cryptography or hash function as asymmetric code, and all methods containing asymmetric cryptography have been found.
It has to be noted though, that this does not mean that the analysis is perfect for real-world applications. Thus, the next section present the analysis results for additional applications.

9.3.2 Manual Evaluation

The automated evaluation results yielded excellent results. But what about applications in the wild? In order to get a better understanding of the analysis performance, we performed the same manual evaluation as for the symmetric case and manually examined the analysis results for 98 password safes.

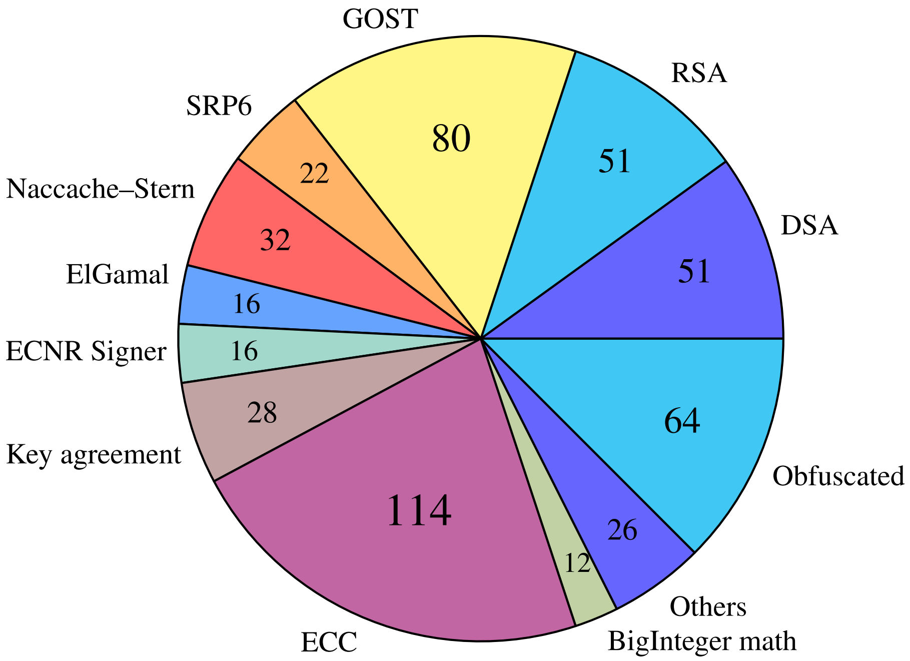


Figure 9.3: Manual evaluation results: Asymmetric cryptography details.

Table 9.4 shows the analysis results for these 98 password safes. Since asymmetric cryptography can be implemented very efficiently, with just a couple of lines, the analysis targets all methods that perform more than 5 operations, as opposed to the minimum of 30 operations for symmetric cryptography detection. Hence, this analysis plugin examined significantly more methods than the previous plugin for symmetric code.

Again, we looked at methods the analysis labeled as asymmetric cryptography and then categorized these methods accordingly. Figure 9.3 shows these different categories. 64 methods have been obfuscated and we did not further analyze them. In general, it can be observed that a majority of the detected methods are parts of the Bouncy Castle library, or repackaged Bouncy Castle variants.

- **RSA**
  Of the 98 password safes, 8 included RSA code from Bouncy Castle. Furthermore, we found two custom RSA implementations. Both of these two custom implementations provide methods to perform RSA en- and decryption, as well as to sign content and to verify signatures. In addition, a test case with the name *TestRSA* has been detected by our analysis, as well as an RSA key pair generator. In total, 51 methods include RSA computations.
- **ElGamal**
  Most ElGamal implementations are from repackaged Bouncy Castle libraries. Only a single other custom implementation has been detected; this implementation offers method for en- and decryption, as well as for signing and verifying a signature. Furthermore, a test case for this custom implementation has also been detected, which verifies the correctness of the implemented algorithm. Also, a class that generates ElGamal key pairs is included in this custom implementation as well.
- **DSA**
  The DSA implementations detected by the analysis are again mostly from (repackaged)

Bouncy Castle libraries. However, we also detected two custom implementations, found in two different applications. The first one can be found in Apache libraries, namely *SHA1withDSA_SignatureImpl*[8]. As the name states, this algorithm uses SHA1 and DSA. Thus, the signature generation and verification methods have been detected by our analysis. The second custom implementation is called *RawDSASignature* and holds a standard DSA algorithm.

- **Other signature algorithms**
  Furthermore, several other signature algorithms have been detected. The package `org.bouncycastle.crypto.signers` contains several of these signature algorithms. For the classes *DSASigner* and *GOST3410Signer* (based on the GOST R 34.10-94 Signature Algorithm, see Leontiev and Shefanovski [31]) the two methods `generateSignature` and `verifySignature` have been labeled as asymmetric cryptography. For *ECDSASigner*, *ECNRSigner*, and *ECGOST3410Signer* the `generateSignature` methods have been detected. All of these methods perform asymmetric cryptography.

- **Naccache–Stern cryptosystem**
  The public-key cryptosystem described in Naccache and Stern [36] is based on the higher residuosity problem. This cryptosystem also uses an RSA modulus n, which consists of two large prime numbers multiplied together. The encryption process itself utilizes the Chinese Remainder Theorem. Thus, the computations are similar to RSA, which explains why it has been correctly recognized as asymmetric-key cryptography.
  From the 98 password safes, 8 include the Bouncy Castle Naccache-Stern implementation, and for each of these implementations, several methods involved in the en- and decryption process, as well as key generation mechanisms have been detected. In total, this cryptosystem accounts for 32 methods.

- **Elliptic Curve Cryptography**
  Elliptic Curve Cryptography (ECC) contributes with 114 methods. Since the Bouncy Castle ECC implementation relies heavily on BigIntegers, these methods have been correctly classified as asymmetric-key cryptography.

- **SRP**
  The *Secure Remote Password Protocol* [63] is an asymmetric key exchange protocol. Bouncy Castle includes this SRP-6a protocol, impelemented according to Wu et al. [62]. The *Semantic Pattern Analysis* labeled several methods of this implementation (for the server and for the client) as asymmetric cryptography. A total of 6 password safes include methods from this SRP implementation. Listing 9.2 shows the code of one of these methods, namely `generateServerCredentials()`. It can be seen, that this method really performs asymmetric cryptography.
  Furthermore, another password safe includes a custom SRP implementation, that operates on a custom *BigInteger* class, called *UBigHexInteger*.

- **Key agreement**
  Key-agreement protocols are used in order to create a shared key between two parties. One common key-agreement protocol is the Diffie-Hellman key exchange (see Menezes, Vanstone, and Oorschot [34]). It is based on the same operations as RSA, and thus detected by the *Semantic Pattern Analysis*. Furthermore, other key-agreement protocols, like Elliptic Curve Menezes-Qu-Vanstone (ECMQV), are detected as well.

[8] org.apache.harmony.security.provider.crypto.SHA1withDSA_SignatureImpl

```
/**
 * Generates the server's credentials that are to be sent to the client.
 * @return The server's public value to the client
 */
public BigInteger generateServerCredentials() {
  BigInteger k = SRP6Util.calculateK(digest, N, g);
  this.b = selectPrivateValue();
  this.B = k.multiply(v).mod(N).add(g.modPow(b, N)).mod(N);
  return B;
}
```

Listing 9.2: Bouncy Castle `org.bouncycastle.crypto.agreement.srp.SRP6Server` excerpt for generating server credentials, according to SRP-6a.

Again, 8 applications include the respective Bouncy Castle implementations, which have been detected. In addition, one application includes a custom SSL Diffie-Hellman implementation.

- **Zero-knowledge proof (ZKP)**
  According to Schneier [45], a zero-knowledge proof is used by a party to prove to a second party that a given statement is true (e.g., that the first party knows a secret), without revealing any additional information. In order to implement such zero-knowledge proofs, strategies similar to asymmetric encryption mechanisms can be used. For more information, we refer to Schneier [45] and Menezes, Vanstone, and Oorschot [34]. One of the applications we have analyzed included such a custom ZKP implementation by Mozilla, called *JPakeCrypto*[9]. Three methods found in this class, namely `createZkp`, `checkZkp`, and `round2` have been labeled as asymmetric cryptography. The code for the first two methods is shown in Listing 9.3. The operations used for creating the ZKP as well as for verifying the ZKP is very similar to asymmetric cryptography. Thus it makes sense that this code has been detected by the analysis.

- **Fractions**
  Another method labeled as asymmetric cryptography is called `addSub` and can be found in `org.apache.commons.lang3.math.Fraction`. The implementation of this method is given in Listing 9.4. This implementation utilizes many BigIntegers and uses them for various mathematical operations similar to asymmetric encryption algorithms. More information on the implemented add and subtract algorithm can be found in Knuth [30, Section 4.5.1].

- **Others**
  Finally, 38 other methods have been tagged, 12 of which are part of *BigInteger* helper libraries. The other 26 methods also are helper methods that operate on BigIntegers.

9.4 SMS Handling

In order to evaluate this analysis, we performed both an automated- and a manual evaluation process. The next two sections present their respective evaluation results.

[9] `http://dxr.mozilla.org/mozilla-central/source/mobile/android/base/sync/jpake/JPakeCrypto.java`

```
/* This Source Code Form is subject to the terms of the Mozilla Public
 * License, v. 2.0. If a copy of the MPL was not distributed with this
 * file, You can obtain one at http://mozilla.org/MPL/2.0/. */

/*
* Generate the ZKP b = r - x*h, and g^r, where h = hash(g, g^r, g^x, id).
    (We
* pass in gx to save on an exponentiation of g^x)
*/
private static Zkp createZkp(BigInteger g, BigInteger x, BigInteger gx,
  String id, JPakeNumGenerator gen) throws NoSuchAlgorithmException,
      UnsupportedEncodingException {
  // Generate random r for exponent.
  BigInteger r = gen.generateFromRange(Q);

  // Calculate g^r for ZKP.
  BigInteger gr = g.modPow(r, P);

  // Calculate the ZKP b value = (r-x*h) % q.
  BigInteger h = computeBHash(g, gr, gx, id);
  Logger.debug(LOG_TAG, "myhash: " + h.toString(16));

  // ZKP value = b = r-x*h
  BigInteger b = r.subtract(x.multiply(h)).mod(Q);

  return new Zkp(gr, b, id);
}

private static void checkZkp(BigInteger g, BigInteger gx, Zkp zkp)
  throws IncorrectZkpException, NoSuchAlgorithmException,
      UnsupportedEncodingException {

  BigInteger h = computeBHash(g, zkp.gr, gx, zkp.id);

  // Check parameters of zkp, and compare to computed hash. These shouldn't
  // fail.
  if (gx.compareTo(BigInteger.ONE) < 1) { // g^x > 1.
    Logger.error(LOG_TAG, "g^x > 1 fails.");
    throw new IncorrectZkpException();
  }
  if (gx.compareTo(P.subtract(BigInteger.ONE)) > -1) { // g^x < p-1
    Logger.error(LOG_TAG, "g^x < p-1 fails.");
    throw new IncorrectZkpException();
  }
  if (gx.modPow(Q, P).compareTo(BigInteger.ONE) != 0) {
    Logger.error(LOG_TAG, "g^x^q % p = 1 fails.");
    throw new IncorrectZkpException();
  }
  if (zkp.gr.compareTo(g.modPow(zkp.b, P).multiply(gx.modPow(h, P)).mod(P))
      != 0) {
    // b = r-h*x ==> g^r = g^b*g^x^(h)
    Logger.debug(LOG_TAG, "gb*g(xh) = " + g.modPow(zkp.b,
        P).multiply(gx.modPow(h, P)).mod(P).toString(16));
    ... // original source: additional logging
    Logger.error(LOG_TAG, "zkp calculation incorrect.");
    throw new IncorrectZkpException();
  }
  Logger.debug(LOG_TAG, "*** ZKP SUCCESS ***");
}
```

Listing 9.3: JPakeCrypto zero-knowledge proof excerpt – full implementation can be found at `http://dxr.mozilla.org/mozilla-central/source/mobile/android/base/sync/jpake/JPakeCrypto.java`.

```java
/**
 * Implement add and subtract using algorithm described in Knuth 4.5.1.
 *
 * @param fraction the fraction to subtract, must not be <code>null</code>
 * @param isAdd true to add, false to subtract
 * @return a <code>Fraction</code> instance with the resulting values
 * @throws IllegalArgumentException if the fraction is <code>null</code>
 * @throws ArithmeticException if the resulting numerator or denominator
 *   cannot be represented in an <code>int</code>.
 */
private Fraction addSub(Fraction fraction, boolean isAdd) {
  if (fraction == null) {
    throw new IllegalArgumentException("The fraction must not be null");
  }
  // zero is identity for addition.
  if (numerator == 0) {
    return isAdd ? fraction : fraction.negate();
  }
  if (fraction.numerator == 0) {
    return this;
  }
  // if denominators are randomly distributed, d1 will be 1 about 61%
  // of the time.
  int d1 = greatestCommonDivisor(denominator, fraction.denominator);
  if (d1==1) {
    // result is ( (u*v' +/- u'v) / u'v')
    int uvp = mulAndCheck(numerator, fraction.denominator);
    int upv = mulAndCheck(fraction.numerator, denominator);
    return new Fraction
      (isAdd ? addAndCheck(uvp, upv) : subAndCheck(uvp, upv),
       mulPosAndCheck(denominator, fraction.denominator));
  }
  // the quantity 't' requires 65 bits of precision; see knuth 4.5.1
  // exercise 7.  we're going to use a BigInteger.
  // t = u(v'/d1) +/- v(u'/d1)
  BigInteger uvp = BigInteger.valueOf(numerator)
    .multiply(BigInteger.valueOf(fraction.denominator/d1));
  BigInteger upv = BigInteger.valueOf(fraction.numerator)
    .multiply(BigInteger.valueOf(denominator/d1));
  BigInteger t = isAdd ? uvp.add(upv) : uvp.subtract(upv);
  // but d2 doesn't need extra precision because
  // d2 = gcd(t,d1) = gcd(t mod d1, d1)
  int tmodd1 = t.mod(BigInteger.valueOf(d1)).intValue();
  int d2 = tmodd1==0?d1:greatestCommonDivisor(tmodd1, d1);

  // result is (t/d2) / (u'/d1)(v'/d2)
  BigInteger w = t.divide(BigInteger.valueOf(d2));
  if (w.bitLength() > 31) {
    throw new ArithmeticException
      ("overflow: numerator too large after multiply");
  }
  return new Fraction(w.intValue(),
     mulPosAndCheck(denominator/d1, fraction.denominator/d2));
}
```

Listing 9.4: Apache fractions implementation. Excerpt from `org.apache.commons.lang3.math.Fraction` Licensed under the Apache License, Version 2.0.

Label	Broadcast receivers	Correct	Wrong	Correct [%]
SMS receivers	34	34	0	100%
Other receivers	220	219	1	99.54%
Total	254	253	1	99.96%

Table 9.5: Automated evaluation results for detecting SMS broadcast receivers. 10-fold cross-validation has been used.

9.4.1 Automated Evaluation

For the automated analysis, we used the training data as discussed in Section 8.4.2. In total, we have 253 broadcast receivers, split into 34 SMS receivers and 220 other receivers that do not handle SMS broadcasts. We evaluated the analysis using 10-fold cross validation. These evaluation results are given in Table 9.5. On average, all but one receiver have been correctly classified, resulting in 99.96% correctly classified broadcast receivers.

9.4.2 Manual Evaluation

For the manual, empiric evaluation, we analyzed several top free applications from 11 selected categories on Google Play, namely *Arcade & Action*, *Brain & Puzzle*, *Cards*, *Communication*, *Entertainment*, *Finance*, *Live Wallpapers*, *Media & Video*, *Music & Audio*, *Photography*, and *Tools*.

Table 9.6 lists the analysis results and gives the number of detected SMS receivers. Overall, 372 applications have been analyzed, which contained 4397 broadcast receivers. In 31 of these applications, the analysis found at least one SMS receiver. In total, these 31 applications included 53 SMS broadcast receivers. Furthermore, we noticed that 33 application did not include any broadcast receiver at all.

We then manually verified the functionality of these 52 SMS receivers. The 4344 *normal* receivers have not been further analyzed since this task is very time-consuming. Thus, we currently cannot give accurate numbers about the true false negative rate of the analysis.

- **Arcade & Action**
  Out of the 29 games, 4 included SMS receivers. One application included 5 receivers, all others included a single receiver each, resulting in a total of 8 receivers. Normally, one would not expect SMS receivers in games, but 4 of these receivers really listen to incoming SMS messages. One of these receivers is called *BillingSMSReceiver*, thus it can be assumed that it is used for in-app billing via SMS messages.
  The other 4 receivers are tracking receivers, that are activated on application installation. These tracking/installation receivers mainly utilize the *TelephonyManager* included in the `android.telephony` package used for the training process as filtering criteria.

- **Brain & Puzzle**
  Our analysis did not find any SMS receivers in the 26 games of this category, which is the expected result since games normally should not include SMS code.

- **Cards**
  Similarly, of the 30 analyzed card games, only 2 contained SMS receivers. These two receivers are identical, from the same advertisement library, and are used as installation

Category	**Apps**			**Broadcast receivers**		
	Total	**SMS**	**No receivers**	**Total**	**SMS**	**Normal**
Arcade & Action	29	4	0	322	8	314
Brain & Puzzle	26	0	2	273	0	273
Cards	30	2	3	281	2	279
Communication	66	18	5	1063	27	1036
Entertainment	21	2	2	273	2	271
Finance	55	1	6	205	1	204
Live Wallpapers	63	0	4	802	0	802
Media & Video	43	2	10	245	2	243
Music & Audio	18	1	0	281	1	280
Photography	33	1	1	376	1	375
Tools	17	4	0	276	9	267
Total	**372**	**31**	**33**	**4397**	**53**	**4344**

Table 9.6: Analysis results: SMS receivers in different application categories. 11 different application categories have been analyzed. The number of analyzed applications is given for each category, as well as the number of apps that include at least one SMS receiver, and the number of apps without any receiver. Then, the total number of analyzed broadcast receivers is given, which is split up in SMS- and normal receivers.

receivers similar to the false positives of the *Arcade & Action* category. Again, they access the *TelephonyManager* in order to retrieve the device ID. Thus, these two receivers are false positives.

- **Communication**
  Since communication applications are more likely to include SMS code, we used a bigger test set consisting of 66 applications for this category. 18 of these applications included SMS broadcast receivers, 27 in total. Of these 27 receivers, 24 really perform SMS operations, for example, used by replacement SMS applications, and by blacklisting utilities. Three receivers have not been classified correctly; they are again install receivers that read the device ID and unlock special features according to the install referrer.

- **Entertainment**
  Two of the 21 entertainment applications include SMS receivers. One of these receivers is used for in-app purchases via SMS messages. The second one is used to remote-control a media player – but not via SMS messages. Thus, the second receiver is a false positive.

- **Finance**
  Next, 55 finance applications have been analyzed. One out of these applications included a real SMS receiver.

- **Live Wallpapers**
  As expected, no live wallpaper application included SMS code.

- **Media & Video**
  Media and video applications normally should not include SMS receivers either. However, the analysis detected two receivers, both of which really handle SMS messages. One of these receivers is part of a third-party library though, which may not be used by the application – since the application also does not request the SMS permission.

- **Music & Audio**
  Similar to the previous category, only one SMS broadcast receiver has been detected in the 18 applications of this category. However, since this receiver is a call receiver that monitors incoming phone calls instead of SMS messages, this receiver has not been classified correctly.

- **Photography**
  Of the 33 applications, one included an SMS broadcast receiver. This receiver is, again, an installation receiver that requests the device ID, similar to the other false positives previously mentioned.

- **Tools**
  The last category, tools, is again more likely to include SMS code. As the analysis result show, 9 SMS receivers have been detected. Our manual analysis shows, that all of these 9 receivers really are SMS receivers. Three of these applications are mobile security applications, which contribute with 6 SMS receivers. The last application is a utility that allows to send SMS messages from your PC. Thus, three different receivers are implemented to provide this functionality.

In total, of the 53 detected SMS receivers, 12 are false positives. The other 41 receivers really handle SMS messages. Most of the false positives have a similar structure, they access classes found in the `android.telephony.*` package, mostly the *TelephonyManager*. Since we use the package name as feature value for method calls, method calls to the *SmsMessage* class and to the *TelephonyManager* are both mapped to `android.telephony`. Thus, the machine learning algorithms is not able to distinguish between these two classes. By using different feature values, the analysis performance could be further improved; in particular, a great approach could be to use the class name as feature value, but without the package name, since the Android platform provides two different implementations of *SmsMessage* and *SmsManager*, located in `android.telephony` and `android.telephony.gsm`. The latter implementation has been deprecated since API level 4 (Android 1.6), but applications could still use this class. Hence, by using the class name only, both implementations would result in the same feature values.

Furthermore, in future work, the analysis could be refined to be able to differentiate between different SMS-handling scenarios; for example, one refinement could be to detect SMS command receivers, normal SMS applications, or blacklisting applications.

9.5 Obfuscation and Optimization

We also analyzed the impact of code obfuscation and optimization on the analysis performance. In order to measure this impact, we analyzed the Bouncy Castle library obfuscated with different *ProGuard* settings and compared the analysis results with the original version. For this test, we used the two analysis plugins for symmetric- and asymmetric cryptography detection.

Table 9.7 compares the results for the normal library as well as for the obfuscated one. For obfuscation, the default optimized Android ProGuard configuration provided by the Android SDK has been used. Only shrinking and method name obfuscation has been disabled in order to be able to better compare the results. Listing 9.5 shows the full ProGuard configuration. It can be observed, that more cryptographic methods are detected for the obfuscated library. Furthermore, the total number of analyzed methods is larger as well. Keep in mind, that the number of analyzed methods depends on the number of opcodes of the given method. Since ProGuard performs various optimizations, including method inlining, cryptographic code has been inlined in some additional methods, instead of performing the method call of the original version. Thus, more cryptographic code is detected since the same cryptographic code is found in multiple methods. The discrepancy in the total number of analyzed methods can also be explained by method inlining. Since the number of opcodes of selected methods has been increased due to inlining, methods that were below the threshold in the normal version are now above the threshold, and thus, analyzed by the framework.

Other than that, the analysis results are very similar. Hence, obfuscation does not have a significant impact on the analysis performance. We also performed an additional evaluation, where class- and method name obfuscation has been enabled. The same number of methods have been classified as cryptographic- and normal code. Since we do not consider class- and method names as features, this is the expected behavior. Thus, obfuscation does not have a significant impact on the analysis performance.

9.6 Performance

This section is dedicated to the performance of the *Semantic Pattern Analysis* conducted within the *Semdroid* framework. For this performance evaluation, we randomly selected *50 Android applications* from various categories on Google Play, found in the ”Top Free” lists of their respective

```
—dontshrink
—optimizations !code/simplification/arithmetic,!code/simplification/cast,
    !field/*,!class/merging/*
-optimizationpasses 5
-allowaccessmodification
-dontobfuscate
-dontusemixedcaseclassnames
-keepattributes *Annotation*
-dontpreverify
-verbose
-dontwarn android.support.**


-keep public class com.google.vending.licensing.ILicensingService

-keep public class com.android.vending.licensing.ILicensingService

# keep setters in Views so that animations can still work.
# see http://proguard.sourceforge.net/manual/examples.html#beans
-keepclassmembers public class * extends android.view.View {
    void set*(***);
    *** get*();
}

# We want to keep methods in Activity that could be used in the XML
    attribute onClick
-keepclassmembers class * extends android.app.Activity {
    public void *(android.view.View);
}

-keep class * extends android.os.Parcelable {
    public static final android.os.Parcelable$Creator *;
}

-keepclassmembers class **.R$* {
    public static <fields>;
}

# Also keep - Enumerations. Keep the special static methods that are
    required in
# enumeration classes.
-keepclassmembers enum  * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

# Keep names - Native method names. Keep all native class/method names.
-keepclasseswithmembers,allowshrinking class * {
    native <methods>;
}
```

Listing 9.5: ProGuard configuration used for obfuscation. These settings are from the default Android ProGuard settings (optimized version). Only shrinking has been disabled, as well as method name obfuscation.

Analysis	Label	Normal library	Obfuscated library
Symmetric cryptography	Cryptography	293	344
	Normal	1513	1570
	Total	1806	1914
Asymmetric cryptography	Cryptography	27	32
	Normal	6941	7026
	Total	6968	7058

Table 9.7: Obfuscation results comparison containing the number of detected cryptographic methods for the normal Bouncy Castle library and the obfuscated version.

Analysis	AP	FLG	SPT	ML	Total
App-instance-based	984	30	2	1	1017
Class-instance-based	984	30	24	23	1061
Method-instance-based	984	115	57	53	1209
All 3 combined (expected)	984	175	83	77	1319
All 3 combined (measured)	984	174	71	69	1298

Table 9.8: Performance overview: PC. The analyses have been conducted on a PC using an Intel® Core™ i7-920 (2.66 GHz) with 6 GB of RAM, running Windows 8.1. All values are given in milliseconds **[ms]**.

categories. These 50 applications have a total size of 501 mb, resulting in an average application size of roughly *10 mb*.

We evaluated three different analysis plugins with a different granularity: app-, class- and method-instance-based. These three analysis plugins have been evaluated separately, and bundled together in a single test suite. The first analysis utilizes a single feature layer containing solely app instances. Since this analysis operates on the app level, we only have a single instance per application. The class-based analysis creates more instances, one per analyzed Java class. The average number of classes included in one of the 50 test applications is 1157. Keep in mind that the analysis ignores small classes with less than 30 opcodes. Similarly, the third analysis based on method instances creates 2006 method instances on average. Again, the analysis only considers methods with more than 30 opcodes.

Applications can be analyzed on a personal computer, or directly on an Android device. We analyzed all 50 applications separately on both a PC and an Android device and recorded detailed timing information. Then, we averaged these results to get an estimate of the average analysis performance.

Table 9.8 shows the average timings for the PC-based analysis. All timings are given in milliseconds. The evaluation has been broken down into several core stages of the analysis process: First, the App object parsing time (**AP**) is given, followed by the timings required for feature layer generation (**FLG**), Semantic Pattern Transformation (**SPT**), and machine learning (**ML**). The total required time, i.e., the sum of all core timings, is listed as well. Since the App parsing process is identical for all analysis plugins, the time required for this process has been averaged once and then used for all AP values in Table 9.8.

Analysis	AP	FLG	SPT	ML	Total
App-instance-based	16827	598	10	15	17450
Class-instance-based	16827	548	592	322	18289
Method-instance-based	16827	2176	1794	843	21640
All 3 combined (expected)	16827	3322	2396	1180	23725
All 3 combined (measured)	16827	3963	2400	1190	24380

Table 9.9: Performance overview: on-device analysis. The analyses have been conducted on a Google Nexus 5 (Qualcomm® Snapdragon™ 800) running Android 4.4.2. All values are given in milliseconds **[ms]**.

The fastest analysis is the app-instance-based analysis. Since this analysis generates a single instance per application, the Semantic Pattern Transformation and the machine learning process have to be performed only once per application.

The runner-up is the class-based analysis. The feature layer generation takes roughly the same time as for the app-instance-based analysis. Since there are 1157 class instances on average that have to be converted and analyzed, both the SPT and ML process take considerably longer compared to the app-instance case. These two processes do not exactly need 1557 times more time due to the fact that the analysis plugins have different Semantic Pattern networks and machine learning models. One difference between these two analyses is, that the features are split up between multiple instances for the clas-based analysis, whereas the other analysis utilizes a single app instance. This also means that app instances can be considerably larger than a single class instance, which explains the fact that SPT and ML do not require 1557 times more time. Feature layer generation takes approximately the same time for both analysis plugins, since approximately the same features have to be traversed.

The method-based analysis takes roughly 150 ms longer than the class-based analysis. In detail, the feature layer generation takes almost four times longer, requiring 115 ms on average. This is the case because we have nearly twice as many instances (2006 as opposed to 1157 on average). For each of these instances, separate opcode and local variable histograms have to be calculated and averaged. In addition, the method call inclusion depth has been set to 2, meaning that for each method, the opcodes of nested methods have to be traversed as well. Since double the number of instances have to be converted to Semantic Patterns and classified by the machine learning framework, it takes approximately double the time.

The most time-consuming part of the analysis process is the *App parsing*. Since the whole application has to be parsed and the whole application structure has to be reconstructed, this process requires 984 ms on average – it accounts for almost *97%* of the analysis time required by the app-instance-based analysis, or for 81% for the method-instance-based analysis. Due to the architecture of the Semdroid framework (Section 4.1), this App parsing only has to be performed once, no matter how many analysis plugins are included in the test suite. Hence, if we combine multiple analysis, the analysis time does not significantly increase. If we combine all three analysis plugins, the whole process takes 1319 ms, only *27%* longer than for the fastest analysis alone. This measured time is also very close to the expected value, which can be calculated by adding the FLG, SPT, and ML times for all three analysis plugins and the App parsing process (AP) once.

Table 9.9 lists the same timings for on-device analysis. Since the computational power of

Analysis	Components	PC [ms]	On-device [ms]	Times slower
App-instance-based	1	1017	17450	17.15
Class-instance-based	1157	1061	18289	17.24
Method-instance-based	2006	1209	21640	17.90
All 3 combined	4392	1298	24380	18.78

Table 9.10: Performance comparison for PC-based and on-device analysis. “Components” gives the average number of analyzed application components.

a mobile device is significantly lower than the power of a personal computer, the analysis takes much longer, ranging from 17.5 to 21.6 seconds depending on the analysis. The observations from the PC results also apply to the on-device results. The App parsing is again the most time-consuming operation, accounting for approximately 97% of the required time for the app-based analysis – very similar to the PC-based case. Again, combining analyses is a very good idea; all three analysis plugins combined only require 24.4 seconds.

Table 9.10 compares the results for both the PC-based and the on-device analysis process. On average, analyzing applications directly on an Android device takes 17-18 times longer than on a PC. In addition, Table 9.10 also lists the average number of analyzed application components.

Chapter 10

Conclusions and Outlook

Smartphones and mobile devices in general are becoming an integral part of our daily lives. Users entrust their devices with sensitive data, like login credentials, credit card details, or private pictures. Thus, it is very important to protect this information, to keep this data secure.
By thoroughly analyzing the applications users have installed on their Android devices, the security of this data can be improved. Potentially dangerous or unwanted functionality can be detected – in the worst case, even malicious applications – and the user can be warned.

In this thesis, we presented *Semdroid*, a powerful static Android application analysis framework. With this framework, it is possible to analyze Android application packages and to pinpoint certain functionality within these applications. The analysis plugins used for this purpose can be based on any static analysis technique. Semdroid performs application pre-processing, manages all analysis plugins, collects their results, and performs various post-processing steps. Furthermore, Semdroid provides a training- and evaluation framework that can be used to create new analysis plugins and to evaluate their performance.

The Semdroid framework can be used on a personal computer or directly on any Android device. If the framework is used on a personal computer, detailed HTML and XML analysis reports will be generated. For on-device analysis, a special Semdroid Android application is available, which allows to analyze installed applications and to examine the analysis results.

The proposed new *Semantic Pattern Analysis*, which allows to assess various application functionality, can be employed within the Semdroid framework. This analysis approach extracts so-called feature layers from the application under analysis. These layers contain a number of instances, where each one represents a component of the application, like a class or a single method. Each of these instances holds a list of features representative for the component. Possible features include, amongst others, opcodes, local variables, and method calls. The *Semantic Pattern Transformation* developed by Teufl [52] is then applied on the feature layers to transform the instances to Semantic Patterns – vectors containing simple double values. These patterns are then supplied to machine learning algorithms that label the components according to their functionality.

The first two analysis plugins presented in this thesis target cryptographic code and yielded great results. They are based on the *Semantic Pattern Analysis* and are able to pinpoint symmetric- and asymmetric-key cryptography included in Android application packages.
Then, the SMS-handling capabilities of applications can be detected by a third analysis plugin. By using the Semantic Pattern Analysis, it is possible to detect broadcast receivers that handle incoming SMS messages.

All analysis plugins have been thoroughly evaluated and we also showed that code obfuscation and optimization does not have a negative impact on the analysis performance. Furthermore, we also compared the performance for on-device and PC-based application analysis.

The *Semantic Pattern Analysis* and the *Semdroid* framework offer many possibilities that could be addressed in future work. We want to give a short overview of what some of these ideas could be:

- **Additional analysis plugins**
  In this thesis, we created several analysis plugins for different use cases. But there are many more applications for the Semantic Pattern Analysis; in theory, any functionality can be targeted, like boot receivers, UI code, background services.
  The Semdroid framework itself is not limited to analysis approaches based on the Semantic Pattern Analysis. Hence, different analysis approaches could also be integrated into the Semdroid framework.

- **Malware detection**
  Known malicious code could be targeted as well. An advantage of the *Semantic Pattern Analysis* is, that it should be able to detect different variants of a given functionality – meaning that a slightly modified malicious code should be detected as well.

- **Anomaly detection**
  New analysis plugins could also be based upon anomaly detection, also called outlier detection, as described in Section 6.7.2.

- **Deep learning and multilayer strategies**
  Current analysis plugins only utilize a single feature layer. In future work, multiple feature layers could be used as well. The resulting layers of Semantic Patterns created by the Semantic Pattern Transformation can then be supplied to machine learning algorithms, which could be based on deep learning strategies.

- **Additional feature types**
  Besides the currently used features found in the Dalvik executable and the Android manifest, additional feature sources could be used as well. These new features can, for instance, be found in various resource files included in the Android application package, like in layout files, used drawables, or defined strings. Native code could also be added to this feature pool; the native instructions could be treated similar to the currenty used Dalvik opcodes. Furthermore, other metadata, like the application description found on Google Play could be considered as a feature source as well.

- **Feature order**
  Currently, the order of the features is not considered. As already explained, this brings some advantages, as well as disadvantages. Future work could consider this feature order by using a modified version of the *Semantic Pattern Transformation*.

- **Analysis granularity**
  Current analysis plugins operate on one of these three levels: methods, classes, or apps. New plugins could also use any other scope. This scope could, for example, be more fine grained by analyzing parts of methods separately, or more course grained by considering Java packages as a whole.

- **Branches**
  Usually the Java code of an application has different branches (execution paths). It could

be possible to analyze each of these branches separately by using a modified variant of the Semantic Pattern Analysis. For example, if the code of the application under analysis includes an `if` operation, both of these branches could be analyzed separately.

- **Code usage**
  Currently, the analysis process checks all classes and methods included in the Android application package – but the code is not necessarily used by the application. For example, if an application includes a cryptographic library, but only utilizes a single encryption algorithm, all other encryption algorithms included in the library will also be detected – even if they are never actually used by the application itself. Future work could include additional checks that indicate whether a given component is used.

- **Combining analysis approaches**
  For now, all analysis plugins operate independently. Future work could look at the results of multiple analysis plugins and make automated assumptions based on the analysis results. A concrete example for this approach would be malware. If we know that a certain malware family requires a boot receiver, an SMS catcher, and some other known code *X*, three separate analysis plugins that detect these three code fragments can be employed. If all these code fragments are found, the application under analysis could have malicious intentions and can thus be labeled as malware.

- **Deeper Android integration**
  The analysis framework could also be deeply integrated into the Android operating system. If a suspicious application is to be installed on the device, the installation process can be denied in order to protect the device. Similarly, if the user wants to install an insecure application, like a password manager that does not sufficiently protect the login credentials, the user can be alerted that the application might be insecure. The Semdroid framework already provides an API that allows external applications to analyze applications and to retrieve the analysis results.

With *Semdroid* and the *Semantic Pattern Analysis* a very flexible framework for analyzing Android applications is available. The performance of the analysis plugins evaluated in this thesis exceeded our expectations. These analysis plugins allow to detect symmetric- and asymmetric-key cryptography as well as SMS-handling capabilities. New analysis plugins can easily be added and evaluated. In future work, the framework could be extended to support deep learning, as well as additional feature types, like native code. Since on-device analysis is feasible, a deeper Android integration could provide additional security for the Android platform.

Appendix A

Opcode Groups

Original Opcode	Symmetric Cryptography	Asymmetric Cryptography	SMS Handling
THROW_VERIFICATION_ERROR	-	-	-
EXECUTE_INLINE	-	-	-
INVOKE_SUPER_QUICK	-	-	-
INVOKE_VIRTUAL_QUICK	-	-	-
IGET_QUICK	-	-	GET
IPUT_QUICK	-	-	PUT
NOP	-	-	-
MOVE	-	MOVE	MOVE
MOVE_RESULT	-	MOVE	MOVE
MOVE_EXCEPTION	-	MOVE	MOVE
RETURN_VOID	-	-	-
RETURN	-	-	-
CONST	-	-	-
CONST_STRING	-	-	-
CONST_CLASS	-	-	-
MONITOR_ENTER	-	-	-
MONITOR_EXIT	-	-	-
CHECK_CAST	-	-	-
INSTANCE_OF	-	-	-
ARRAY_LENGTH	-	ARRAY_LENGTH	-
NEW_INSTANCE	-	-	-
NEW_ARRAY	-	-	-
FILLED_NEW_ARRAY	-	-	-
FILL_ARRAY_DATA	-	-	-
THROW	-	-	-
GOTO	-	-	-
PACKED_SWITCH	-	-	-
SPARSE_SWITCH	-	-	-
CMPL	-	CMP	CMP
CMPG	-	CMP	CMP
CMP	-	CMP	CMP
IF_EQ	-	IF	IF
IF_NE	-	IF	IF
IF_LT	-	IF	IF
IF_GE	-	IF	IF

IF_GT	-	IF	IF
IF_LE	-	IF	IF
IF_EQZ	-	IF	IF
IF_NEZ	-	IF	IF
IF_LTZ	-	IF	IF
IF_GEZ	-	IF	IF
IF_GTZ	-	IF	IF
IF_LEZ	-	IF	IF
AGET	-	-	GET
APUT	-	-	PUT
IGET	-	-	GET
IPUT	-	-	PUT
SGET	-	-	GET
SPUT	-	-	PUT
INVOKE_VIRTUAL	-	-	-
INVOKE_SUPER	-	-	-
INVOKE_DIRECT	-	-	-
INVOKE_STATIC	-	-	-
INVOKE_INTERFACE	-	-	-
NEG	NEG	NEG	NEG
NOT	NOT	NOT	NOT
X_TO_Y	X_TO_Y	X_TO_Y	X_TO_Y
ADD	ADD	ADD	ADD
SUB	SUB	SUB	SUB
MUL	MUL	MUL	MUL
DIV	DIV	DIV	DIV
REM	REM	REM	REM
AND	AND	AND	AND
OR	OR	OR	OR
XOR	XOR	XOR	XOR
SHL	SHL	SHL	SHL
SHR	SHR	SHR	SHR
USHR	USHR	USHR	USHR
ADD_INT_LIT_X	ADD_INT_LIT_X	ADD_INT_LIT_X	ADD_INT_LIT_X
RSUB_INT_LIT_X	RSUB_INT_LIT_X	RSUB_INT_LIT_X	RSUB_INT_LIT_X
MUL_INT_LIT_X	MUL_INT_LIT_X	MUL_INT_LIT_X	MUL_INT_LIT_X
DIV_INT_LIT_X	DIV_INT_LIT_X	DIV_INT_LIT_X	DIV_INT_LIT_X
REM_INT_LIT_X	REM_INT_LIT_X	REM_INT_LIT_X	REM_INT_LIT_X
AND_INT_LIT_X	AND_INT_LIT_X	AND_INT_LIT_X	AND_INT_LIT_X
OR_INT_LIT_X	OR_INT_LIT_X	OR_INT_LIT_X	OR_INT_LIT_X
XOR_INT_LIT_X	XOR_INT_LIT_X	XOR_INT_LIT_X	XOR_INT_LIT_X
SHL_INT_LIT_X	SHL_INT_LIT_X	SHL_INT_LIT_X	SHL_INT_LIT_X
SHR_INT_LIT_X	SHR_INT_LIT_X	SHR_INT_LIT_X	SHR_INT_LIT_X
USHR_INT_LIT_X	USHR_INT_LIT_X	USHR_INT_LIT_X	USHR_INT_LIT_X

Table A.1: Opcode groups used for the analysis process. The left column lists the original opcodes (pre-grouped according to *dex2jar* [9]). Then, the opcode group names are listed for all analysis plugins.

Bibliography

[1] *Androguard*. http://code.google.com/p/androguard/. 2013.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN 0387310738.

[3] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. "Crowdroid: Behavior-based Malware Detection System for Android". In: *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*. SPSM '11. Chicago, Illinois, USA: ACM, 2011, pages 15–26. ISBN 978-1-4503-1000-0. doi:10.1145/2046614.2046619. http://doi.acm.org/10.1145/2046614.2046619.

[4] Carlos A Castillo. "Android malware past, present, and future". In: *White Paper of McAfee Mobile Security Working Group* (2011). http://www.mcafee.com/us/resources/white-papers/wp-android-malware-past-present-future.pdf.

[5] Chih-Chung Chang and Chih-Jen Lin. "LIBSVM: A library for support vector machines". In: *ACM Transactions on Intelligent Systems and Technology* 2 (3 2011). Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm, 27:1–27:27.

[6] Erika Chin et al. "Analyzing Inter-application Communication in Android". In: *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*. MobiSys '11. Bethesda, Maryland, USA: ACM, 2011, pages 239–252. ISBN 978-1-4503-0643-0. doi:10.1145/1999995.2000018. http://doi.acm.org/10.1145/1999995.2000018.

[7] Corinna Cortes and Vladimir Vapnik. "Support-Vector Networks". In: *Machine Learning* 20.3 (Sept. 1995), pages 273–297. ISSN 0885-6125. doi:10.1023/A:1022627411411. http://dx.doi.org/10.1023/A:1022627411411.

[8] P. Deutsch and J-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950 (Informational). Internet Engineering Task Force, May 1996. http://www.ietf.org/rfc/rfc1950.txt.

[9] *dex2jar*. http://code.google.com/p/dex2jar/. 2013.

[10] Gianluca Dini et al. "MADAM: A Multi-level Anomaly Detector for Android Malware". In: *Computer Network Security*. Edited by Igor Kotenko and Victor Skormin. Volume 7531. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pages 240–253. ISBN 978-3-642-33703-1. doi:10.1007/978-3-642-33704-8_21. http://dx.doi.org/10.1007/978-3-642-33704-8_21.

[11] *DroidBox*. http://code.google.com/p/droidbox/. 2013.

[12] Manuel Egele et al. “An Empirical Study of Cryptographic Misuse in Android Applications”. In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*. CCS ’13. New York, New York, USA: ACM Press, Nov. 2013, pages 73–84. doi:10.1145/2508859.2516693. `http://dl.acm.org/citation.cfm?id=2508859.2516693`.

[13] William Enck et al. “TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones”. In: *Design*. OSDI’10 49.4 (2010). Edited by Remzi H Arpaci-Dusseau and Brad Chen, pages 1–6. ISSN 03601315. `http://static.usenix.org/events/osdi10/tech/full_papers/Enck.pdf`.

[14] 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*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pages 50–61. ISBN 978-1450316514. doi:10.1145/2382196.2382205. `http://doi.acm.org/10.1145/2382196.2382205`.

[15] 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, pages 627–638. ISBN 978-1-4503-0948-6. doi:10.1145/2046707.2046779. `http://doi.acm.org/10.1145/2046707.2046779`.

[16] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. “SCanDroid: Automated security certification of Android applications”. In: *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa* (2009).

[17] Martin Georgiev et al. “The Most Dangerous Code in the World: Validating SSL Certificates in Non-browser Software”. In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS ’12. Raleigh, North Carolina, USA: ACM, 2012, pages 38–49. ISBN 978-1450316514. doi:10.1145/2382196.2382204. `http://doi.acm.org/10.1145/2382196.2382204`.

[18] M. Ghorbanzadeh et al. “A neural network approach to category validation of Android applications”. In: *Computing, Networking and Communications (ICNC), 2013 International Conference on*. 2013, pages 740–744. doi:10.1109/ICCNC.2013.6504180.

[19] Google Inc., Android Developers. *Android Permissions*. 2013. `http://developer.android.com/guide/topics/security/permissions.html`.

[20] Google Inc., Android Developers. *App Manifest*. 2013. `http://developer.android.com/guide/topics/manifest/manifest-intro.html`.

[21] Google Inc., Android Developers. *App Resources*. 2013. `http://developer.android.com/guide/topics/resources/index.html`.

[22] Google Inc., Android Developers. *Application Fundamentals*. 2013. `http://developer.android.com/guide/components/fundamentals.html`.

[23] Google Inc., Android Developers. *Fragments*. 2013. `http://developer.android.com/guide/components/fragments.html`.

[24] Google Inc., Android Developers. *Intents and Intent Filters*. 2013. `http://developer.android.com/guide/components/intents-filters.html`.

[25] Michael Grace et al. “RiskRanker: Scalable and Accurate Zero-day Android Malware Detection”. In: *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*. MobiSys ’12. Low Wood Bay, Lake District, UK: ACM, 2012, pages 281–294. ISBN 978-1-4503-1301-8. doi:10.1145/2307636.2307663. `http://doi.acm.org/10.1145/2307636.2307663`.

[26] Sheran Gunasekera. *Android Apps Security*. 1st. Berkely, CA, USA: Apress, 2012. ISBN 1430240628.

[27] Mark Hall et al. “The WEKA Data Mining Software: An Update”. In: *SIGKDD Explorations* 11.1 (Nov. 2009), pages 10–18. ISSN 1931-0145. doi:10.1145/1656274.1656278. `http://doi.acm.org/10.1145/1656274.1656278`.

[28] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648 (Proposed Standard). Internet Engineering Task Force, Oct. 2006. `http://www.ietf.org/rfc/rfc4648.txt`.

[29] Scott Kirkpatrick and Erich P Stoll. “A very fast shift-register sequence random number generator”. In: *Journal of Computational Physics* 40.2 (1981), pages 517–526.

[30] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0201896842.

[31] S. Leontiev and D. Shefanovski. *Using the GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms with the Internet X.509 Public Key Infrastructure Certificate and CRL Profile*. RFC 4491 (Proposed Standard). Internet Engineering Task Force, May 2006. `http://www.ietf.org/rfc/rfc4491.txt`.

[32] Scott Main and David Braun. *Getting Your SMS Apps Ready for KitKat*. 2013. `http://android-developers.blogspot.in/2013/10/getting-your-sms-apps-ready-for-kitkat.html`.

[33] Reto Meier. *Professional Android 4 Application Development*. Wrox Professional Guides. Wiley, 2012. ISBN 1118102274.

[34] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. 1st. Boca Raton, FL, USA: CRC Press, Inc., 1996. ISBN 0849385237.

[35] Thomas M. Mitchell. *Machine Learning*. New York, NY, USA: McGraw-Hill, 1997. ISBN 0070428077.

[36] David Naccache and Jacques Stern. “A new public key cryptosystem based on higher residues”. In: *Proceedings of the 5th ACM conference on Computer and communications security*. ACM. 1998, pages 59–66.

[37] National Institute of Standards and Technology. “Advanced Encryption Standard (AES)”. In: (2001).

[38] National Institute of Standards and Technology. *Descriptions of SHA-256, SHA-384, and SHA-512*. 2001. `http://csrc.nist.gov/groups/STM/cavp/documents/shs/sha256-384-512.pdf`.

[39] Godfrey Nolan. *Decompiling Android*. Apress, 2012. ISBN 1430242485.

[40] Oracle. *jarsigner - JAR Signing and Verification Tool*. 2013. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html.

[41] Colin Percival and Simon Josefsson. “The scrypt Password-Based Key Derivation Function”. In: (2012).

[42] Niels Provos and David Mazieres. “A Future-Adaptable Password Scheme.” In: 1999.

[43] B. Sanz et al. “On the automatic categorisation of android applications”. In: *Consumer Communications and Networking Conference (CCNC), 2012 IEEE*. 2012, pages 149–153. doi:10.1109/CCNC.2012.6181075.

[44] A.-D. Schmidt et al. “Static Analysis of Executables for Collaborative Malware Detection on Android”. In: *Communications, 2009. ICC ’09. IEEE International Conference on*. 2009, pages 1–5. doi:10.1109/ICC.2009.5199486.

[45] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. 2nd. Wiley, 1996, pages I–XXIII, 1–758. ISBN 0471117099.

[46] Asaf Shabtai, Yuval Fledel, and Yuval Elovici. “Automated Static Code Analysis for Classifying Android Applications Using Machine Learning”. In: *Proceedings of the 2010 International Conference on Computational Intelligence and Security*. CIS ’10. Washington, DC, USA: IEEE Computer Society, 2010, pages 329–333. ISBN 978-0-7695-4297-3. doi:10.1109/CIS.2010.77. http://dx.doi.org/10.1109/CIS.2010.77.

[47] Asaf Shabtai et al. “"Andromaly": a behavioral malware detection framework for android devices”. In: *J. Intell. Inf. Syst.* 38.1 (Feb. 2012), pages 161–190. ISSN 0925-9902. doi:10.1007/s10844-010-0148-x. http://dx.doi.org/10.1007/s10844-010-0148-x.

[48] *smali*. http://code.google.com/p/smali/. 2013.

[49] N.P. Smart. *Cryptography: An Introduction*. 3rd. 2012. http://www.cs.bris.ac.uk/~nigel/Crypto_Book/.

[50] James Steele and Nelson To. *The Android Developer’s Cookbook: Building Applications with the Android SDK*. Addison-Wesley Professional, 2010. ISBN 0321741234.

[51] Douglas R. Stinson. *Cryptography: Theory and Practice, Third Edition*. Taylor & Francis, 2005. ISBN 1584885084.

[52] Peter Teufl. “Semantic Patterns”. PhD thesis. Graz University of Technology, 2012.

[53] Peter Teufl and Günther Lackner. “RDF Data Analysis with Activation Patterns”. In: *10th International Conference on Knowledge Management and Knowledge Technologies 1–3 September 2010, Messe Congress Graz, Austria*. Edited by Klaus Tochtermann und Hermann Maurer. Journal of Computer Science. 2010, pages 18–18.

[54] Peter Teufl, Herbert Leitold, and Reinhard Posch. “Semantic Pattern Transformation”. In: *Proceedings of the 13th International Conference on Knowledge Management and Knowledge Technologies*. Edited by ACM. ACM, 2013, pages –.

[55] Peter Teufl et al. “Android Market Analysis with Activation Patterns”. In: *MOBISEC*. 2011.

[56] The Android Open Source Project. *Bytecode for the Dalvik VM*. 2013. http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[57] The Android Open Source Project. *Dalvik Executable Format*. 2013. `http://source.android.com/devices/tech/dalvik/dex-format.html`.

[58] The Android Open Source Project. *Dalvik VM Instruction Formats*. 2013. `https://source.android.com/devices/tech/dalvik/instruction-formats.html`.

[59] Wade Trappe and Lawrence C. Washington. *Introduction to Cryptography with Coding Theory*. 2nd. Pearson Prentice Hall, 2005. ISBN 0131862391.

[60] Ian H. Witten, Eibe Frank, and Mark A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Third Edition. Morgan Kaufmann, 2011. ISBN 0123748569.

[61] Dong-Jie Wu et al. "DroidMat: Android Malware Detection through Manifest and API Calls Tracing". In: *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*. 2012, pages 62–69. doi:10.1109/AsiaJCIS.2012.18.

[62] Thomas Wu et al. "Srp-6: Improvements and refinements to the secure remote password protocol". In: *Submission to IEEE P* 1363 (2002).

[63] Thomas D Wu et al. "The Secure Remote Password Protocol." In: *NDSS*. Volume 98. 1998, pages 97–111.

[64] Lok Kwong Yan and Heng Yin. "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis". In: *Proceedings of the 21st USENIX Security Symposium*. 2012.

[65] Thomas Zefferer et al. "Power Consumption-based Application Classification and Malware Detection on Android Using Machine-Learning Techniques". In: *FUTURE COMPUTING 2013: The Fifth International Conference on Future Computational Technologies and Applications*. IARIA, 2013, pages 26–31.

[66] Min Zhao et al. "AntiMalDroid: An Efficient SVM-Based Malware Detection Framework for Android". In: *Information Computing and Applications*. Edited by Chunfeng Liu, Jincai Chang, and Aimin Yang. Volume 243. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2011, pages 158–166. ISBN 978-3-642-27502-9. doi:10.1007/978-3-642-27503-6_22. `http://dx.doi.org/10.1007/978-3-642-27503-6_22`.

[67] Min Zhao et al. "RobotDroid: A Lightweight Malware Detection Framework On Smartphones." In: 2012, pages 715–722. `http://dx.doi.org/10.4304/jnw.7.4.715-722`.

[68] Yajin Zhou and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution". In: *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pages 95–109. doi:10.1109/SP.2012.16. `http://dx.doi.org/10.1109/SP.2012.16`.

[69] Bartosz Zoltak. "VMPC one-way function and stream cipher". In: *Fast Software Encryption*. Springer. 2004, pages 210–225.